Arrays

Second largest element in an array

arr[] → {1, 2, 4, 7, 7, 5}

Brute force: Sort & (n-2)th index is ans ✗

as 1 2 4 5 7 7
(0 1 2 3 4 5)

↳ largest is 7
ans = 5

∴ after sorting go backwards from (n-2)th index & find element ≠ 7

```
second = -1;
for (i = n-2; i >= 0; i--)
{
    if (arr[i] != largest)   // or arr[n-1]
    {
        second = arr[i]; // might be case second largest don't exist, ∴ return -1
        break;
    }
}
```

if array has +ve, use INT_MIN ← assuming all the elements in array

$TC = \overbrace{O(n \log n)}^{\text{sort}} + \overbrace{O(n)}^{\text{loop}}$

Better

arr[]: [1, 2, 4, 7, 7, 5]

use 1st pass of loop to find → largest
2nd " " " " " → 2nd largest

$TC = O(n) + O(n) = O(2n)$

Optimal

find second largest & second smallest and return (slargest, ssmallest) in this order

PTO

```
int secondlargest(vector<int>&a, int n)
{
    int largest = a[0];
    int slargest = INT_MIN;
    for(int i=1; i<n; i++)
    {
        if(a[i] > largest)
        {
            slargest = largest;
            largest = a[i];
        }
        else if(a[i] != largest && a[i] > slargest)
        {
            slargest = a[i];
        }
    }
    return slargest;
}
int secondsmallest(vector<int> &a, int n)
{
    int smallest = a[0];
    int ssmallest = INT_MAX;
    for(int i=1; i<n; i++)
    {
        if(a[i] < smallest)
        {
            ssmallest = smallest;
            smallest = a[i];
        }
        else if(a[i] != smallest && a[i] < ssmallest)
        {
            ssmallest = a[i];
        }
    }
    return ssmallest;
}
vector<int> find(int n, vector<int> a)
{
    int slargest = secondlargest(a,n);
    int ssmallest = secondsmallest(a,n);
    return {slargest, ssmallest};
}
```

$TC = O(n) + O(n)$

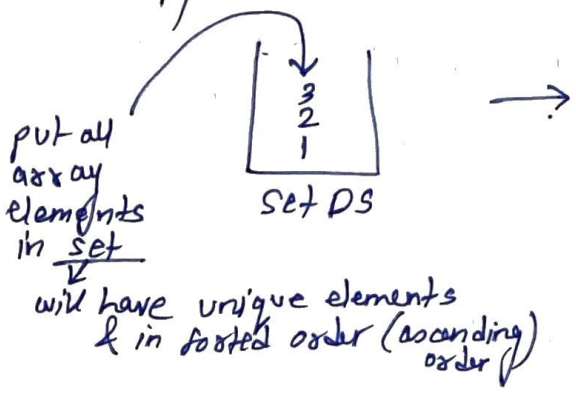
↙ for second largest ↘ for second smallest

→ Remove duplicates in-place from sorted array -

↳ remove duplicates from array (ie modify array) & also return no. of unique elements.

a[] → {1, 1, 2, 2, 2, 3, 3}

Brute force -



will have unique elements & in sorted order (ascending order)

```
set<int> st;
for(i=0; i<n; i++)
    st.insert(arr[i]);
//put it in array (unique elements)
ind = 0;
for(auto it : st)
{
    a[ind] = it;
    ind++;
}
return st.count; // ind
```

TC = O(n log n) + O(n) & SC = O(n)

- O(n log n): to insert 1 element in set → O(log n) for n elements
- O(n): to iterate in set
- O(n): for set

Optimize

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

I think it will be -

```
arr[i+1] = a[j];
```

→ left rotate the array by one -

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

↙ in left rotation

2, 3, 4, 5, 1 ← went to last

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

TC = O(n)
SC = O(1)

→ left rotate the array by D-places -

a[7] → {1, 2, 3, 4, 5, 6, 7} & n = 7

if D = 7

if D = 2

1, 2, 3, 4, 5, 6 ie 3, 4, 5, 6, 7, 1, 2 (ans)

if we rotate by same size of array, we get the same array.

∴ if D > size of array

eg D = 8 & n = 7

↙

7 + 1 ← 1 rotation

↙ get same array

∴ we use D % n

```
void leftrotate(int arr, int n, int d)
{
    d = d%n;
    reverse(arr, arr+d);
    reverse(arr+d, arr);
    reverse(arr, arr+n);
}
```

first yeh part.

Bas tho la sa modify Krna hai

(Note — left rotate ka concept right rotate mein nai lagega)

→ Move all zeroes to end of the array –

a[] → {1, 0, 2, 3, 2, 0, 0, 4, 5, 1}

o/p: {1, 2, 3, 2, 4, 5, 1, 0, 0, 0} & n = 10

Brute force –

temp[] main non-zero no. daldo ie temp → {1, 2, 3, 2, 4, 5, 1} size = 7

now, put elements of temp in arr

ie {1, 2, 3, 2, 4, 5, 1, 4, 5, 1}
(indices 0 1 2 3 4 5 6 7 8 9)

1, 2, 3, 2, 4, 5, 1 ← of temp

4, 5, 1 ← put to zeroes

i = temp.size() ie 7 till n-1

(S1)
```
temp[]
for(i=0 to n)
{ if(arr[i] != 0)
    temp.insert(arr[i]);
}
```

(S2)
```
for(i=0; i<temp.size(); i++)
{
  arr[i] = temp[i];
}
```

(S3)
```
for(i=temp.size(); i<arr.size(); i++)
{
  arr[i] = 0;
}
```

non zero nos.

TC = O(n) + O(x) + O(n-x)
(S1) (S2) (S3)

(S1) say # elements in temp

= O(2n)

SC = O(x)
temp array

Optimal (2 pointer approach)

S1) first find first '0' in array

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

(S2)
```
for(i=j+1; i<n; i++)
{ if(arr[i] != 0)
  { swap(arr[i], arr[j]);
    j++;
  }
}
```

(S1) to find 0

for (S2)

TC = O(x) + O(n-x)
= O(n)

SC = O(1)

Code

```
vector<int> nonzeroes(int n, vector<int> a)
{
  int j=-1;
  for(int i=0; i<n; i++)
  {
    if(a[i]==0)
    {
      j=i;
      break;
    }
  }
  // no zeroes
  if(j==-1)
    return a;
  for(int i=j+1; i<n; i++)
  {
    if(a[i]!=0)
    {
      swap(a[i], a[j]);
      ++j;
    }
  }
  return a;
}
```

Sab se best

```
int i=0, j=0;
while(i<n)
{
  if(a[i]!=0)
  {
    swap(a[i], a[j]);
    ++j;
  }
  ++i;
}
```

→ Missing Number in an array -

N=5 & given (n-1) nos, these (n-1) nos. will contain nos. 1 to n

eg arr[1, 2, 4, 5] & N=5
(N-1 nos.)
→ ans = 3 as 3 is not in array.

Brute force

```
for(i=1; i<=N; i++)
{
  flag=0;
  for(int j=0; j<n-1; j++) // Linear search (as array is of size n-1)
  {
    if(arr[j]==i)
    {
      flag=1;
      break;
    }
  }
  if(flag==0) // means i not there
    return i;
}
```

TC = O(N×N)
SC = O(1)

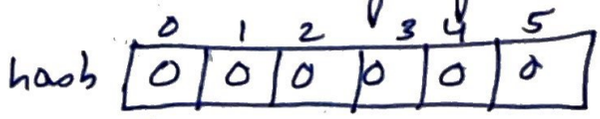
Better

$N = 5$

we know nos. are b/w 1 to 5

∴ we need 0th to 5th index if we want them in hash.

as we need 5th index, declare array of size = 6



set all to 0 initially

∴ hash[n+1] = {0}; → as n-1 elements in array

```
for (i = 0; i < n-1; i++)
{
    hash[arr[i]] = 1;
}
for (i = 1 → n)        → not 0 as in hash[], we want to check for 1 to N as has[0] will always have 0
{
    if (has[i] == 0)
        return i;
}
```

TC = O(n) + O(n)

SC = O(n)
↳ hash array

optimal

$n = 5$

M1- $Sum = \frac{n(n+1)}{2}$

$= \frac{5 \times 6}{2} = 15$

$S = 1 + 2 + 4 + 5$

$= 12$

return (Sum - S);

→ TC = O(N)
SC = O(1)

M2- Using XOR

↳ Note- $a \wedge a = 0$

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

$0 \wedge 0 = 0$

eg) $\underbrace{2 \wedge 2}_{0} \wedge \underbrace{2 \wedge 2 \wedge 2}_{0}$

∴ $0 \wedge 2 = 2$

N: 1 2 3 4 5

arr: 1 2 4 5

$\therefore XOR1 = 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5$

for array: $XOR2 = 1 \wedge 2 \wedge 4 \wedge 5$

now, $XOR1 \wedge XOR2$

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

$= 3$

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

takes 2 loops

$TC = O(n) + O(n)$

$SC = O(1)$

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

XOR1 will have 1 to N-1

$TC = O(n)$

$SC = O(1)$

Note: if $n = 10^5$

$\therefore$ in M1, $sum = \frac{10^5 \times (10^5+1)}{2} \approx 10^{10}$

↳ can't be stored in integers (it will overflow)

So we need bigger data type ie long.

