#268. Missing Element

- Given an array nums containing n distinct numbers in the range [0,n] return the only number in the range that is missing.

| 3 | 0 | 1 |

// 2

| 9 | 6 | 4 | 2 | 3 | 5 | 7 | d | 1 |

// 8

// Brute force :

| 0 | 1 | 3 | 4 |

N = 4

i=0 , check for 0 is present
i=1 , check for 1 is present
i=2 , check for 2, is present X return 2.

- run a loop from 0 → N, for each value b/w 0 to N check if that no. is present in array or not. using linear search.

```
for(i=0 → N){
  flag = 0;
for(j=0 → N-1){
  if(a[j] == i){
    flag = 1;
    break;
  }
}
if(flag == 0) return i;
}
return -1;
```

time complexity: O(n²)

space complexity: O(1)

// Better solution (hashing)

- use a hash array to store the frequency of each element of the array.
- return the element having frequency 0.

0	5	6	4	3	2

6	1
5	1
4	1
3	1
2	1
1	0
0	1

return

```
hash[n+1] = {0};
for (i=0 → n){
  hash[a[i]]++;
}

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

return -1
```

time complexity: $O(n) + O(n+1)$

space complexity: $O(n+1)$

// Optimal Approach:

- Summation: sum1 = sum of numbers 0 → N
- find arraySum:
- Subtract the arraySum from NSum, the difference will be the answer.

0	2	5	3	4

nSum = 5 × (5+1)/2 = 15

arraySum = 0+2+5+3+4 = 12

ans = 15 - 12 = 3 ✓

```
nsum = n*(n+1)/2;
sum = 0;
for(i=0 -> n){
    sum += a[i];
}
return nsum - sum;
```

time complexity: O(n)

space complexity: O(1)

// Optimal Solution: (XOR)

- using XOR,

$a \wedge a = 0$

$a \wedge 0 = a$

- find XOR of array elements. = xor1
- find XOR of numbers from 0 to N. = xor2
- find xor1 ^ xor2, it will give the missing number.

xor1 = 1^2^3^4^5^6^7^8^9

xor2 = 1^2^3^5^6^7^8^9

Ans = 4

```
xor1 = 0, xor2 = 0;
for(i=0 -> n){
    xor2 = xor2^a[i];
    xor1 = xor1^(i+1);
}
xor1 = xor1^N;
return xor1^xor2;
```

time complexity: O(n)

space complexity: O(1)

#485. Max Consecutive Ones

- Given a binary array nums, return the maximum number of consecutive 1's in the array.

[1,1,0,1,1,1] // 3
[1,0,1,1,0,1] // 2

// Brute force:

- initialize maxi = 0, cnt = 0.
- iterate over the array, if you encounter 1, cnt++, else, update maxi with max(maxi, cnt) & cnt = 0.

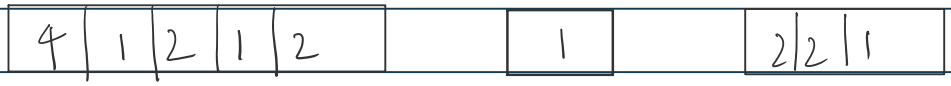
```
maxi = 0, cnt = 0;
for(i=0 → n){
 if(a[i] == 1) cnt++;
else{ maxi = max(maxi, cnt);
      cnt = 0;
 }
}
return maxi;
```

time complexity : O(n)

space complexity : O(1)

#136. Single Number

- Given an array of integers nums, every element appears twice except one.
- Return the number appearing once:



// 4 // 1 // 1

// Brute force

- initialize cnt = 0, iterate over the array for each element,
- for each element cnt the occurrence, if cnt == 1 return the element.

```
for (i = 0 → n)
    num = a[i]
    cnt = 0;
    for (j = 0 → n) {
        if (a[j] == num)
            if (cnt == 2) break;
                cnt++;
        }
    }
    if (cnt == 1) return num;
}
```

time complexity: $O(n^2)$

space complexity: $O(1)$

// Better Approach (Hashing)

- find the maxi in the given array, initialize a hash array of size maxi+1, with all elements set as 0.
- count the frequency of each element using hashing
- iterate over the hash & return the element with frequency = 1.

```
for(i=0 → n){
  maxi = max(maxi, a[i]);
}
hash[maxi+1] = {0};
for(i=0 → n){
  hash[a[i]]++;
}
for(i=0 → n){
  if(hash[a[i]] == 1)
    return a[i];
}
return 0;
```

time complexity: O(n)

space complexity: O(n)

Note: only applicable if array contains only the integers

// Better Solution - II

- instead of hash, use a map<int, int>.
- using a loop iterate over the array store the element along with its frequency in the map.
- use another loop to iterate over the map, return the element with frequency = 1.

```
map<int, int> ;
for(i=0 → n){
    map[ a[i] ]++;
}
for(auto it : map){
    if(it.second == 1)
        return it.first;
}
return -1;
```

time complexity: $O(n \log m) + O(m)$, $m = n/2 + 1$

space complexity: $O(m) = O(\frac{n+1}{2})$

// Optimal Solution (XOR)

- XOR every element of the array, XOR will cancel out duplicates and the remaining ans would be the single occurring element.

```
ans = 0;
for (i=0 → n)
    ans = ans ^ a[i];
return ans;
```

time complexity: $O(n)$

space complexity: $O(1)$

Longest Subarray with given sum K (only positives)

- Given an array and a sum k, return the length of the longest subarray with sum equal to k.

