MEDIUM

2 SUM : check if a pair with given sum exists in Array.

arr = {2, 6, 5, 8, 11} target = 14

result = [1, 3]

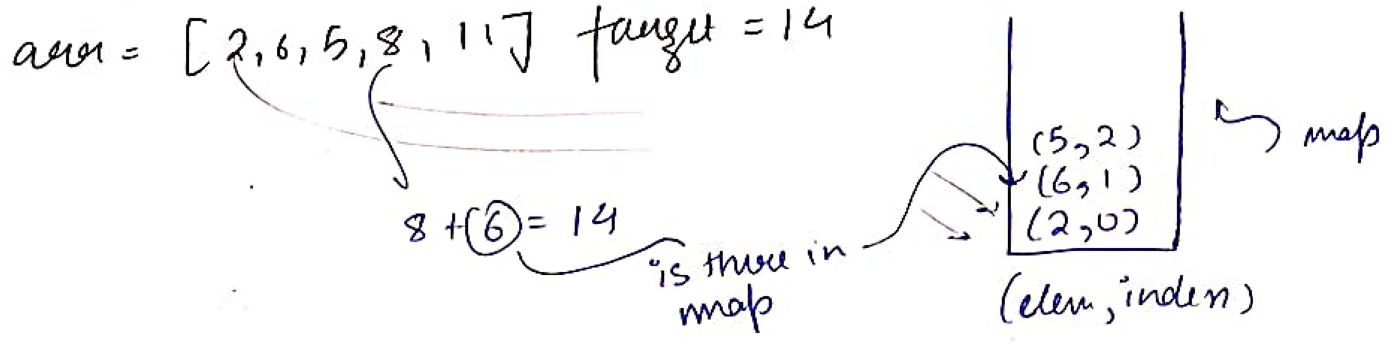
BRUTE

Pick one element scan the remaining array for (target - n). Repeat

```
for(i = 0 → n){
        for(j = i+1 → n){
                if(arr[i] + arr[j] == target){
                        return [i, j]
                }
        }
}
```

$O(n^2)$

BETTER (Hashing)



```
map<int, int > mpp;
for ( i:0→ n){
        int a = arr[i];
        int more = target - a;
        if( mpp.find(more) != mpp.end()){
                    return [i, map.second];
        }
       else map.put [arr[i], i]
    }
 }
```

O(n) - average

O(n²) - worst but rare

space complexity - O(n)

OPTIMAL is space

2 pointer

arr[] = {2, 6, 5, 8, 11}

↓

Sort

↓

arr[] = {2, 5, 6, 8, 11} — when (-8)

i → ←j

when (+)

i = 2 j = 11 2+11 = 13 less < 14 ∴

i++ = 5 j = 11 5+11 = 16 >14 ∴

i = 5 --j = 8 5+8 = 13 < 14 ∴

++i = 6 j = 8 6+8 = 14 ~~wowo~~ wow

```
for(i—0→n){
    sum += a[i];

    if(sum == k){
            maxlen = i+1
        }

    long rem = sum -k;
    if (presum.containskey(rem)){
            int len = i-presum.get(rem);
            maxlen = Math.max(maxlen, len);

        }

    if(! presum containkey(sum)){
                presum.put(sum, i);
        }
}
```

calculating sum till i

if prefix sum is =k then max length would be that

checking if (n-k) exist in map or not

updating sum if it does not exists

Time complexity → $O(\underset{\text{traversing}}{n} \times \underset{\text{finding in ordered map}}{\log N})$

or $O(\underset{\text{traversing}}{n} \times \underset{\text{finding in unordered map}}{1})$

but $O(n \times n)$ ∴ → at worst case

Space complexity:

Edge case

arr[] → [2, 0, 0, 3]   K = 3

presum = ~~0~~ 2

len = 0

= 2+0 = 2

= 2+0 = 2

= 2+3 = 5

2, ~~0~~ 1 | 2

∴ to have 3 as sum in subarray we need n-K ⇒ 5-3 = 2 sum previously which exist in map but index "1" [~~0~~2, 0, 0] [3]

n-K   K

∴ length of subarray ~~& K sum~~ whose sum is K = 1 which is wrong as it gives minimum length but answer should be [0, 0, 3] = len = 3

∴ we should not update sum when going to new element which is zero gives same sum.

CODE

```
int n = a.length;
Map<long, integer> presum = new HashMap<>();
long sum = 0
int maxlength = 0;
```

```
  if (sum == k) {
    maxlen = Math.max(maxlen, right - left + 1);
    }
   right ++;
   if (right < n) sum += a[right];
  }
return maxlength;
}
```

Now inner while loop will not run "n" times

as $\underset{0}{\overset{1}{\downarrow}}$ $\underset{0}{\overset{2}{\downarrow}}$ $\underset{0}{\overset{3}{\downarrow}}$ $\underset{3}{\overset{4}{\downarrow}}$ $\underset{0}{\overset{5}{\downarrow}}$ $\underset{2}{\overset{6}{\downarrow}}$ = 5

as sometime while do not even get executed ∴ it doest not run n times for every element but overall it runs n time (~~summation~~) summation

∴ O(n) +

∴ O(n) + O(n) = O(2n)

In case of (−) dry run this

→ {2, −1, 2, 3, −2, 4} k = 4

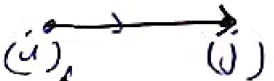
for optimal ✗ and hashing ✓ solution

here if sum crosses k then we have a solution to include next number which can be negative that will make the sum = k but in optimal approach we just decrease the window from left.

OPTIMAL when only +ve and 0s   use prefix when ~~no~~ (-) included

2 pointer approach.

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



~~count upto~~

increase ⎵ upto K = 6 ~~on~~

same length

now if sum > K then
decrease length from left side

# increase from right to get to K
and decrease from left to if sum > k

[1, 2, 3, 1
(i) ⟵ (j) sum = 7

[1, 2, 3, 1
(i) ⟶ (j) sum = 6

O(n)

```
fun (int a[], K){
  int n = a.length;
  int left =0; right = 0;
  long sum = a[0];
  int maxlength =0;
  while(right <n){

    while (left ≤ right && sum > k){
        sum -= a[left];
        left++
    }
```

keep decreasing size as long as sum > K

MAJORITY ELEMENT (> n/2 times)

BRUTE

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

2 = (4times > 7/2)

pick element search the array increase count

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

O(n²)

BETTER (hashing)

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

```
int n = arr.length;
Hashmap mapp;
for (i=0 → n) {
      int value = mpp.get (v[i])
      mmap.put (v[i], value + 1);
}
```

O(n) × 1 when unordered map

↓

when ordered (O(n × ~~a~~)) logn

(1, 1)

(3, ~~1~~, 2)

(2, ~~1 2 3~~ 4)

(ele, count)

but if you need index as answer then put each element in another data structure {(2,0) (6,1) (8,3)} as index will be disturbed after sorting. ∴ only "yes" or "no" can be answered here.

```
read( ... ){
        int left = 0; right = n-1;
Arrays.sort(arr);  ←—— O(nlogn)
        while(left < right){
            int sum = arr[left] + arr[right];
            if(sum == target){ return "yes";
            else if(sum < target) left++;
            else right --;                ←—— O(n)
        }
    return "NO";
}
```

time complexity : O(n) + O(nlogn)

space complexity : O(1)

```
for (Map.Entry<Int, Int> it : mpp.entrySet()){
    if (it.getValue() > n/2)){
        return it.getKey();
    }
}
```

O(n)

time - O(n) + O(n) - O(2n)

space - O(n) - when unique elements

OPTIMAL (Moore's Voting Algorithm)

arr = [7, 7, 5, 7, 5, 1, 5, 7, 5, 5, 7, 7, 5, 5, 5, 5]

this works on logic that to cannot ~~cancel~~ out a majority element as consider arr = [1, 1, 1, 2, 2] = $\frac{5}{2}$ = 2 there majority element count is 3 > $\frac{n}{2}$ therefore we need 3 element to cancel out of majority element but it is not possible as already majority covers more than $\frac{n}{2}$ space of array i.e. not other element combined can cancel out majority element other element ≠ 7 ∴ cancel out]

arr = [7 7 5

element = 7

count = ~~0~~ ~~1~~ ~~2~~ 1

arr = [7 7 5 7

elem = 7

count = 1 + 1 = 2

when element (current) = picked element

↓

increase count

else decrease count

arr = [7 7 5 7 5 1 7

elem = 7

count = ~~2~~ ~~1~~ 0

we can say that in ths subarray there is no majority element as the count becomes 0. which means no element have cont > $\frac{n}{2}$

Now

pick next element

arr = [7 7 5 7 5 , 5 7 5 5, 7 7, 5, 5, 5, 5]

element = 5

count = ~~1~~ 0

element = 5

count = ~~1~~ ~~2~~ ~~1~~ 0

element = 5

count = ~~1~~ ~~2~~ 3 $\boxed{4}$

as count ≠ 0 ∴ element = 5 will be majority elem

kyuki kisi aur mein dum nahi use maarne ka.

```
majority(arr){
  int n = arr.length
  count = 0
  el = 0
  for(i=0 → n){
      if(count=0){
          ele = v[i];
          count = 1;
      }
   else if( el == v[i]) count++;
   else count--;
  }
  int count1 = 0
  for(i=0 → n){
      if( arr[i] == ele ) count++;
  }
 if (count1 > (n/2)) return el;
 return -1;
}
```

algo... O(n)

check if that is majority element or not (case when there is no majority element)

O(n)

time – O(2n) when there can be or cannot be a majority element

O(n) – when it exists always

space – ~~NO~~ O(1)

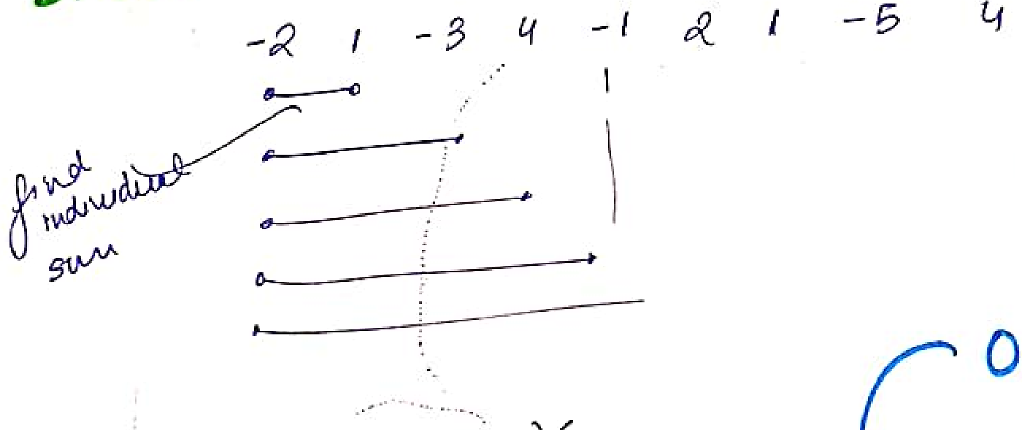
MAXIMUM SUBARRAY

Given an integer array nums, find contiguous subarray (at least on number) which has largest sum and return its sum

arr = [-2, 1, -3, $\underline{4, -1, 2, 1}$, -5, 4]

output : 6

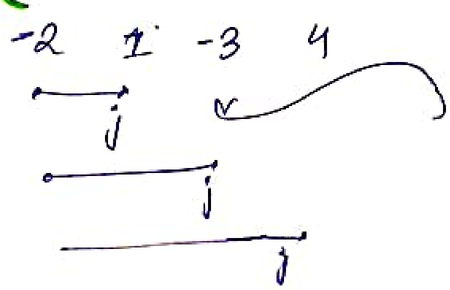
BRUTE



O(n³)

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

BETTER



here j is moving only one place but we are recalculating whole sum

∴ we can just add extra element only to existing sum

```
for( i=0 → n) {
        sum = 0;
        for( j = ~~0~~ i → n) {
            sum += a[j];
        }
        max = Max(sum, max);
}
```

$O(n^2)$

OPTIMAL (Kadane's Algorithms)

intusion: we carray a subarray sum as long as it gives us a (+) sum

-2 -3 4 -1 -2 1 5 -3

sum = -2

max = -2

Now after going to another element we check if sum is negative before then up we update (sum = 0) as.

∴

-2 -3 (4) -1 -2 +1 5 -3

sum = ~~2~~ 0 + -3 = -3

update

update

= 0 + 4 = 4

keep as (+)

= 4 - 1 = 3

3 - 2 = 1

1 + 1 = 2 → 2 + 5 = 7 - 3 = (4)

max = -2

max = 4

(max = 7)

G+
52

```
maxSubarray( arr ) {
    int sum=0;
    int maxSum = 0;
    for(i=0 -> n){
        sum += arr[i];
        maxSum= max(sum, maxSum);
        if (sum <0) sum=0;
    }
return maxSum;
}
```

num
if we need index of array to print

```
maxSubarray (arr){
    int sum =0;    int ansStart;  int ansEnd;
    int maxSum=0;
        ~~start = -1;~~                local variable
    for( i=0 -> n ){
        if (sum==0) start = i
        sum ± arr[i];
        if (sum> maxSum){
            ansStart = start;
            ansEnd = i;
            maxSum = sum;
        }
    if (sum <0) sum =0;
}
```

SORT ARRAY OF 1S, 0S AND 2S.

BRUTE - Sort → TC (

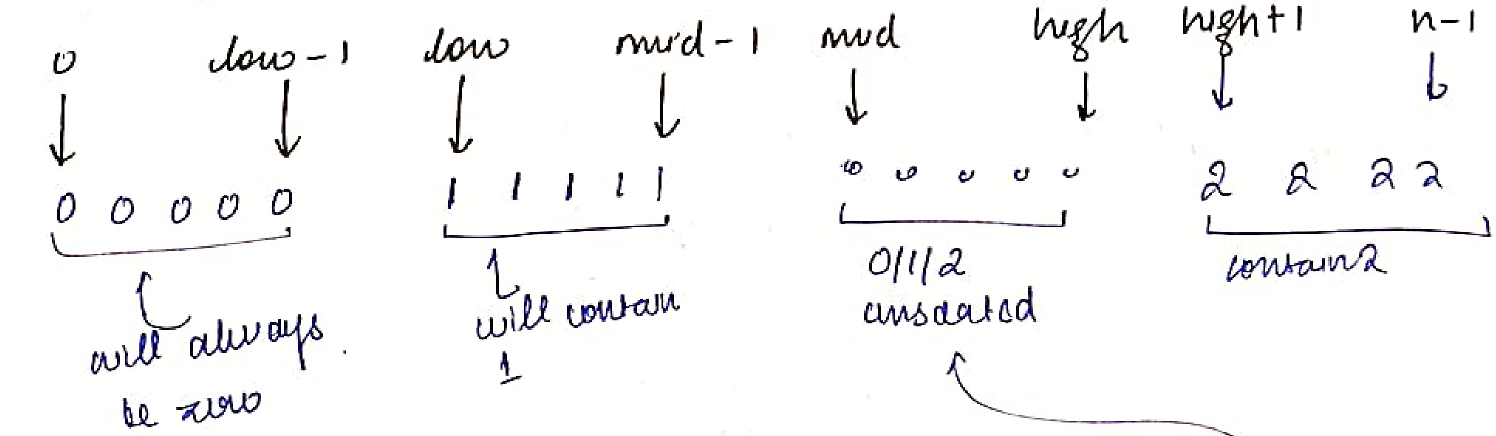
BETTER - keep count of all these no → 0 count
1's count
2's count

iterate array count
~~with~~ reiterate array and overwrite
O(2N)

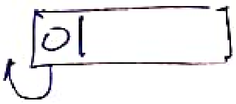
OPTIMAL - Dutch National Flag Algorithm.

arr = 0 1 1 0 1 2 1 2 0 0 0

Rules



point ki ya baat shuru main poore array unsorted
∴ when we discover any (no) then a by
using low and high pointer we place them there



0 → tu low ke paas jaa
1 → tu mid -1 pe reh
2 → hight ke pass ja

low

if>0

arr = [0 1 1 0 1 2 1 2 2 0 0]

mid high

a[mid] == 0 { if mid == 0 then we have to put in place of low which contains ~~ar~~ "1"; ∴ we ⊗

swap (arr[mid], arr[low])

low++; as it is sorted .

mid++; as 1 is there and array sorted upto 1

∴ unsorted array after mid-1;

}

arr[mid] == 1 { mid++; as 1 is at correct place ∴ unsorted array is after 1 }

arr[mid] == 2 { swap(~~na~~ arr[mid], arr[high]);

∴ upto ~~up~~ high to n-1 sorted

∴ high--;

and whatever value comes to arr[mid] after swapping is unsorted

∴ not change;

}

low
arr = [0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0]
mid high

~~swap~~ arr[mid] == 0 case.

low
arr = [0, 1, - -
mid

arr[mid] = 1
low
arr = [0, 1, 1,
mid

arr = [0, 1, 1, 0
low mid

arr = [0, 0, 1, 1, 0
low mid

arr = [0, 0, 0, 1, 1, 2
low mid

high and mid swapped

arr = [0, 0, 0, 1, 1, 0, 1, 2, 0, 0, 2]
low mid high

check mid apply rule

arr = [0, 0, 0, 0, 1, 1, 1, 2, 0, 0, 2]
low mid high

arr = [0 0 0 0 1 1 1 0, 0 2 2]
low mid high

arr = [0 0 0 0 0 1 1 1 ,0 , 2 , 2 , 2]

low high mid mid

arr = [0,0,0,0,0,0,1,1,1,1,2,2,2]

low high

```
{
low = 0, high = ~~0~~ n-1 ; mid = 0 ;

while (mid ≤ high ){
        if (arr[mid] == 0) {
                swap(arr[mid], arr[low]);
                mid++;
                low++;
        }
    else if (arr[mid] == 1) { mid++; }
    else { swap(arr[mid], arr[high]);
           high--;
    }
}
```

O(n)

STOCK BUY AND SELL

you want to maximize your profit by choosing a single day to buy on stock and choosing a different day in the future to sell stocks. Return max profit if you cannot make profit return 0;

int prices = [7, 1, 5, 3, 6, 4]

output - 5

BRUTE

```
        j →
    i ↷ j →
[7, 1, 5, 3, 6, 4]
 i → i → i
```

O(n^2)

OPTIMAL

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

minelement = ~~7~~ 1 → 1 → 1 → 1 → 1
profit = ~~0~~ 0 4 → ~~4~~ 5 → 5
                 ↓          ↓
               (5-1)      (6-1)
```

```
maxProfit(prices[]) {
    maxPro = 0
    minPrice = IN_MAX;
    for (i=0 → n) {
        minPrice = min(minPrice, prices[i]);
        maxPro = max(maxPro, Prices[i] - minPrice);
    }
    return maxPro;
}
```

O(1)

REARRANGE BY SIGN

Array with equal number of +ve and -ve elements.
without ordering the relative order of +ve and -ve number
return an array of alternate +ve and -ve

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

{1, -4, 2, -5}

BRUTE

Take +ve array
Take array for -ve number
fill by passing through array ← $O(n)$
overwrite original array.

-> even index = +ve
-> odd index = -ve

filling array

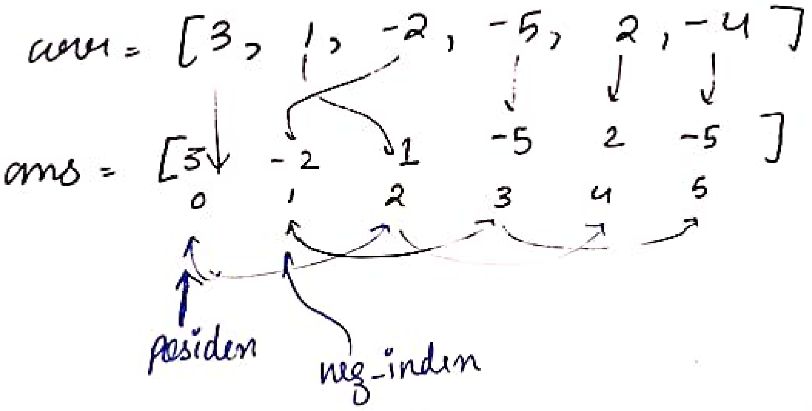
```
for (i=0 → n/2) {
        arr[2×i] = pos[i];
        arr[2×i+1] = neg[i];
}
```

← $O(\frac{n}{2})$

time - $O(n + \frac{n}{2})$

space → $O(n)$

OPTIMAL



when we will encounter an element put elements on respective index and increment appropriately in ans array

return ans array.

TC → O(N)

space → O(N)

```
rearrange (nums) {
  int ans[nums.length];
  int posIndex; negIndex;
  for(i=0 → n) {
        if (nums[i] < 0) {
                      ans[negIndex] = nums[i];
                      negIndex += 2;
              }
        else {
                 ans[posIndex] = num[i];
                    posIndex += 2;
              }
     }
}
```

Variation - if some elements are left put them in end without altering the order.

} answer = fall Back to brute force method.

```
                 pos
ArrayList<Integer> pos = new ArrayList<>();
   "               neg =  "      "          O(n)

   for (i=0 -> n){
       if(arr[i] > 0) pos.add[arr[i]];
        else neg.add[arr[i]];
                                        O(Min(pos, neg))
 if (pos.size() < neg.size()){
        for (i=0 -> pos.size()){
              arr[i×2] = pos.get(i);
              arr[2×i+1] = neg.get(i);
        }
                             -> index from where to
                                fill remaining element
   int index = pos.size()×2;//

     for(i= pos.size(); i< neg.size(); i++){
           arr[index]= neg[i];

     }
                                      O(remaining)
 else{

                  similar above

 }
```

∴ O(n)+ O(~~Max~~ Min (pos, neg)) +O(rem)

when ~~all +ve~~ or -ve then → O(n)

∴ O(n) + O(n) = O(2n)

NEXT PERMUTATION

Arr = {1, 3, 2}

output → {2, 1, 3} ← next permutation after above

BRUTE

Generate all permutation find next permuation using linear search.

arr[] = [3, 1, 2]

1 2 3
1 3 2
2 1 3
2 3 1
$\boxed{3\ 1\ 2}$
3 2 1

Just learn how to find using recursion

3 number = 3! = 6

$O(n! \times n)$

n! → permuation

n → every permutation of n size

OPTIMAL

arr = [2 , 1 , 5, 4, 3, [illegible], 0]

for next permution this is sorted in descending order ∴ arranging these would not help in getting next bigger number than current.

∴ step: find a a[i] < a[i+1] because if we re arrange now we can make the next greater number

step2: find nums > 1 but smallest one. now if any element is picked eg 2 5 but 2 3 was should come befor ∴ we need to find next greater no than 1 in right part

Step 3: Now arrange the remaining in ascending order so that ~~big~~ next number can be formed by arranging inner element first

```
indx = -1;

for (i = n-2; i >= 0; i--) {
        if (a[i] < a[i+1]) {
                indx = i; break
        }

if (index == -1) reversearray(arr); return;
```

[2, 1, 5, 4, 3, 0, 0]
indx

as reached as last one i.e array already in descending order.

```
for (i = n-1; i >= index; i--) {
        if (arr[i] > arr[indx]) {
                swap(arr[i], arr[indx])
        }
```

[2, 1, 5, 4, 3, 0, 0]
[2, 3, 5, 4, 1, 0, 0]

```
reverse (arr, index+1, n-1);
```

[2, 3, $\underbrace{0, 0, 1, 4, 5}_{\text{reversed.}}$]

LEADERS IN AN ARRAY

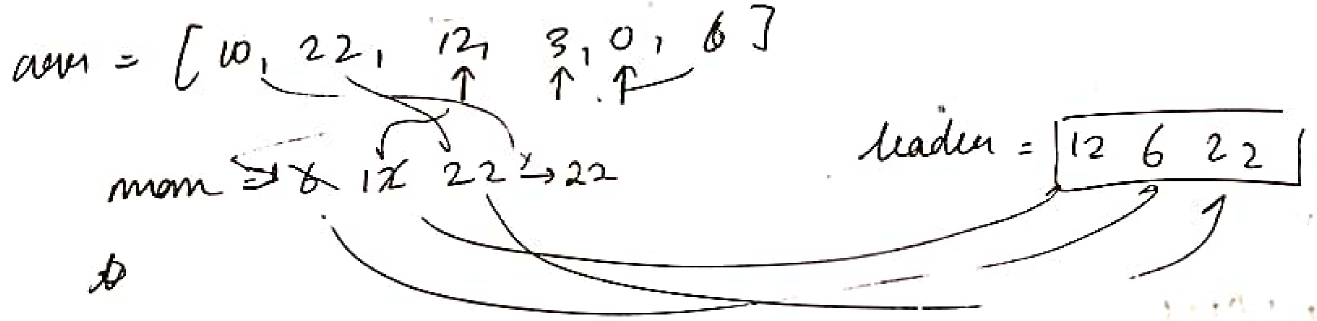
→ A leader is an element that is greater from all of the elements on its right side in the array.

→ rightmost element is always the leader.

→ print all leaders.

OPTIMAL

◦ we can on encountering an element compare it with max element of right side instead of all elements on right update max when a real leader is found.



```
int max = arr[n-1];
print(max);
for(i=n-2 ⟶ 0){
            if (arr[i] > max){
                    max = arr[i];
                    print(arr[i]);
        }
```

O(n)

BRUTE (FIND LONGEST CONSECUTIVE SQU)

sequence

arr = [100, 200, 1, 3, 2, 4]

length = 4

arr = [3, 8, 5, 7, 6]

length = 4 as [5, 7, 6, 8] rearrangement can be done.

arr[] = [102, 4, 100, 1, 101, 3, 2, 1, 1]

Pick one do linear search for next consecutive element increase count if any (repeat)

```
for (i=0 → n) {
    x = arr[i];
    count = 1;
    max = ~~0~~1;
    while ( ~~arr~~ linearSearch(arr, x+1) == true ) {
            x = x+1;
            count = count + 1;
    }
    max = Max(count, max);
}
```

BETTER

Sort element first then count:

sort(arr); ⟵ O(nlogn)

```
longest = 1 ;   lastSmaller = INT_MIN,   count = 0

    for ( i=0 → n ) {
        if ( arr[i] - 1 == lastSmaller ) {
                count = count + 1 ;
                lastSmaller = arr[i] ;
        }
        else if ( arr[i] != lastSmaller ) {
                count = 1 ;
                lastSmaller = arr[i] ;
        }
        longest = max (longest, count )
    }
```

1 2

1 2 8

!=

also if

1 2 2

no count increase

O(n)

time - O(n) + O(nlogn)

but we are modifying array.

∴

OPTIMAL

arr = [102, 4, 100, 1, 101, 3, 2, 1, 1]

→ use set

102
4
100
101
3
2

set

Now iterate element by element

we should start from here

→ 102 now do not start from here as 102 can 100

∴ ~~search for ne smaller consecutive~~

∴ move one step down

→ 4 now 3 exist in set ∴ has no right element to start ∴ counting with ∴ move down

→ 100 now 99 do not exist ∴ start with 100 and count

100
101 } count = 3
102

∴ Repeat:

~~5~~
→ 1
2
3 } 4
4

∴ 100 101 → go down

∴ 3 → 2 → go down

∴ 2 → 1 → go down

```
→ { int n = a.length;
    if (n==0) return;
    int longest = 1;
    set <Integer> set = new HashSet<>();
    for (i=0 → n ){ set.add (a[i]) ; }
    for (int it : set ) {
        if (! set.contains (it - 1 ) ){
            int cnt = 1 ;
            int n = it ;
            while ( set.contain(n+1 ) ){
                n = n+1;
                count = count + 1;
            }
            longest = Math.max (longest, count );
        }
}
```

O(n)

if it contains previous one or not

O(2N) as N + n

element

(7+7)

a more iteration for sequence length (1,2,3,4), (100,101,102)

O(3N) → time

O(N) → space

SET MATRIX ZEROS

Given a matrix if an element in the matrix is 0 then you will have to set its entire column and row to 0

ex

$$\begin{matrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{matrix} \longrightarrow \begin{matrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{matrix}$$

BRUTE

$$\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & \cancel{0}1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{matrix} \longrightarrow \begin{matrix} 1 & -1 & -1 & 1 \\ -1 & 0 & 0 & -1 \\ -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & 1 \end{matrix}$$

STEP 1

mark row and colum value as $\boxed{-1}$ whenever we find 0 and do not chang 0 to -1 in between

↓

STEP 2

mark change -1 to 0 in next iteration

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

O(n×m)

O(m)

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

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

O(n)

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

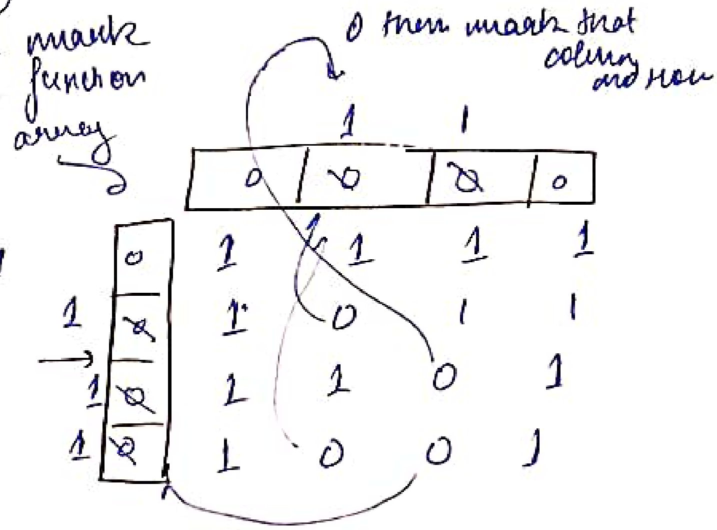
O(m×n)

∴ time - $O((n \times m) \times (n+m) + m \times n) \approx$ cubic

for mark function

BETTER

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



Now we are running mark function for same row and column every time we encounter a zero in that row/column → now → but not whenever we will get a 0 we will only mark that column to set set to 0 at end in the end the marked column/row will be set to 0

```
col[m] = {0} , row[n] = {0}

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

O(n×m)

```
for(i=0 → n){
    for(j=0 → m){
        if(row[i] || colu[j]) arr[i][j] = 0
    }
}
```

O(n²)

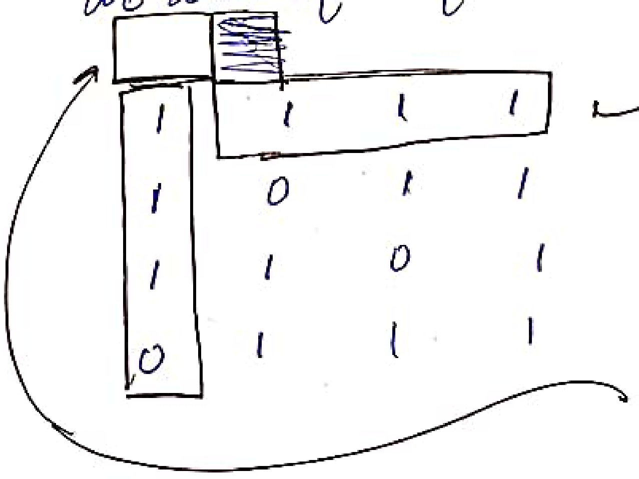
O(n×m)

time - O(2(n×m))

space → O(n) + O(m)

OPTIMAL

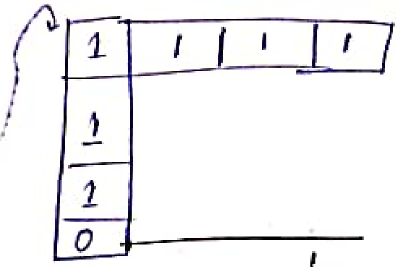
Thought process → how to reduce the space complexity eliminate
we will store the array in the matrix itself.



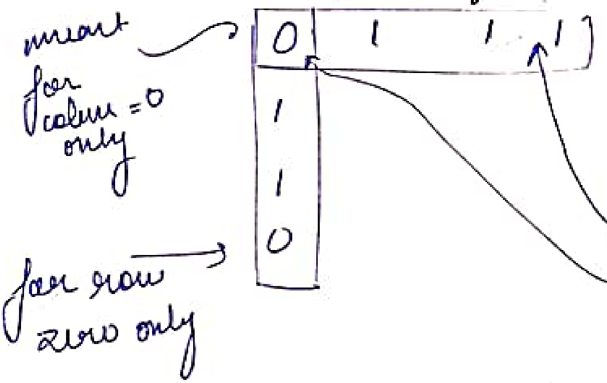
1 | 1 1 1
1 | 0 1 1
1 | 1 0 1
0 | 1 1 1

this will be used to mark column

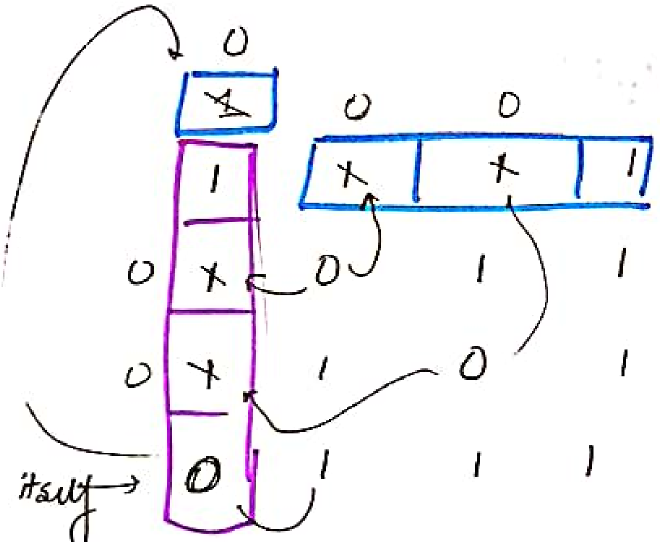
three extra variables used as there there will be overlap between row 0 and column 0 tracking as



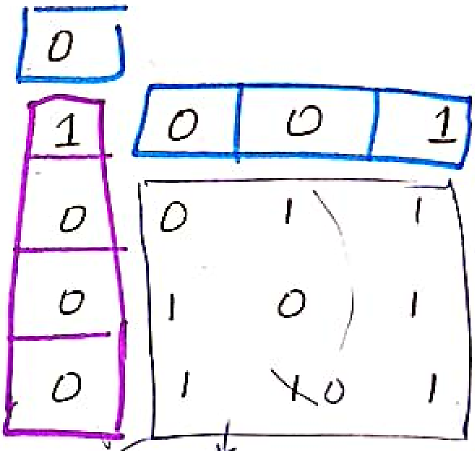
This is 0 ∴ will be zero and this column will be zero if no extra variable then the matrix will look like



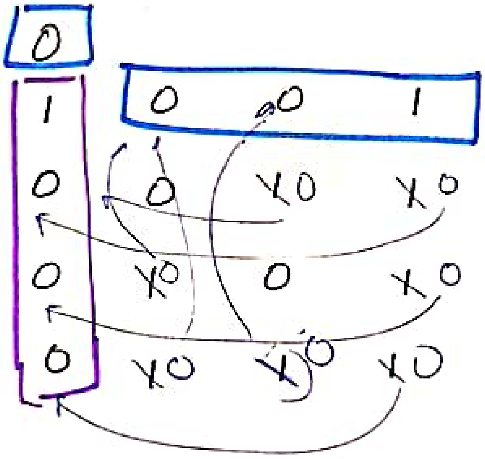
But when we will get to this element it will look here as it is 0 it will turn to 0 which should not be done as it was meant for column not row. ∴ extra variable



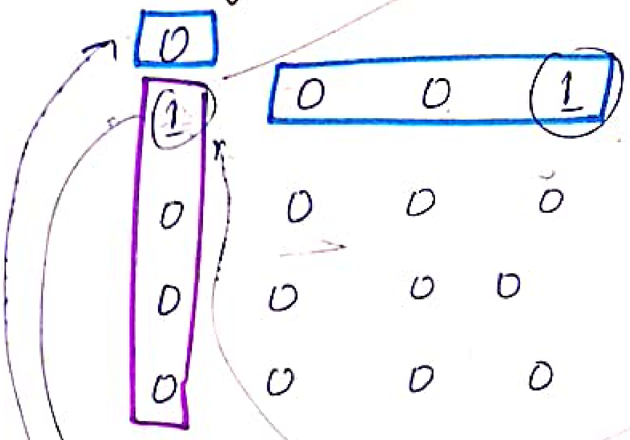
will work for as row but for column



now ~~[illegible]~~ will start iterating from the inner elements only as iterating through array stored as matrix would disturb its own element therefore making unnecessary elements 0



Now we iterate through column array first then 71
now array.



→ is dependent on this value therefore if we traverse now array first

this one will be converted to 0 as this is 0 and whenever we will solve column array every element will become zero as this will be 0

∴ we start from colum array

```
int col0 = 1;  ↝  var for column "0"

for (i=0 ⟶ n) {
    for (j=0 ⟶ m) {
        if (matrix[i][j] == 0) {
            matrix[i][0] = 0;
            ~~matrix[0][j]~~ = 0;
            if (j != 0) {
                matrix[0][j] = 0;
            }
            else col0 = 0;
    }
}
```

O(m×m)

marking for row

ensuring that your column variable "col0" is used in in 0 column

```
for(i=1 → n){
    for(j=1 → m){
            if(matrix[i][j] != 0){
                    if(matrix[0][j] || matrix[i][0] == 0){
                        matrix[i][j] = 0;
                    }
            }
    }
}
if(matrix[0][0] == 0){
        for(j=0 → m) matrix[0][j] = 0;
}
if(col0 == 0){
        for(i=0 → n) matrix[i][0] = 0;
}
```

iterating inner element

checking if column 0 row needs to be set to 0

setting column 0 to 0 if variable says so.

O(m×n) → combined

time → O(2×(m×n))

space → O(1)

ROTATE IMAGE BY 90°

$$\begin{matrix}1&2&3\\4&5&6\\7&8&9\end{matrix} \xrightarrow[\text{size}]{[n \times n]} \begin{matrix}7&4&1\\8&5&2\\9&6&3\end{matrix}$$

BRUTE

→ use an external array

Now

$$\begin{tikzcd}
{[0][0]} \arrow[r] & {[0][3]} \\
{[0][1]} \arrow[r] & {[1][3]} \\
{[0][2]} \arrow[r] & {[2][3]} \\
{[0][3]} \arrow[r] & {[3][3]}
\end{tikzcd}$$

$0 \rightarrow 3$

$\downarrow \quad \downarrow$

$(i) \quad (n-1)-i$

i j

same here as (i) similar for other rows.

$$\begin{tikzcd}
{[1][0]} \arrow[r] & {[0][2]} \\
{[1][1]} \arrow[r] & {[1][2]} \\
{[1][2]} \arrow[r] & {[2][2]} \\
{[1][3]} \arrow[r] & {[3][2]}
\end{tikzcd}$$

$① \rightarrow ②$

$i \quad \uparrow \quad (n-1)-i$

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

✓ $O(n^2)$

but !! space also $O(n^2)$

OPTIMAL

Now

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

find transpose →

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

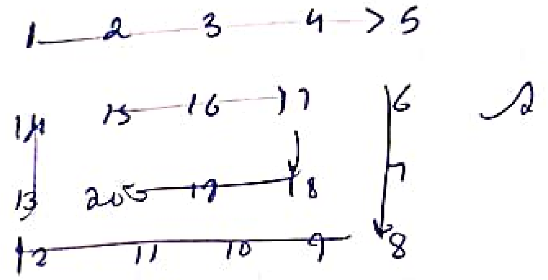
↓ reverse each row

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

kya baat hai!! answer aa gya.

```
for (i=0 → n) {                         ← transpose
    for (j= i+1 → n) {
        swap(arr[i],[j], arr[j][i]);
                                        ⟵ O(n/2 × n/2)
    }
}

for (i=0 → n) {                         → O(n × n/2)
    int start = 0 ; end = n-1;
    while (start < end) {
        swap (arr [i] [start], arr [j] [end]);
    }
    start ++ ; end --;
    }
}
```

SPIRAL MATRIX



left   righ

$$\begin{array}{c|cccccc} \text{top} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 20 & 21 & 22 & 23 & 24 & 7 \\ 2 & 19 & 32 & 33 & 34 & 25 & 8 \\ 3 & 18 & 31 & 36 & 35 & 26 & 9 \\ 4 & 17 & 30 & 29 & 28 & 27 & 10 \\ 5 & 16 & 15 & 14 & 13 & 12 & 11 \end{array}$$

bottom

top = 0   bottom = 5

left = 0   right = 5

while (top ≤ bottom && left ≤ right)

inside

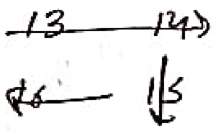
}

```
for (i = left → righ) {
        arr[top][i];
        }

~~for~~(   top++;
 for( i = top → bottom) {
        print(arr[i][right]
        }

    right --;
 for( i = right → = left ) {
        print(arr[bottom][i] }

    bottom++;
 for ( i = bottom → = top ) {
        print (arr[i][left])
        }
    left ++;
```

```
while ( top ≤ bottom && left ≤ right ) {

    for ( i = left ——> = right ) print ( arr[top][i] );

    top ++ ;

   for ( int i = top ——> = bottom ) print ( arr[i][right] )

   right -- ;

  if ( top ≤ bottom ) {
          for ( i = right ——> = left ) { print ( arr[bottom][i] )

          bottom -- ;
          }
  if ( left ≤ right ) {
  for ( i = bottom ——> top ) { arr[i][left]

   left ++ ;
       }
}
```

when only single row given

no top here



no need to print 13 again.

$O(n \times n)$