ARRAYS - CONTD

Union of two sorted Arrays

Note:- Set

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

arr2[] = [2, 3, 4, 4, 5, 6]

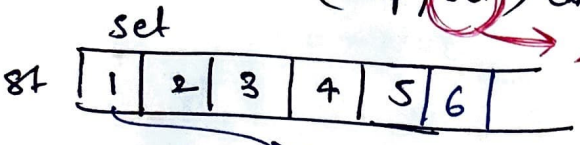
union[] = [1, 2, 3, 4, 5, 6] → no repeat in union.

Brute force approach

(map / set) can be used

set will store the elements in sorted order.

NOTE: dont use unordered set.



union[] = {1, 2, 3, 4, 5, 6}

```
set<int> st;
for(int i=0; i<n1; i++) {          → O(N1 log N)
    st.insert(arr1[i]);
}
for(int i=0 → n2) {
    st.insert(arr2[i]);
}                                  → O(N2 log N)
union[st.size()];
i=0;
for(auto it: st) {
    union[i++] = it;               → worst case O(N1+N2) if every element is unique
}
vector<int> temp;
for(auto it: st) {
    temp.push_back(it);
}
return temp;
```

N is the size of the set which would vary at every iteration

TC: O(N1 log N + N2 log N + N1 + N2)

SC: O(N1+N2)

to store the arrays in set.

+ O(N1+N2)

to return the set - to show output

Optimal Approach — 2-pointer approach

arr1 = [1, 1, 2, 3, 4, 5] (i → NULL)

arr2 = [2, 3, 4, 4, 5, 6] (j)

union [ ] = [1, 2, 3, 4, 5, 6]

If iteration in one of the array is over, you will not compare anymore but you will iterate in the other array till that finishes

Pseudocode :-

```
int i = 0;  // 1st pointer
int j = 0;  // 2nd pointer
vector<int> unionArr;

while (i < n1 && j < n2) {
    if (a[i] < b[j]) {
        if (unionArr.size() == 0 || unionArr.back != a[i]) {
            unionArr.push_back (a[i]);
        }
        i++;
    }
    else {
        if (unionArr.size() == 0 ||
            unionArr.back != b[j]) {
            unionArr.push_back (b[j]);
        }
    }
}

while (i < n1) {
    .
}

while (j < n2) {
    .
}

return unionArr;
}
```

when i = 0 & j = 0 & first element is going to be inserted in unionArr

O(n1) or O(n2)

TC : $O(n_1 + n_2)$

SC : $O(n_1 + n_2)$ ↓ to print

Missing Number

n distinct numbers [0, n]

nums = [3, 0, 1]
↳ nums.size = 3
∴ [0, 1, (2) 3] ← return. missing

Brute force TC: O(N²) SC: O(1)

```
for (int i=0; i<=n; i++) {
    bool found = false;
    for (int j=0; j<n; j++) {
        if (i == nums[j]) {
            found = true;
            break;
        }
    }
    if (found == false) {
        return i;
    }
}
return -1; // input invalid
```

N=5

hash | 0 | 0 | 0 | 0 | 0 | 0 |
0 1 2 3 4 5

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

TC: O(N) + O(N)
SC: O(N)

BETTER SOL^n

Optimal sol^n TC: O(N), SC: O(1)

sum of 1st n natural no.s [0 to n] = $\frac{n \times (n+1)}{2}$

sum of array find → sum_arr = __

sum of 1st n no.s → sum = __

missing number = sum − sum_arr = Ans

More optimal (Shivee's method) ★

(Using XOR)

$a \wedge a = 0$

$5 \wedge 5 = 0$

$2 \wedge 2 \wedge 5 \wedge 5 = 0$

$(2 \wedge 2) \wedge (5 \wedge 5) = 0$

$0 \wedge 0 = 0$

XOR of 2 same no.s = 0

XOR of any number with 0 is the number itself.

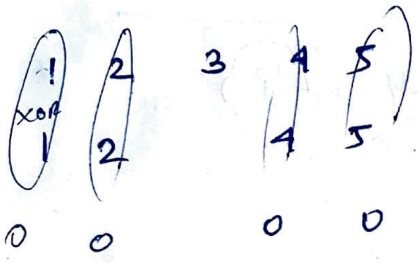
$2 \wedge 2 \wedge 2 \wedge 2 = 0$

$0 \wedge 2 = 0$

$(2 \wedge 2) \wedge (2 \wedge 2) \wedge 2$

$0 \wedge 0 \wedge 2$

$= 2$



XOR1 = $1 \wedge 2 \wedge 3 \wedge 4 \wedge 5$ → find XOR of 1 to N

XOR2 = $1 \wedge 2 \wedge 4 \wedge 5$ → Iterate through the array

XOR1 ^ XOR2

$(1 \wedge 1) \wedge (2 \wedge 2) \wedge (3) \wedge (4 \wedge 4) \wedge (5 \wedge 5)$

$0 \wedge 3 = 3$

Pseudo code :-

```
XOR1 = 0
for (i = 1 → n) {
    XOR1 = XOR1 ^ i;
}
XOR2 = 0
for (i = 0; i < n; i++) {
    XOR2 = XOR2 ^ arr[i];
}
XOR1 = XOR1 ^ (i+1)
XOR1 = XOR1 ^ N
return XOR1 ^ XOR2
```

to avoid this loop

can add it here.

Maximum Consecutive Ones

arr[] = [1, 1, 0, 1, 1, 1, 0, 1, 1]

2 ③ 2

cnt = 0 → 1 → 2 → 0 → 1 → 2 → 3 → 0 → 2 max = 0 → 1 → 2 → 3

is (1 > max) { No }
else { max = cnt }

(See code)

Find the number that appears once, and the others twice

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

VERY IMP

Method 1
Brute force sol"

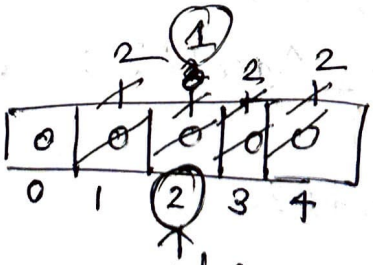
```
for (i = 0 → n) {
    num = arr[i]
    cnt = 0;
    for (j = 0 → n) {
        if (arr[j] == num) {
            cnt++;
        }
    }
    if (cnt == 1) return num;
}
```

TC: O(N²)
SC: O(1)

Method 2 Better sol"

(Hashing)

last number is 4
∴ hash size = 5



return

But the Q is → what size of hash array should be defined?

```
hash [maxelement + 1]
maxi = arr[0]
for (i=0 → n) {                    O(N)
    maxi = max(maxi, arr[i]);
}
hash [maxi] = {0}
for (i=0 → n) {                    O(N)
    hash [arr[i]] ++;
}
for (i=0 → n) {
  if (hash [arr[i]) == 1           O(N)
      cout arr[i];
}
```

TC: O(3N)

SC: O(maxi) ← depends on input

★ But can hash be used everytime?

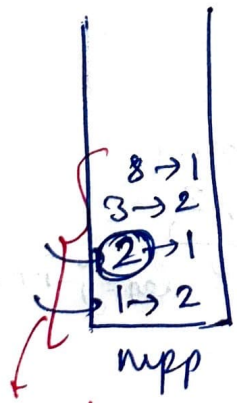
Ans: No, if values are ~~-ve~~ -ve or numbers very big like order of 10^9 or 10^{12}

then we will have to use map

map <long long, int>

key — count.

```
for (i=0 → n) {
    mpp [arr[i]] ++;
for (auto it: mpp)
  { if (it.second == 1)
      cout it.first;
}
```



stored 4 elements

(N/2)+1 elements

every element appear twice

ordered map: $N \log M$ (M = size of mpp)

unordered map: $O(N)$

$O(\frac{N}{2}+1)$

TC: $O(N \log M) + O(\frac{N}{2}+1)$

$(\frac{N}{2}+1)$

SC: $O(\frac{N}{2}+1)$ mpp

Method 3 Optimal soln XOR.

n^0 = n

n^n = 0

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

1^1 ^2 ^3 ^2 ^4 ^4

= 2

Longest sub array with sum k [positives]

sub array → contiguous part of the array

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

k = 3 ↑ sum

Find: longest subarray whose sum is k

return 3 → length of longest subarray.

Method 1: Brute force soln

generate every sub array → but how?

1
1 2
1 2 3
1 2 3 1
↓

2
2 3
2 3 1
2 3 1 1
↓

TC: $O(N^3)$

(i → j) & move j till it reaches last

SC: $O(1)$

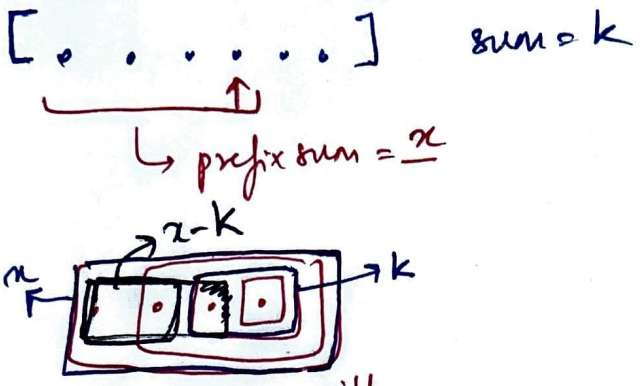
Pseudo code:

```
len = 0;
for (i → n) { sum = 0;
    for (j = i → n) {
        for (k = i → j) {
            sum = sum + arr[k];
        }
        if (sum == k) len = max(len, j-i+1);
    }
}
print(len);
```

TC: $O(N^2)$ can be written in easy way → sum += arr[j] — avoids one more loop

j=0 j=1
1, 2
i=0
len = j-i+1

Method 2: Hashing



(Refer to yt video)

⭐ IMPORTANT

This method is most optimal if array consists of ~~-ve's~~ & zeros.

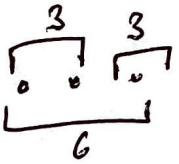
Refer method 3 → for optimal sol^n if array only consists of positives.

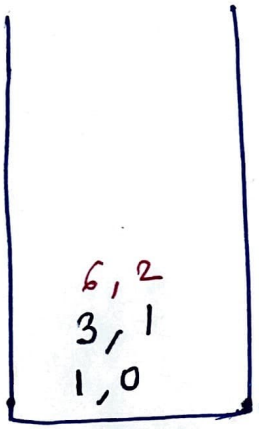
* if there exists a sub array with sum k & with (•) as the last element

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

k = 3

prefix sum = ~~0~~ 1 ~~3~~ 6 , len = ~~0~~ 2





hashmap

SC: O(N) → all prefix sum are stored.

TC: ordered map O(N log N)

unordered map O(N×1) best case or O(N×N) worst case

Pseudo code

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

Code only valid for positives

change for zeros & -ves

```
if (preSumMap.find(sum) == preSumMap.end()) {
    preSumMap[sum] = i;
}
```

Method 3 Optimal Sol^n