So M2 is good

↳ XOR of all nos. will not be that big.

→ Maximum Consecutive ones -

given an array of 0 & 1.

arr[] → {1, 1, 0, 1, 1, 1, 0, 1, 1}
(1,1 → 2; 1,1,1 → 3; 1,1 → 2)

↳ max consecutive 1 is 3

Code

```
int maxi=0;
int count=0;
for(int i=0; i< nums.size(); i++)
{
    if(nums[i]==1)
    {
        ++count;
        maxi= max(count, maxi);
    }
    else
    {
        count=0;
    }
}
return maxi;
```

other way

```
if(nums[i]==1)
{
    ++count;
}
else
{
    maxi=max(count, maxi);
    count=0;
}
}
maxi=max(count, maxi);
return maxi;
```

0, 1, 0, 1, 1, 0, 0, 1, 1, 1 → for this case

→ Longest subarray with given sum K [positives]

arr[] → {1, 2, 3, 1, 1, 1, 1, 4, 2, 3} & K = 3

Note – subarray is contiguous part of array

eg {1, 1, 1} ← (subarray)

{1, 2} ← (not subarray as not contiguous)

here for K=3

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

- 1 → =K & len=2
- 1,1,1 → =K & length=3
- 1,1,1 → =K & length=3
- 3 → =K & length=1

Ans = 3

Brute force

- generate all subarrays

ie

```
[ 1
  1,2
  1,2,3
  ⋮
  1,2,3,1,1,1,1,4,2,3

  2
  2,3
  ⋮
  2,3,1,1,1,4,2,3
  ⋮
```

```
len=0
for(i=0; i<n; i++) //to generate all subarray
  for(j=i; j<n; j++)//
    s=0;
    for(k=i; k<=j; k++)
      s=s+a[k];
      if(s==k) //and this k
        len=max(len, j-i+1);
  }
return len;
```

TC ≃ O(n³)
SC = O(1)

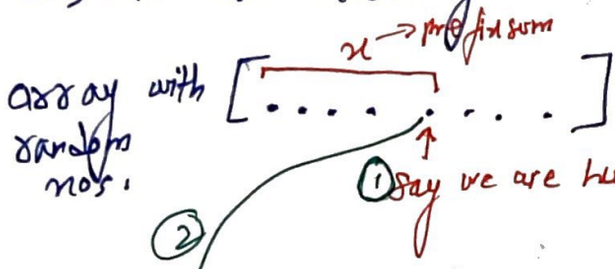
→ good brute force

```
len=0;
for(i=0; i<n; i++)
  s=0;
  for(j=i; j<n; j++)
    s=s+a[j];
    if(s==k)
      len=max(len, j-i+1);
  }
return len;
```

→ TC = O(n²)

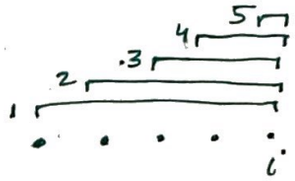
Better

↳ will use hashing



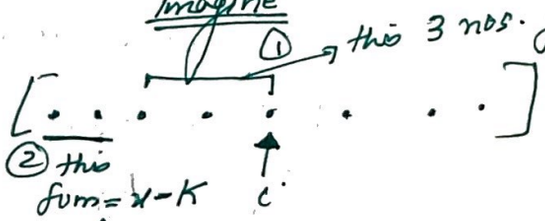
prefix sum = from front till i
= (say) x

there are 5 subarrays w/ '.' as last



if there exists a subarray with sum = K
as (.) as the last element

imagine



↳ then we can say, ① is 'sum of=K

Dry run

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

prefix sum = ~~0~~ maxLen = ~~0~~ ~~2~~ 3

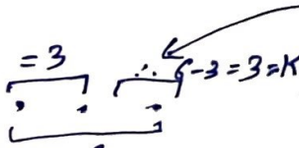
~~3~~ == K . . . . .

6

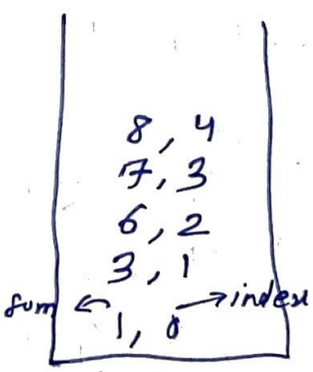
~~7~~

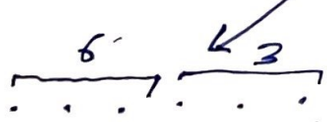
~~8~~

9



∴ we get subarray of length = 1 (↪2)





sum = 9
we get 6 at index = 2 ↳ at ind = 5 we get 9, ∴ len = 5 − 2 = 3

```
int longestSubarray(vector<int> a, long long K)
{
    map<long long, int> map;
    long long sum=0;
    int maxlen=0;
    for(int i=0; i<a.size(); i++)
    {
        sum += a[i];
        if(sum == K) // initially always check if array from start till current index gives you K
        {
            maxlen = max(maxlen, i+1);
        }                                  // as from ind=0 (start) checking
        long long rem = sum-K;
        if(map.find(rem) != map.end()) // that is rem exists in map
        {
            int len = i - map[rem];
            maxlen = max(maxlen, len);
        }
        map[sum] = i;
    }
    return maxlen;
}
```

put in map if sum previously not there

for ②

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

→ this code works –

① only for +ve nos. & -ve nos.

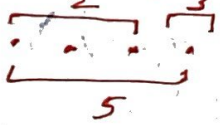
② only for +ve nos. & zeros; this won't work

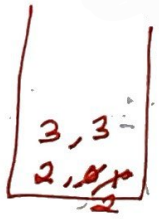
Why?

arr[] → {2, 0, 0, 3} (0, 1, 2, 3) & K = 3

prefixsum = 0 → 2 → 5

maxlen = 0 → 1





but we need longest

ans should be len = 3 i.e. 0, 0, 3

∴ if a sum previously exist here, we should not reupdate its index

changes : see above

③ This code is optimal for array having +ve, -ve nos.

$TC = O(n * \log n)$ (or) $O(n * 1)$

- n → to iterate in array
- $\log n$ → ordered map
- 1 → unordered map, WC = 1 will be N rare

$SC = O(n)$ ↳ in wc every prefix sum is unique

Optimal

↳ 2 pointers approach

arr[] → [1, 2, 3, 1, 1, 1, 1, 3, 3] & K = 6

sum = 0 → 1 → 3 → 6 ==K → 7 (>K)

Len = 0 → 3 → 4

∴ 7 - 1 ↳ at 0 = 6 ==K

7 (>K) ↳ ∴ shrink & remove 2

∴ 7 - 2 = 5 → 6 (==K) ↳ len = 4 → 7 (>K) → 4 → 7 (>K) → 6 (==K) ↳ len = 4 → 9 → 8 → 7 → 6 (==K) ↳ len = 2

Aditya verma works for only +ves

sliding window

see

Works for tres & 0's [not for -ves]

```
int longSubarray(vector<int> a, long long K){
    int left=0, right=0;
    long long sum = a[0];
    int maxlen = 0;
    int n = a.size();
    while(right<n){
        while(left<=right && sum>K){
            sum = sum - a[left];
            ++left;
        }
        if(sum==K){
            maxlen = max(maxlen, right-left+1);
        }
        ++right;
        if(right<n)
            sum += a[right];
    }
    return maxlen;
}
```

there is a subarray

$TC = O(2N)$

outer while runs till N & inner while(). overall for outer while runs for O(N)

$SC = O(1)$

→ Two sum problem -

arr[] → {2, 6, 5, 8, 11} & target = 14

Your task is tell if there exists 2 elements in array such that a + b = target

eg target = 14
∴ 6 + 8 = 14 ✓

eg target = 15
can't find a way in which 15 is possible

Brute

```
for(i=0; i<n; i++)
    for(j=i+1; j<n; j++)
        if(arr[i]+arr[j]==target)
            return true; // if asked possible or not
            (or)
            return {i, j}; // if index is asked
}return false
```

→ TC = $O(N^2)$

Better

→ use hashing

arr[] → {2, 6, 5, 8, 11} & target = 14

- 2 → 14-2 = 12 → not in map ∴ put 2 in map
- 6 → 14-6 = 8 → put 6 in map
- 5 → 14-5 = 9
- 8 → 14-8 = 6 → present in map & 6 is present at index 1

∴ return true
or return (8, 5) → index

hashmap (element, ind)

element	ind
5	2
6	1
2	0

code → return "YES" or "NO"

```
string 2sum (int n, vector<int> book, int target)
{
  map<int,int> m;
  for(int i=0; i<n; i++)
    {
      int more = target-book[i];
      if(m.find(more) != m.end()){
        return "YES"; // return {m[more], i};
      }
      m[book[i]] = i;
    }
  return "NO";
}
```

if asked for index

TC = O(N × log N) // for ordered map
= O(N × 1 (or N)) → wc // for unordered map

SC = O(N)

optimal → 2 pointers approach

arr[] → {2, 6, 5, 8, 11} & target = 14

sort → arr: {2, 5, 6, 8, 11}

right ① ③ ② left

- 2+11 = 13 (<target) ∴ ↑right
- 5+11 = 16 (>target) ∴ ↓left
- 5+8 = 13 (<target) ∴ ↑right
- 6+8 = 14 (==target) ← return Yes

& if at any moment r cross l, return NO.

& if asked for return indexes, optimal

```
string 2sum (int n, vector<int> book, int target)
{
    int left=0, right=n-1;
    sort(book.begin(), book.end());
    while(left<right)
    {
        int sum = book[left] + book[right];
        if(sum==target)
            return "YES";
        else if(sum<target)
            ++left;
        else
            --right;
    }
    return "NO";
}
```

sort ↑ while() ↑

$TC = O(N\log N) + O(N)$

$SC = O(1)$

→ **Majority element in an array** = (> N/2 times) (appears)

arr[] → [2 2 3 3 1 2 2] (indices 0 1 2 3 4 5 6) & n = 7

'2' appears for 4 times (>7/2) (>3)

Brute force

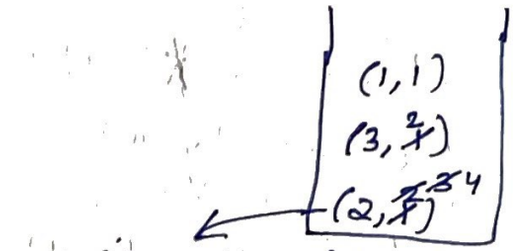
```
for(i=0; i<n; i++)
{
    cnt=0;
    for(j=0; j<n; j++)
    {
        if(arr[j]==arr[i])
            ++cnt;
    }
    if(cnt > n/2)
        return arr[i];
}
```

$TC = O(n^2)$

$SC = O(1)$

Better – use hashing

arr[] → [2 2 3 3 1 2 2]



2 has frequency > n/2
∴ Ans = 2

```
int majority_element(vector<int> v)
{
    map<int, int> m;
    for(int i=0; i<n; i++)
    {
        m[v[i]]++;
    }
    for(auto it : m)
    {
        if(it.second > (v.size())/2)
            return it.first;
    }
    return -1; //if no majority element
}
```

iterate map → O(n)

→ TC = O(n * log n) + O(n)
(n: loop, log n: insert in map)

SC = O(n) if all are unique elements

Optimal : Moore's Voting Algorithm

arr[] : {7, 7, 5, 7, 5, 1, 5, 7, 5, 5, 7, 7, 5, 5, 5, 5}

tells who is the majority element

ele = 7 7 5 5

cnt = 0 1 2 1 0 → as we get 5, reduce cnt by 1 as it is not 7

→ as 5

0 ← we get 0 → since we get 0 → we have reached

↳ now make ele = 5, as count = 0

7 7 5 7 5 1 & cnt = 0

ele = 7 & others (5, 1)

3 times ++ | 3 times --

∴ for this subarray, 7 is not the majority element, if 7 is the majority element, it should be > (size of subarray)/2

& cnt will not be zero

now, 0 1 0 1 2 1 0 1 2 3 4

since, we ended the iteration & ele = 5

now, we have
ele = 5
↙
to verify it is majority element or not,
iterate through array & see if it occurs > n/2 times,
it is ans else return -1

this

Steps: ① Apply moore's voting algo
② verify it is majority or not (iterate array again)

```
int majorityelement(vector<int> a)
{
    int cnt = 0;
    int el;
    for(int i=0; i<v.size(); i++)
    {
        if(cnt == 0)
        {
            cnt = 1;
            el = v[i];
        }
        else if (v[i] == el)
            ++cnt;
        else
            --cnt;
    }
    int cnt1 = 0;
    for (int i = 0; i<v.size(); i++)
    {
        if (v[i] == el)
            ++cnt1;
    }
    if (cnt1 > (v.size()/2))
        return el;
    return -1;
}
```

→ value of cnt at end does not represent anything.

// this loop not needed if problem states there always exists majority element, then return el.

$$TC = O(n) + O(n)$$
$$SC = O(1)$$

→ Majority element ($> n/3$ times) -

- for majority element ($> n/2$) → only 1 ans possible (max)
  - a $\frac{n}{2}$ is half of array → and majority element occurs $\geq \frac{n}{2}+1$ times
- for majority element ($> n/3$) → 2 ans possible (max)
  - can be 0 or 1 or 2

eg [1, 1, 1, 3, 3, 2, 2, 2] & n = 8

∴ to be majority element here, element should occur $\geq \frac{n}{3}+1$

say, 3 majority element occuring $\frac{n}{3}+1$ times

$\left(\frac{n}{3}+1\right) + \left(\frac{n}{3}+1\right) + \left(\frac{n}{3}+1\right)$

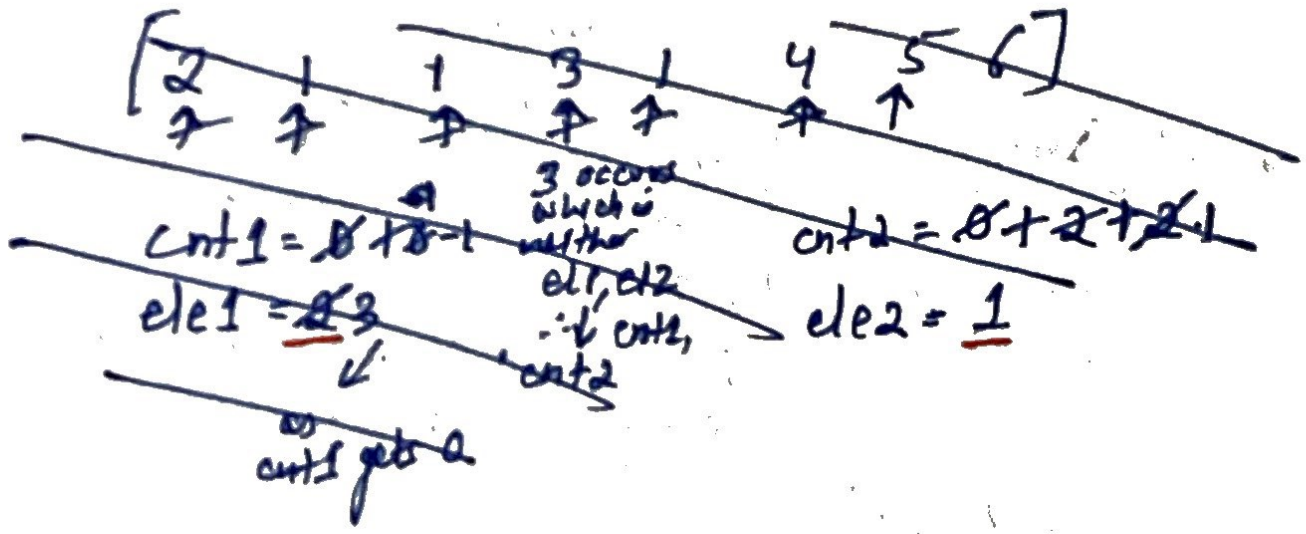
$= n+3 > 3$

∴ not possible

Only 2 at max

in given array, 1 & 2 are majority element

We will use extend version of Moore's voting algo. Since, atmost 2 majority element possible, we have cnt1=0, cnt2=0



$A = [2, 2, 1, 3, 1, 1, 3, 1, 1]$

ele1 = 2
cnt1 = 0 ✗ 2 ✗ 0

ele2 = ✗ 1
cnt2 = 0 ✗ 0 ✗ 1 ✗ 2 ✗ 1 ✗ 3

now, we get ele1 = 2, ele2 = 1
iterate through array & find their count
if count > $\frac{n}{3}+1$ → majority element

O/P = 1

```
vector<int> MajEle(vector<int> v)
{
    int cnt1=0, cnt2=0;
    int el1=INT_MIN, el2=INT_MIN;
    for(int i=0; i<v.size(); i++)
    {
        if(cnt1==0 && v[i]!=el2)
        {
            cnt1=1;
            el1=v[i];
        }
        else if(cnt2==0 && v[i]!=el1)
        {
            cnt2=1;
            el2=v[i];
        }
        else if(v[i]==el1)
            ++cnt1;
        else if(v[i]==el2)
            ++cnt2;
        else
        {
            --cnt1;
            --cnt2;
        }
    }
    vector<int> ans;
    cnt1=0, cnt2=0;
    for(int i=0; i<v.size(); i++)
    {
        if(el1==v[i])
            ++cnt1;
        if(el2==v[i])
            ++cnt2;
    }
    int mini = (int)(v.size()/3)+1;
    if(cnt1>=mini)
        ans.push_back(el1);
    if(cnt2>=mini)
        ans.push_back(el2);
    sort(ans.begin(), ans.end());
    return ans;
}
```

TC = O(n)+O(n)

SC = O(1)

Kadane's Algorithm [Maximum subarray sum]

contiguous part of array

arr[] → {-2, -3, 4, -1, -2, 1, 5, -3}

ans = 7

Brute force

Try out all subarrays & find the subarray with max sum.

```
maxi = -infinity;
for(i=0; i<n; i++)
{
    for(j=i; j<n; j++)
    {
        sum = 0;
        for(k=i to j)
        {
            sum += a[k];
        }
        maxi = max(sum, maxi);
    }
}
```

→ TC = O(n³)
SC = O(1)

→ every subarray generation

Better

```
for(i=0; i<n; i++)
{
    sum = 0;
    for(j=i; j<n; j++)
    {
        sum += arr[j];
        maxi = max(sum, maxi);
    }
}
```

→ TC = O(n²)
SC = O(1)

Optimal - Kadane's Algorithm

arr[] → {-2, -3, 4, -1, -2, 1, 5, -3}
indices: 0, 1, 2, 3, 4, 5, 6, 7

maxi = INT_MIN ~~-2~~ ~~47~~

sum = ~~0~~ -2 → make it to 0 ~~0~~ → -3 → ~~0~~ → 4 → 3 → 1 → 2 → 7 → 4

Note: if at any moment sum < 0, no need to take as it will ~~my~~ sum in future

Ans = maxi = 7 & subarray is [4, -1, -2, 1, 5]

```
long long maxSubSum(int arr[], int n)
{
    long long sum=0, maxi = LONG_MIN;
    for(int i=0; i<n; i++)
    {
        sum += arr[i];
        if(sum > maxi)
            maxi = sum;
        if(sum < 0)
            sum = 0;
    }
    // if(maxi < 0)
    //     maxi = 0;
    return maxi;
}
```

in some cases the ~~question might~~ consider the sum of empty subarray while solving this problem.
So, in these cases before returning the answer we will compare the maximum subarray sum calculated with 0 (the sum of an empty subarray is 0) & after that return the maximum one.
eg {-1, -4, -5}
ans is 0 instead of -1 for this question.

TC = O(n)
SC = O(1)

To print the subarray of maximum sum-

```
sum=0, maxi=LONG_MIN, ansStart=-1, ansEnd=-1;
for(i=0; i<n; i++)
{
    if(sum==0) // whenever sum=0 we are having a new start
        start=i;
    sum = sum + a[i];
    if(sum > maxi)
    {
        maxi = sum;
        ansStart = start;
        ansEnd = i;
    }
    if(sum < 0)
        sum = 0;
}
```

→ TC = O(n)
SC = O(n) for this

now put subarray in vector & return it.

Rearrange array elements by sign -

Given an array w/ equal +ve & -ve elements
ie if length of array is N then → N/2 +ve elements
→ N/2 -ve elements
∴ N is always even

arr[] → {3, 1, -2, -5, 2, -4}

↙ rearrange as +, -, +, -, +, - like this

ie
0 1 2 3 4 5
{3, -2, 1, -5, 2, -4}
+ - + - + -

here, +ves : 3, 1, 2 here if you see relative position of +ve & -ves are maintain
-ves : -2, -5, -4

Brute -

taking empty array pos[] & neg[] of each size N/2

pos[] → {3, 1, 2}
neg[] → {-2, -5, -4}

in terms of indexes → +ve nos. are at even index } in
→ -ve nos. " " odd index } our

pos
0 2 4 → index of final ans
3, 1, 2
0 1 2 → index of pos

neg
1 3 5 → index of final ans
-2, -5, -4
0 1 2 → index in neg

```
for (i=0; i<n/2; i++) {
    arr[2*i] = pos[i];
    arr[2*i+1] = neg[i];
}
```

→ TC = O(n) + O(n/2)
↙ to get +ve & -ve nos in pos & neg

SC = O(n/2) + O(n/2)
= O(n)
↙ for pos & neg array

optimal -

arr[] → {3, 1, -2, -5, 2, -4}
(indices: 0, 1, 2, 3, 4, 5)

all +ve elements are at even indices
-ve " " " " odd " } in our final ans

```
vector<int> rearrangeArray(vector<int> &nums)
{
    int n = nums.size();
    vector<int> ans(n, 0);
    int posind = 0, negind = 1;
    for(int i = 0; i < n; i++)
    {
        if(nums[i] < 0)
        {
            ans[negind] = nums[i];
            negind = negind + 2;
        }
        else
        {
            ans[posind] = nums[i];
            posind = posind + 2;
        }
    }
    return ans;
}
```

- Same Question with condition

if any of the +ve & -ve nos. are left (i.e. size of pos & neg array won't be necessary they should always be n/2), add them at the end w/o altering the order.

say, arr[] → {1, 2, -4, -5, 3, 6}

when no. of pos ≠ no. of neg

#pos > #neg

here, 2 negs & 4 pos

∴ #pos > #neg

#neg > #pos

∴ Ans is → {1, -4, 2, -5, 3, 6} ✓

for this we can't use optimal solution
as it was lying on the fact
#pos = #neg &
- we are filling them alternative wise in final array

we fall back to brute force method

∴ for array[] → {-1, 2, 3, 4, -3, 1}
pos[] → {2, 3, 4, 1}
neg[] → {-1, -3}
here, #pos > #neg
∴ first 4 elements will be in alternative order
ie #neg = 2
#pos = 2
& remaining = 2 pos nos will be put at end.

```
vector<int> alternative(vector<int> &a)
{
    int n = a.size();
    vector<int> pos, neg;
    for(int i=0; i<n; i++)
    {
        if(a[i]>0)
            pos.push_back(a[i]);
        else
            neg.push_back(a[i]);
    }
    if(pos.size()>neg.size())
    {
        for(int i=0; i<neg.size(); i++)
        {
            a[2*i] = pos[i];
            a[2*i+1] = neg[i];
        }
        int index = 2*neg.size();
        for(int i=neg.size(); i<pos.size(); i++)
        {
            a[index] = pos[i];
            ++index;
        }
    }
    else // for pos.size() <= neg.size()
    {
```

} for #pos == #neg

PTO

```
for (int i=0; i<pos.size(); i++)
{
    a[2*i] = pos[i];
    a[2*i+1] = neg[i];
}
int index = 2*pos.size();
for (int i = pos.size(); i<neg.size(); i++)
{
    a[index] = neg[i];
    ++index;
}
return a;
```

if #pos == #neg the loop executes & this will not be executed

$TC = O(N) + O(\min(pos, neg)) + O(\text{leftovers})$

min → occurs when all are +ve or -ve
$\therefore O(0) + O(N)$
leftovers

max → when pos = neg = N/2
$O(N/2) + O(0)$
leftovers = 0

∴ in both cases, it is $O(N)$

$TC = O(N) + O(N)$
$SC = O(N)$ ↳ for pos & neg array

→ Next Permutation -

↳ given an array of integers (can has repitition of nos.), find the next permutation of array.

given arr[] → {3, 1, 2}

all permutations

total = 3! = 6 ways

1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

all are in sorted order (for string lexographic order)

here, arr[] is {3 1 2}
∴ next permutation is {3 2 1} (ans)

& if arr[] is {3, 2, 1} → no one after {3,2,1}
∴ go to ~~first~~ permutation ie {1, 2, 3} (Ans)

Say, arr[] → {2, 1, 5, 4, 3, 0, 0} & n = 7

- we are looking for longer prefix match

2 1 5 4 3 0 0
if we match everything, we get same array

2 1 5 4 3 0 0
our prefix match — try to match this one — only 1 no. left to rearrange - we get og array only

2 1 5 4 3 0 0
try to match this — only 1 way & we get og array (array still remains same)

2 1 5 4 3 0 0
try to match — 3 nos to rearrange
030 < og array
003 < " "
whatever we did we get <= og array as we want some one > 3

2 1 5 4 3 0 0
try to match — we try any combo, we get <= og array as we want some one > 4

2 1 5 4 3 0 0
try to match this — try any combo, we get <= og array

2 1 5 4 3 0 0
try to match ie 1 prefix match — possible as we have no > 1 ie 3, 4, 5
→ PTO

∴ for 2 1 5 4 3 0 0

- 2 5 4 3 1 0 0 (but not next permutation)
- 2 4 5 3 1 0 0 (but not next permutation)

but we can't have 2 0 0 4 5 3 1 as log array

there can be lot of arrangements but what will be next permutation?

for this, go from backward & find no just > 1 i.e 3 & swap 1 & 3

∴ 2 ① 5 4 3 0 0

∴

2 ③ 5 4 1 0 0 (but this is not next permutation)

if you see -

it is in descending order

→ if you sort it → we get 0 0 1 4 5

∴ 2 3 0 0 1 4 5 is next permutation

→ but just reverse this

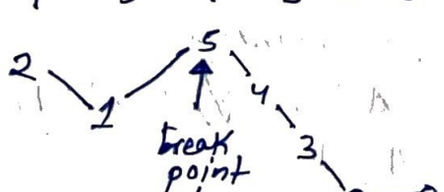
i.e 5 4 1 0 0 —rev→ 0 0 1 4 5

∴ 2 3 0 0 1 4 5 is next permutation

Algo

if you see -

2 1 5 4 3 0 0



∴ 2 1 | 5 4 3 0 0

i i+1

1) find break point i.e a[i] < a[i+1]

2) from i+1 to end, find no. just larger than a[i] (here we get 3) & swap both.

3) from i+1 till end reverse it (as it is in descending order, so we get ascending order)

for next permutation

Note - 1 2 3 4 5 → from this we get to know, the last index we can have dip is (n-2)th index

5 4 3 2 1 ← no dip (∵ for this case no dip) reverse of 5 4 3 2 1 ∴ return 1 2 3 4 5 for it, (next permutation)

Code

```
vector<int> nextPermutation(vector<int> A)
{
    int ind = -1;
    int n = A.size();
    for(int i=n-2; i>=0; i--) // to find dip point
    {
        if(A[i] < A[i+1]) // dip point
        {
            ind = i;
            break;
        }
    }
    if(ind == -1) // no dip point
    {
        reverse(A.begin(), A.end());
        return A;
    }
    for(int i=n-1; i>ind; i--) // to find no just greater than A[i]
    {
        if(A[i] > A[ind])
        {
            swap(A[i], A[ind]);
            break;
        }
    }
    //reverse remaining
    reverse(A.begin()+ind+1, A.end());
    return A;
}
```

[zero talk jaise koi jaroorat nai]

$TC = O(n) + O(n) + O(n) = O(3n)$

$SC = O(1)$

Leaders in an Array =

→ everything on the right should be smaller

arr[] → {10, 22, 12, 3, 0, 6} → note: last element is always the leader

Leaders = {22, 12, 6}
→ as right of them everything < them

return in this format
ie relative position same

Brute force

```
for (i=0 ; i<n ; i++)
{
    leader = true;
    for (j = i+1 ; j<n ; j++)
    {
        if (a[j] > a[i])
        {
            leader = false;
            break;
        }
    }
    if (leader == true)
        ans.push_back(arr[i]);
}
```

→ TC = O(n²)
SC = O(n)
for ans vector
WC
arr 5 4 3 2 1
all are leaders

Optimal

arr[] → {10, 22, 12, 3, 0, 6}
start from end

max = ~~INT_MIN~~, 6, 12, 22

Leaders [6 12 22] → reverse: 22, 12, 6 (ans)
to maintain relative position

6 > INT_MIN

[if asked to return in sorted order, sort this]

```
vector<int> leaders (vector<int>& a)
{
    vector<int> ans;
    int maxi = INT_MIN;
    int n = a.size();
    for (int i = n-1; i >= 0; i--)
    {
        if (a[i] > maxi) {
            ans.push_back(a[i]);
            maxi = a[i];
        }
    }
    sort(ans.begin(), ans.end());
    return ans;
}
```

TC = O(n)
SC = O(n)
↳ for ans array
↳ WC when everyone is leader

→ Longest consecutive sequence –

arr[] → {102, 4, 100, 1, 101, 3, 2, 1, 1}

↳ given an array of integers, ↳ longest consecutive seq is {1, 2, 3, 4}
& length of seq = 4

if we pick {100, 102, 102}
↳ Len = 3

Brute force

```
longest = 1;
for (i=0; i<n; i++)
{
    x = a[i];
    cnt = 1; // initially 1 element
    while (Linearsearch(arr, x+1) == true)
    {
        x = x+1;
        ++cnt;
    }
}
```

→ as atleast len of 1 longest consecutive seq is possible

Ans
(return the length)

```
Linearsearch(arr, num)
{
    // write khud se
}
```

TC = O(n) × O(n)
= O(n²)

SC = O(1)

Better

arr[] → {100, 102, 100, 101, 101, 4, 3, 2, 3, 2, 1, 1, 1, 2}

↓ sort

{1, 1, 1, 2, 2, 2, 3, 3, 4, 100, 100, 101, 101, 102}

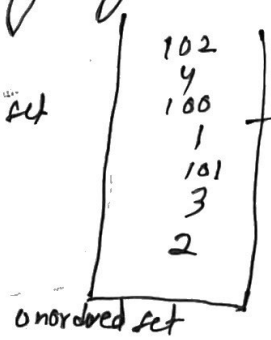
```
int longest(vector<int> &nums)
{
  sort(nums.begin(), nums.end());
    int n = nums.size();
    int lastSmaller = INT_MIN;
    int cnt = 0;
    int longest = 1;
    for(int i=0; i<n; i++)
    {
      if(nums[i]-1 == lastSmaller)
      {
        cnt += 1;
        lastSmaller = nums[i];
      }
      else if(lastSmaller != nums[i])
      {
        cnt = 1;
        lastSmaller = nums[i];
      }
      longest = max(longest, cnt);
    }
    return longest;
}
```

TC = O(nlogn) + O(n)

Optimal

arr[] → {102, 4, 100, 1, 101, 3, 2, 1, 1}

↳ put everything in set (unordered set)



→ will have unique elements & not sorted as unordered

- started from 102 (say)
  check (102-1) = 101 there in set
  Yes, ∴ ignore
- then 101
  check 101-1 = 100 → it is in set
  ∴ ignore
- 100
  check → 100-1 = 99 in set
  No.

∴ 100 is the starting point, now iterate from 100 we get 100, 101, 102, ∴ len = 3

- similarly we get 1 as starting point we get 1, 2, 3, 4 & len = 4 → our ans

```
int longest(vector<int> &a)
{
    int n = a.size();
    if (n == 0)
        return 0;
    int longest = 1;
    unordered_set<int> st;
    for (int i = 0; i < n; i++)
    {
        st.insert(a[i]);
    }
    for (auto it : st)
    {
        if (st.find(it - 1) == st.end())   // it-1 not in set
        {                                  // ∴ it is the starting point
            int cnt = 1;
            int x = it;
            while (st.find(x + 1) != st.end())
            {
                x = x + 1;
                cnt = cnt + 1;
            }
            longest = max(longest, cnt);
        }
    }
    return longest;
}
```

$TC = O(N) + O(N + N) = O(3N)$ [Under assumptions set takes O(1), unordered]

if TC is mentioned, O(N) is better solution

$SC = O(N)$

to iterate in set

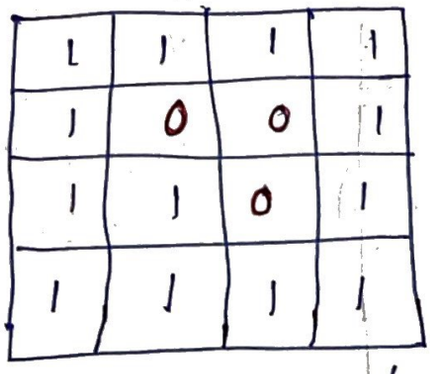
for

102
4
100
1
101
3
2

7 iterations + 3 iterations (100 → 101 → 102) + 4 iterations (1 → 2 → 3 → 4) = 14 ≈ 2N

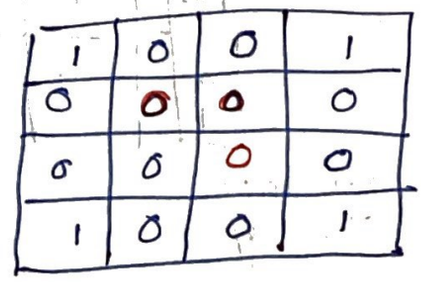
as we always looked for starting point & not anything else

Set matrix Zeroes -



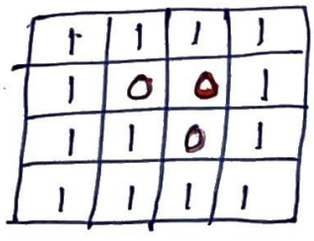
given n×m matrix having 0 & 1, find where is 0 & mark that row & col'm as 0

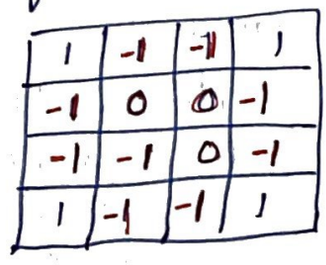
o/p



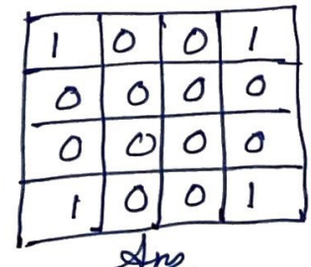
Brute force

Whenever you encounter '0', in that row & col'm whoever is 1, mark them -1. After this is done, replace -1 with 0.





replace -1 with 0



Ans

```
① for(i=0;i<n;i++)
  {
    for(j=0;j<m;j++)
    {
      if(arr[i][j]==0)
      {
        markRow(i);
        markCol(j);
      }
    }
  }
```

```
② markRow(i)
  {
    for(j=0;j<m;j++)
    {
      if(arr[i][j]!=0)
        arr[i][j]=-1;
    }
  }
```

```
③ markCol(j)
  {
    for(i=0;i<n;i++)
    {
      if(arr[i][j]!=0)
        arr[i][j]=-1;
    }
  }
```

TC = O(n*m) * O(n+m)
= O(n×m)
≈ O(n³) if n=m

④ now, replace every -1 w/ 0.
ie

```
for(i=0→n)
{
  for(j=0→m)
  {
    if(arr[i][j]==-1)
      arr[i][j]=0;
  }
}
```

Better

→ create m-size col'm array of m-size

0	0	0	0

0	1	1	1	1
0	1	0	1	1
0	1	1	0	1
0	1	0	0	1

Initially mark every one of them as 0 → means yet not touched

row array of n-size

[here, if we encounter 0, mark that col'm & row as 1 → so that row&col can be replace with 0]

```
col[m] = {0};
row[n] = {0};
for (i=0 → n)
{
  for (j=0 → m)
  {
    if (a[i][j] == 0)
    {
      row[i] = 1;
      col[j] = 1;
    }
  }
}
for (i=0 → n)
{
  for (j=0 → m)
  {
    if (row[i] || col[j])
      a[i][j] = 0;
  }
}
```

→ TC = O(n×m) + O(n×m)

SC = O(n) + O(m)
(n → row[], m → col[])

Optimal (In-place optimization)

optimize it

instead of using extra space → use given matrix

i.e. col[0] → row[n]
row[0] → col[m]

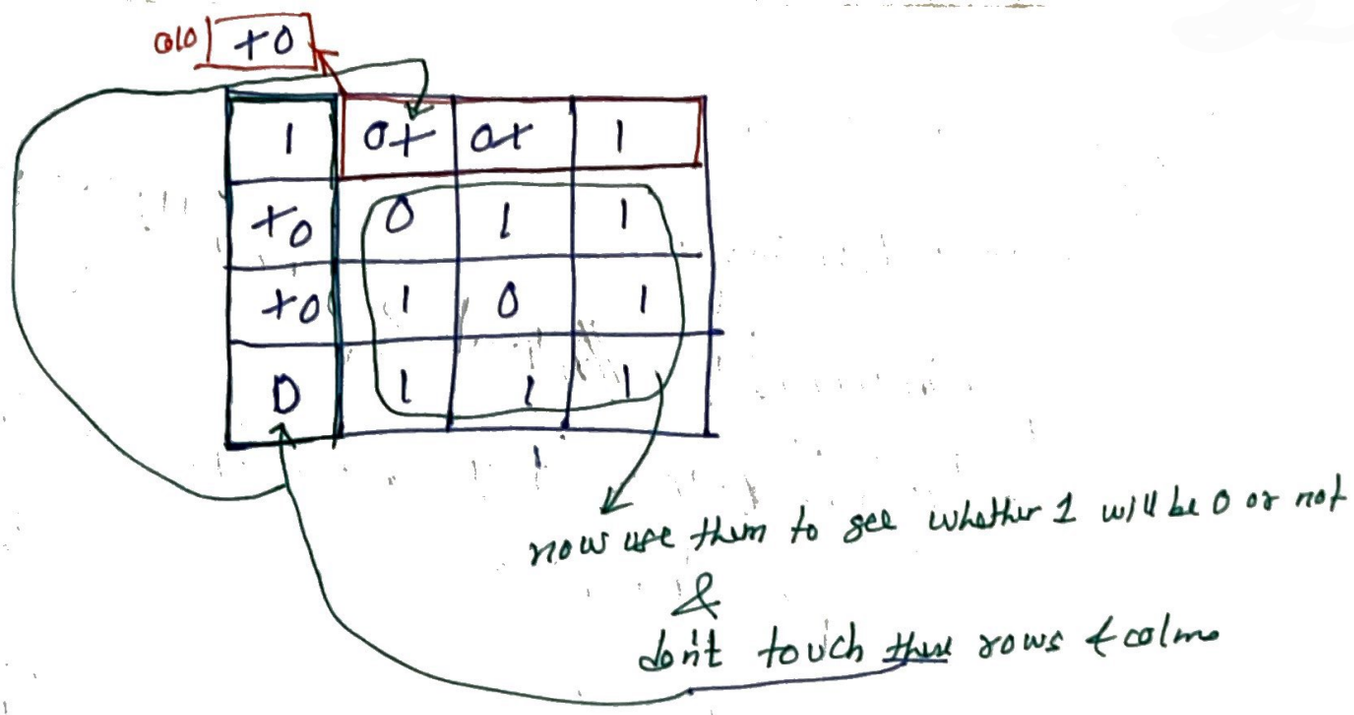
Common point

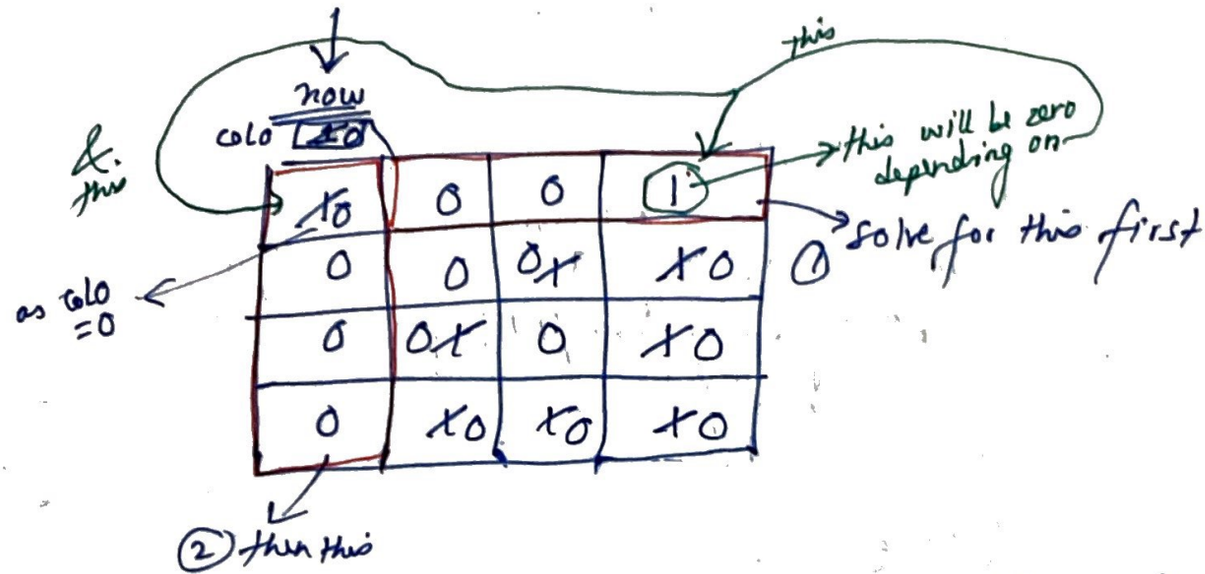
1	1	1	1
1	0	1	1
1	1	0	1
1	0	0	1

∴ to not face problem w/ common point

[1] → col[0] [extra part] =1

1	1	1	1
1	0	1	1
1	1	0	1
1	0	0	1





Code here, `int col[m] = {0}` ⟶ matrix[0][...] (first row)
`int row[n] = {0}` ⟶ matrix[...][0] (first colm)

```
vector<vector<int>> zeromatrix (vector<vector<int>> &matrix, int n, int m)
{
  int col0 = 1;
  for(int i=0; i<n; i++)
  {
    for(int j=0; j<m; j++)
    {
      if(matrix[i][j] == 0)
      {
        //mark ith row
        matrix[i][0] = 0;
        //mark jth col
        if(j != 0)
          matrix[0][j] = 0;
        else col0 = 0;
      }
    }
  }
```

```
for (int i=1; i<n; i++)
{
  for (int j=1; j<m; j++)
  {
    if(matrix[i][j] != 0) //optional → agar nai bhi logo
    {
      if(matrix[0][j]==0 || matrix[i][0]==0)
      {
        matrix[i][j]=0;
      }
    }
  }
}
```

they will make 0 to 0 only & nonzero to zero.

```
if (matrix[0][0]==0)
{
  //every one in 1st row will be 0 ie everyone in 1st row=0
  for(int j=0; j<m; j++)
    matrix[0][j]=0;
}
if(col0==0)
{
  //everyone in 1st col'm will be 0
  for(int i=0; i<n; i++)
  {
    matrix[i][0]=0;
  }
}
return matrix;
}
```

TC = O(n x m) + O(n x m)

SC = O(1) ← optimized

→ Rotate matrix/image by 90°-

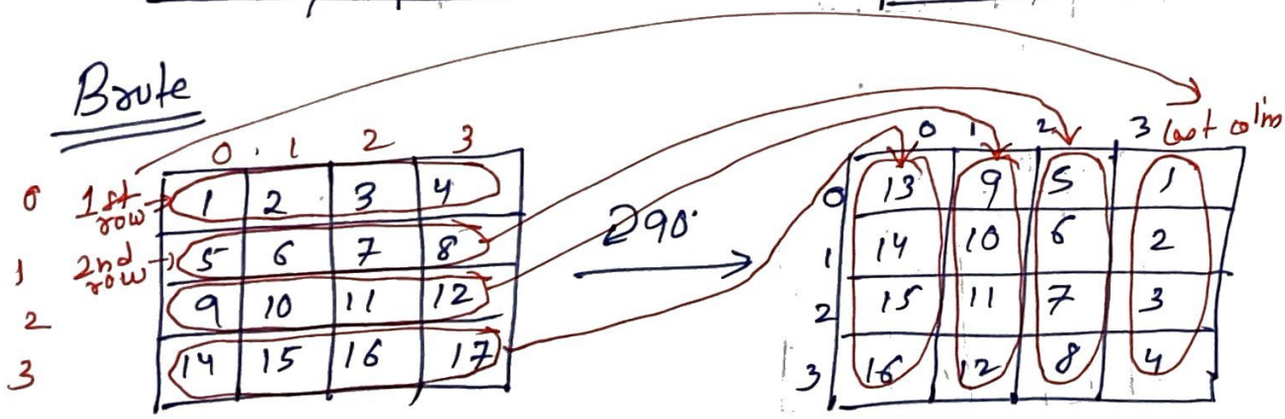
✓✓ Given n×n matrix, rotate it in clockwise direction by 90°

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

→ 90° →

13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

Brute



∴ for 1st row- (j =) 0 → n-1

[0][0] goes to [0][3]
[0][1] a [1][3]
[0][2] " [2][3]
[0][3] → [3][3]

(n-1)-0 ie (n-1)-i

for 2nd row- (j =)

[1][0] → [0][2]
[1][1] → [1][2]
[1][2] → [2][2]
[1][3] → [3][2]

(n-1)-i
≡ (n-1)-1

Declare → ans[n][n] matrix

```
for(i=0→n)
  for(j=0→n)
    ans[j][n-1-i] = matrix[i][j];
return ans;
```

TC = O(n²)
SC = O(n²) ← ans array

optimize it (optimal sol'n)

optimal – solve this problem in-place

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

→ 90° →

13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

① transpose ↓

	0	1	2	3
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

② reverse every row of ①

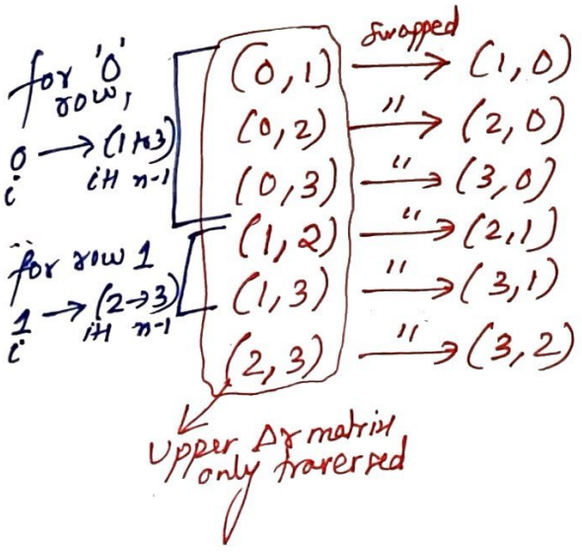
How to transpose –

	0	1	2	3
0	1	2 (swap)	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

diagonal is same in both (remain in same place)

transpose →

	0	1	2	3
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16



```
void rotate(vector<vector<int>> &mat)
{
    int n = mat.size();
    for(int i=0; i<n; i++)   // transpose
    {
        for(int j=0; j<i; j++)
        {
            swap(mat[i][j], mat[j][i]);
        }
    }
    //reverse every row
    for(int i=0; i<n; i++)
    {
        //every row is mat[i]
        reverse(mat[i].begin(), mat[i].end());
    }
}
```

this is for 90° clockwise (↻)

[agar 90° anticlockwise (↺), har column ko reverse karna hoga]

vector row wise har ← but column vector nai hai

$TC = O(n/2 * n/2) + O(n * O(n))$

for transpose; for every row; to reverse

$SC = O(1)$ → optimize as inplace

So,
1. first reverse every row of matrix
2. then, transpose

Count Subarray sum equals 'K'

↳ contiguous part of array

arr[] → {1, 2, 3, -3, 1, 1, 1, 4, 2, -3} & K = 3

here, Ans = 8 subarrays possible

i.e. [1, 2]
[1, 2, 3, -3]
[3]
[2, 3, -3, 1]
[3, -3, 1, 1, 1]
[1, 1, 1]
[4, 2, -3]
[-3, 1, 1, 1, 4, 2, -3]

Brute force

↳ generate all subarrays

```
cnt=0;
for(i=0; i<n; i++)
{
  for(j=i; j<n; j++)
  {
    sum=0;
    for(k=i to j){
      sum = sum + arr[k]
    }
    if(sum == k)
      ++cnt;
  }
}
```

→ TC ≃ O(n³)
SC = O(1)

Better

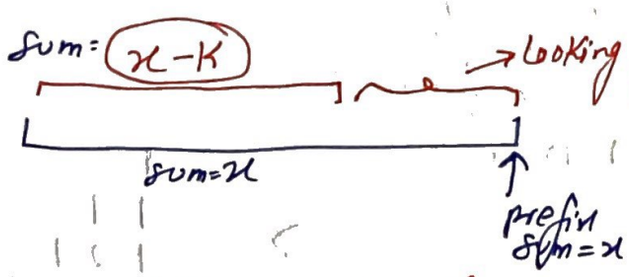
```
cnt=0;
for(i=0; i<n; i++)
{
  sum=0;
  for(j=i; j<n; j++)
  {
    sum += arr[j];
    if(sum == k)
      ++cnt;
  }
}
```

→ TC = O(n²)
SC = O(1)

Optimal

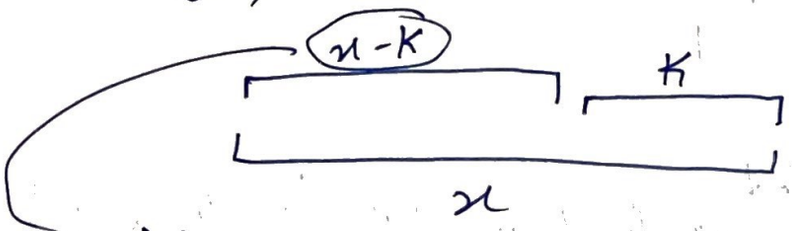
↳ use concept of prefix sum

(already discussed)



Use map (this concept is discussed)

here,



look for how many (x-K) is in map
↳ as it will tell utna subarray hoskta hai of sum = K

Dry run

arr[] → {1, 2, 3, -3, 1, 1, 1, 4, 2, -3} & K = 3

prefixsum = 0 1 3 0 1 4 5 6 10 12 9

3-3=0 present in map ∴ ↑cnt by count value of 0

3-3=0 present ∴ ↑cnt by value of count & count value = 2

6-3=3 ∴ cnt = 4+2 = 6 (count value of 3)

9-3=6 & 6 occurs twice in map ∴ 6+2 = 8 cnt value

no prefix sum = 3 occurs 2 times now

cnt = 0 1 2 3 4 → 4+2 → 6 → +2 → 8 (Ans)

key	count value
9	1
12	1
10	1
7	1
5	1
4	1
6	2
3	1
1	1
0	1

(prefixsum, cnt) ↓ key, ↓ count value

Initially put (0,1)

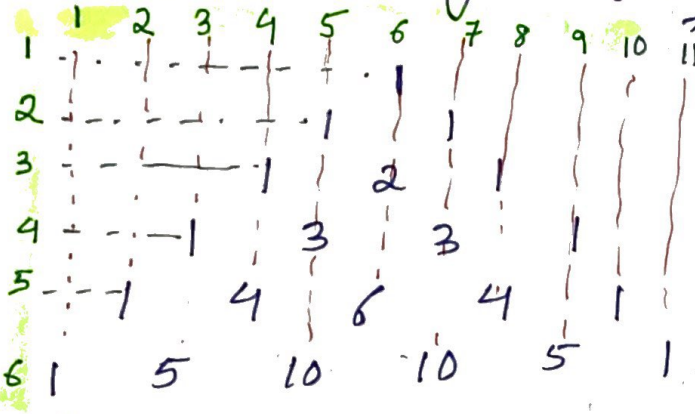
imp of (0,1)

```
int findAll (vector<int> arr, int K)
{
    map<int,int> m;
    m[0] = 1; //imp
    int presum = 0, cnt = 0;
    for(int i=0; i<arr.size(); i++)
    {
        presum += arr[i];
        int remove = presum - K;
        if(m.find(rem) != m.end())
        {
            cnt += m[remove];
        }
        m[presum] += 1;
    }
    return cnt;
}
```

TC = O(n * log n) → for ordered map
O(1) → for unordered map & in WC = O(n)

SC = O(n) ↳ if all prefix sum are unique

→ Pascal triangle [finding ^nC_r in minimal time]



⇒

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Question that can be asked -

① given row no. & col no., tell me the element at that place

say, R=5 & C=3
ans = 6

② print any nth row of pascal Δ.

say N=5
Ans = 1 4 6 4 1

③ print the entire Pascal Δ, given N

N=6

ans

① given row=R & col=C

do, $^{R-1}C_{C-1}$ is answer

Note: $^nC_r = \frac{n!}{r!(n-r)!}$

- for $^7C_2 = \frac{7!}{2! \times 5!} = \frac{7 \times 6}{2 \times 1}$

$^{10}C_3 = \frac{10 \times 9 \times 8}{3 \times 2 \times 1} = \frac{10}{1} \times \frac{9}{2} \times \frac{8}{3}$

$= 10 \times 4 \times 2$

PTO

for ^nC_r

```
int funNCR(n, r)
{
    long long res = 1; // for no overflow
    for(i=0; i<r; i++)
    {
        res = res * (n-i);
        res = res / (i+1);
    }
    return res;
}
```

if(r > n) return 0;

TC = O(r)
SC = O(1)

∴ for ①, call funNCR(r-1, c-1) & return it

② To print any given row-

if you see pascal △ in prev page,
1st row has 1 element
2nd " " 2 "
⋮
6th " " 6 "
∴ Nth row will have N element

Brute

```
for(c=1; c<=n; c++)
{
    cout << funNCR(N-1, c-1) << " ";
}
```

TC = O(N) * O(r) → for funNCR
= O(Nr)

Optimal

for 6th row-

col (say 0-based indexing (c))	0	1	2	3	4	5
	1	5	10	10	5	1
		$^5C_1$	$^5C_2$	$^5C_3$	$^5C_4$	$^5C_5$
	1	$= 1 \times \frac{5}{1}$	$= 1 \times \frac{5}{1} \times \frac{4}{2}$	$= 1 \times \frac{5}{1} \times \frac{4}{2} \times \frac{3}{3}$	$= 1 \times \frac{5}{1} \times \frac{4}{2} \times \frac{3}{3} \times \frac{2}{4}$	$\frac{1 \times 5 \times 4 \times 3 \times 2 \times 1}{1 \times 2 \times 3 \times 4 \times 5}$

ans * (row - col) / col

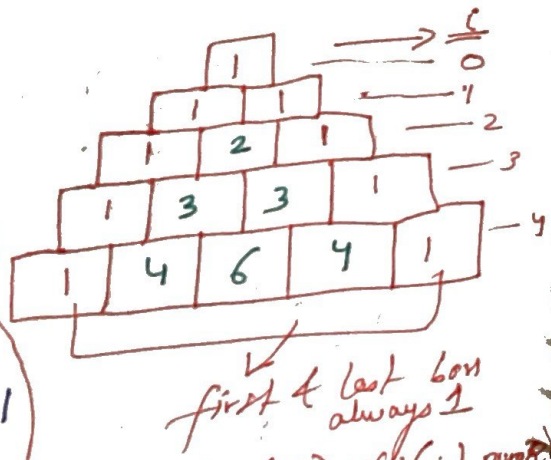
```
ans = 1
print(ans);
for(i=1; i<n; i++)
{
    ans = ans * (n-i);
    ans = ans / i;
    print(ans);
}
```

TC = O(N)
SC = O(1)

③ print entire Pascal △

N=6 ⟶ o/p:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```



first & last box always 1

will use ② to generate ③

```
vector<int> generateRow(int row)
{
    long long ans=1;
    vector<int> ansRow;
    ans.push_back(1);
    for(int col=1; col<row; col++)
    {
        ans = ans * (row-col);
        ans = ans/col;
        ansRow.push_back(ans);
    }
    return ansRow;
}

vector<vector<int>> Pascal(int N)
{
    vector<vector<int>> ans;
    for(int i=1; i<=N; i++)
    {
        vector<int> temp = generateRow(i);
        ans.push_back(temp);
    }
    return ans;
}
```

```
vector<vector<int>> nCr(int numR)
{
    vector<vector<int>> ans;
    for(i=0; i<numR; i++)
    {
        vector<int> v(i+1, 1);
        for(j=1; j<=i-1; j++)
        {
            v[j] = ans[i-1][j]
                 + ans[i-1][j-1];
        }
        ans.push_back(v);
    }
    return ans;
}
```

as first & last box by already '1'

$$TC = O(n) * O(n) = O(n^2)$$

(for loop, function)

$$SC = O(N * \text{col})$$

(#row, #col)