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

// ans = 3

// Brute force :

- Find all the subarrays possible in the given array.
- Find the sum of the current subarray, if sum == k update the max length.
- Algorithm:
  - run a loop from i = 0 → n
  - run a nested loop, j = i → n
  - calculate the sum of the subarray a[i → j], if sum == k, update length of max subarray.

```
//  ans = 0;
    for(i=0 ⟶ n){
        for(j=i ⟶ n){
            sum = 0;
            for(k=i ⟶ j){
                sum += a[k];
            }
            if(sum == k)
                ans = max(ans, j-i+1);
```

time complexity: ≈ O(n³)
Space complexity: O(1)

if (sum == k)
ans = max(ans, j-i+1);
}
} return ans;

// Better Solution.

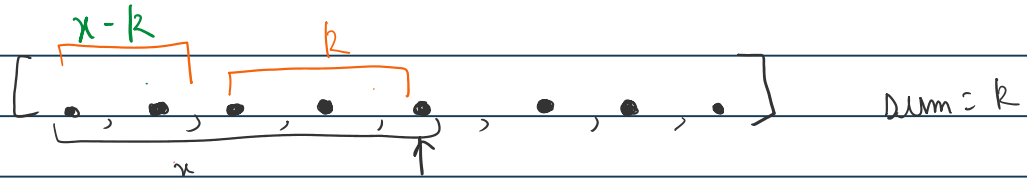
- instead of using a third loop to find the sum of the subarray, just keep on adding the jth element in the second loop.

```
ans = 0;
for(i=0 ⟶ n){
    sum = 0;
    for(j = i ⟶ n){
        sum += a[j];
        if(sum == k)
            ans = max(ans, j-i+1);
    }
}
return ans;
```

time complexity: $O(n^2)$

space complexity: $O(1)$

// Better Solution



- Consider the subarray with the last element as the current element. prefix-sum = x

→ if there exists a subarray with sum k as (.) as the last element

- Use a hash map to store the subarray with sums.
- Use a loop & start from the first element, keep calculating the sum.
  - store the sum as the key, & i as the value.

- if the pre-fix sum == k, calculate the of the subarray.
- if the pre-fix sum > k, calculate prefix-sum − k element.
  - if the you don't find the sum = prefix − k (sum) in the hash, store the prefix-sum in map.
- if sum = prefix-k, is present in the hash, calculate the length

  length = (i − (prefix − k).value) from hash. (sum)

update the length: ⟸

19, 9
16, 8
14, 7
10, 6
9, 5
8, 4
7, 3
6, 2
3, 1
1, 0

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

i = 0 presum = 1, len = 0

i = 1 presum = 3 = k, len = 2

i = 2 presum = 6 > k, len = 2 − 1 = 1 < 2, len = 2

i = 3 pre-sum = 7, len = 2

i = 4, pre-sum = 8, len = 2

i = 5, pre-sum = 9, 6 is present

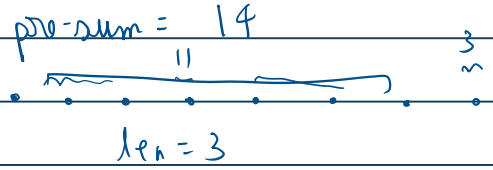
len = 5 − 2 = 3 > 2

⇒ len = 3

i = 6, pre-sum = 10, 7 is present

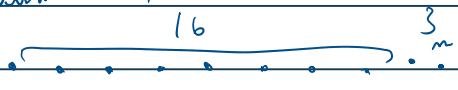
len = 6 − 3 = 3 == 3, len = 3

i = 7, pre-sum = 14



i = 8 pre-sum = 16

i = 9 pre-sum = 19



return len = 3.

```
// map<long long, int> mp;
    sum = 0, ans = 0;
    for(i=0 ⟶ n) {
        sum += a[i];
        if(sum == k)
            ans = max(ans, i+1);
        rem = sum - k;
        if(mp.find(rem) != mp.end()){
            len = i - mp.[rem];
            ans = max(ans, len);
        }
        if(mp.find(sum) == mp.end())
            mp[sum] = i;
    }
    return ans;
```

time complexity: $O(n * \log n)$

space complexity: $O(n)$

Note: This solution is the optimal solution if you have -ve, 0s, +ve in the array

// Optimal Solution: