#1. Two Sum:

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.
You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.
You can return the answer in any order.

From <https://leetcode.com/problems/two-sum/>

[2,7,11,15], target = 9 // [0,1]

[3,2,4], target = 6 // [1,2]

// Brute force approach:

- use two loops, outer loop to select an element, inner loop to find the pair which sums up to target.

```
vector ans;
for(i:0 ⟶ n){
   for(j: i+1 ⟶ n){
      if(a[i] + a[j] == sum){
         ans.push_back(i);
         ans.push_back(j);
      }
   }
}
return ans;
```

time complexity: O(n²)
space complexity: O(1)

// Better Solution: (Using hashing)

- iterate through the array, for each element check if (target - element) is present: return the index of current element & the rem part from the hash map.

otherwise: store the (element, index) in map.

Better Solution:

```
map<int, int> mp;
for(i:0 ⟶ n){
    if(mp.find(target - a[i]) != mp.end())
        return {mp[target - a[i]], i};
    mp[a[i]] = i;
}
return {-1, -1};
```

time complexity: O(n) OR O(nlogn)
space complexity: O(n)

// Optimal Solution: (two pointer approach)

- Only valid if you wont return "YES" or "NO"

[2, 7, 11, 15] target = 9

sort the array,

[2, 7, 11, 15]
(0, 1, 2, 3)

l r ← r ← r

- sum = 2 + 15 = 17 > target
  r--;
- sum = 2 + 11 = 13 > target
  r--;
- sum = 2 + 7 = 9 == target return {l, r}

This solution can work if the given array is already sorted.

while (left < right)

- if the sum would have been smaller than target, we would have incremented l++.

```
sort(array);
l = 0, r = n-1;
while(l < r){
    if(a[l] + a[r] == target) return "YES"
    else if(a[l] + a[r] < target) l++;
    else r--;
}
return "NO";
```

time complexity: O(nlogn) + O(n)
space complexity: O(1)

return "NO";
}

space complexity: O(1)

#167. Two Sum II- Input Array is Sorted:

Given a **1-indexed** array of integers numbers that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 < numbers.length.
Return *the indices of the two numbers,* index1 *and* index2, ***added by one*** *as an integer array* [index1, index2] *of length 2.*
The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.
Your solution must use only constant extra space.

From <https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/description/>

[2, 7, 11, 15] target = 9

sort the array,

0	1	2	3
2	7	11	15
l		r	r

- sum = 2 + 15 = 17 > target
  r--;
- sum = 2 + 11 = 13 > target
  r--;
- sum = 2 + 7 = 9 == target return {l, r}

This solution can work if the given array is already sorted.

while (left < right)

- if the sum would have been smaller than target, we would have incremented l++.

```
sort (array);
l = 0, r = n-1;
while (l < r) {
  if (a[l] + a[r] == target) return {l+1, r+1};
  else if (a[l] + a[r] < target) l++;
  else r--;
}
return {-1, -1};
}
```

time complexity: O(nlogn) + O(n)
space complexity: O(1)

#75. Sort Colors:

Given an array nums with n objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.
We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.
You must solve this problem without using the library's sort function.

From <https://leetcode.com/problems/sort-colors/>

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

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

// Brute force : (Using Sort)

- sort the given array using sorting technique.

// Better Solution : (counting)

- iterate over the array, count the no. of 0s, 1s & 2s in the array.
- again using the loops override 0s, 1s & 2s in the array according to their count.

```
cnt0 = cnt1 = cnt2 = 0;
for(i=0 ⟶ n){
  if(a[i]==0) cnt0++;
  else if(a[i]==1) cnt1++;
  else cnt2++;
}
for(i=0 ⟶ cnt0) a[i] = 0;
for(i=cnt0 ⟶ cnt0+cnt1) a[i]=1;
for(i=cnt0+cnt1 ⟶ n) a[i]=2;
```

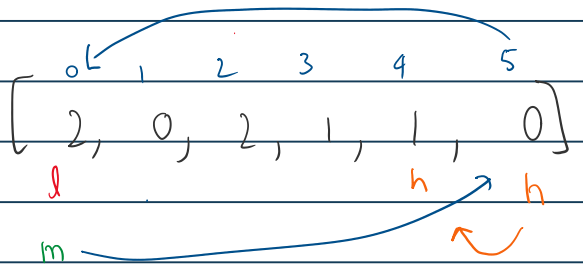
n + n

time complexity : O(2n)
space complexity : O(1)

// Optimal Solution: (Using 3 pointers)

- we will be using three pointer, low, mid & high

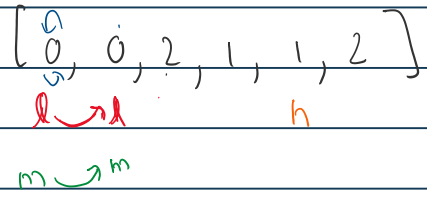
[0 ... low-1] - contains only 0
[low ... mid-1] - contains only 1
[high+1 ... n-1] - contains only 2



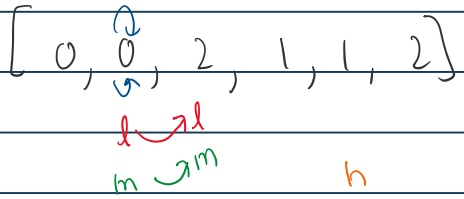
we will move mid pointer

```
if(a[mid]==0){
    swap(a[l], a[mid])
    l++, mid++;
}
else if(a[mid]==1) mid++;
else{
    swap(a[mid], a[h]);
    h--;
}
```

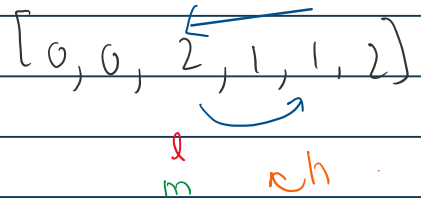
mid=0, a[mid]==2, swap(a[mid], a[h])
h--;



mid=0, a[mid]=0, swap(a[mid], a[low])
l++, mid++



mid=1, a[mid]==0, swap(a[mid], a[l])
l++, mid++



m=2 a[mid]==2, swap(a[mid], a[h])
h--;

[0, 0, 1, 1, 2, 2]
l h
m

mid = 2, a[mid] == 1, mid++

[0, 0, 1, 1, 2, 2]
l m
h

mid = 3, a[mid] == 1, mid++

[0, 0, 1, 1, 2, 2]
m
l h

m > h, EXIT Loop, array is sorted

```
int l = m = 0, h = n-1;
while (m <= h){
  if (a[m] == 0){
      swap(a[m], a[l]);
        l++, m++;
    }
  else if (a[m] == 1) m++;
  else {
      swap(a[m], a[h]);
        h--;
  }
}
```

time complexity: O(n)

space complexity: O(1)

#169. Majority Element: n/2 times

Given an array nums of size n, return *the majority element*.
The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

From <https://leetcode.com/problems/majority-element/>

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

// Brute force : (two loops)

- use outer loop to select the element, the inner loop to count the occurrence of that element.
- check if cnt > $\frac{n}{2}$, if yes return the element.

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

time complexity: $O(n^2)$
space complexity: $O(1)$

// Better Solution : (Hashing)

- initializing a map<int, int>, iterate over the array and store the occurrence of each element is the map as (element, count) pair.
- Iterate over the the map, if the count of any element > $\frac{n}{2}$, return that element;

```
map<int, int> mp;
for(i=0 ⟶ n){
    mp[a[i]]++;
}
for(auto it : mp){
    if ( it.second > n/2)
        return it.first;
}
return -1;
```

time complexity: O(n)

space complexity: O(n)

// Optimal Solution: (Moore's Voting Algorithm)

- if an element occurs more than $\frac{n}{2}$ times, it implies all other element occur less than $\frac{n}{2}$ times.

- we will initialize two variables:
  result : our majority element
  cnt : cnt of the result.

- Iterate over the array, and for each iteration consider the following case:

1. if (cnt == 0) : set our element as majority element
   result = a[i];

2. if(a[i] == result) : increment the count of result

3. if (a[i] != result) : decrement the count of result.

- Outside the loop, cross check the result by counting its occurrence using a single loop.
  if (result occurs more than $\frac{n}{2}$ times) return result;

	0	1	2	3	4	5	6
cnt=0	2	2	1	1	1	2	2

i=0, cnt==0, yes, result = a[i] = 2, cnt=1
i=1, a[i]==result, yes, cnt++ = 2
i=2, a[i]!=result, cnt-- = 1
i=3, a[i]!=result, cnt-- = 0
i=4, cnt==0, result = a[i] = 1, cnt++=1
i=5 a[i]!=result, cnt-- = 0
i=6, cnt==0, result = a[i] = 2 ANS

```
int cnt=0, result;
for(i=0 ⟶ n){
    if (cnt==0){
        result= nums[i];
        cnt++;
    }
else if(nums[i]=result) cnt++;
else    cnt--;
}
```

```
new_cnt=0
for(i=0 ⟶ n){
    if(a[i]==result) new_cnt++;
}
if(new_cnt > n/2) return result;
else return -1;
}
```

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

space complexity: O(1)

#53. Maximum Subarray:

- Given an integer array nums find the subarray with the largest sum, and return the sum.

[-2, 1, -3, 4, -1, 2, 1, -5, 4] // 6

[1] // 1 [5, 4, -1, 7, 8] // 23

// Brute force (Using three loops)

- use three loops, loop1 to select the starting point loop2 to set the endpoint of subarray, loop3 to find the sum of curr subarray.
- check if sum > maxSum, update accordingly.

```
maxSum = INT_MIN;
for(i=0 ⟶ n) {
    for(j=i ⟶ n) {
        int sum = 0;
        for(k=i ⟶ j)
            sum += arr[k];
        maxSum = max(maxSum, sum);
    }
}
return maxSum;
```

time complexity : $O(n^3)$
space complexity : $O(1)$

// Better Solution: (Using two loops)

instead of using the loop to find the sum, we will keep calculating the sum of the curr subarray in the second loop.

- outside the second loop, update the maxSum accordingly.

```
maxSum = INT_MIN;
for(i=0 ⟶ n){
    sum = 0;
    for(j=i ⟶ n){
        sum += arr[j];
    }
    maxSum = max(maxSum, sum);
}
return maxSum;
```

time complexity: $O(n^2)$
space complexity: $O(1)$

Optimal Solution: (Kadane's Algorithm)

- Intuition is not to consider the subarray as a part of the answer.
- A subarray with a sum less than 0 will always reduce the answer so if the sum < 0, don't include the subarray in the ans, by making sum = 0.
- iterate through the array, keep calculating the sum by adding each element
  - if currSum > maxSum : maxSum = currSum;
  - if currSum < 0 : make currSum = 0

```
sum = 0   maxSum = 0;
for(i=0 ⟶ n){
    sum += nums[i];
    maxSum = max(maxSum, sum);
    if(sum < 0) sum = 0;
}
return maxSum;
```

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

Return Subarray with max Sum:

- We will some changes to Kadane's Algo to keep a track of the subarray with max Sum.

start : start point of our subarray
ansStart : start point of maxSubArray
end point of max SubArray

```
int start=0, ansStart=0, ansEnd=0,
int sum=0, maxi = INT_MIN;
for(i=0 ⟶ n){
  if (sum==0) start=i;
  sum += a(i);
  if(sum > maxi){
    ansStart = start
    ansEnd = i;
  }
  if ( sum < 0) sum = 0;

return {a.begin() + ansStart, a.begin() + ansEnd}
```

#121. Best time to Buy & Sell Stock

You are given an array prices where prices[i] is the price of a given stock on the ith day.
You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.
Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

From <https://leetcode.com/problems/best-time-to-buy-and-sell-stock/>

[7, 1, 5, 3, 6, 4] // 5 (6 - 1) = 5

buy sell

[7, 6, 4, 3, 1] // 0 no profit possible

// Brute force : (Using two loops)

- use two loops to track every transaction, and maintain the maxProfit
- outer loop to select the day to buy, inner loop to select the day you want to sell.
- update the currProfit & maxProfit.

```
int maxProfit = 0;
for(i=0 ⟶ n){
    for(j=i+1 ⟶ n){
        if(a[j] > a[i]){
            currProfit = a[j]-a[i];
            maxProfit = max(maxProfit, currProfit);
        }
    }
}
return maxProfit;
```

time complexity: $O(n^2)$
space complexity: $O(1)$

Time Limit Exceed : on leet code, on code studio
$n <= 10^5$

// Optimal Solution: (Using single loop)

- instead of using two loops, we will try to compute currProfit in a single loop, and update maxProfit if currProfit > maxProfit.

	0	1	2	3	4	5
prices[] =	7	1	5	3	6	4

mini = INT_MAX

for each iteration,
mini = min(mini, prices[i])
profit = a[i] - mini
maxProfit = max(maxProfit, profit)

i=0, mini = 7, profit = 7-7 = 0,
i=1, mini = 1, profit = 0,
i=2 mini = 1, profit = 5-1 = 4
i=3, mini = 1, profit = 3-1 = 2
i=4, mini = 1, profit = 6-1 = 5
i=5, mini = 1, profit = 4-1 = 3

maxProfit = ~~0~~ ~~4~~ 5

⇓

5

```
minPrices = INT_MAX, max_Profit = 0;
for(i=0 ⟶ n){
 minPrice = min(minPrice, prices[i]);
 int currProfit = a[i] - minPrice;
 maxProfit = max(maxProfit, currProfit);
}
return maxProfit
```

time complexity: O(n)
space complexity: O(1)

#2149. Rearrange Array Elements by Sign :

You are given a **0-indexed** integer array nums of **even** length consisting of an **equal** number of positive and negative integers.
You should **rearrange** the elements of nums such that the modified array follows the given conditions:

1. Every **consecutive pair** of integers have **opposite signs**.
2. For all integers with the same sign, the **order** in which they were present in nums is **preserved**.
3. The rearranged array begins with a positive integer.

Return *the modified array after rearranging the elements to satisfy the aforementioned conditions.*

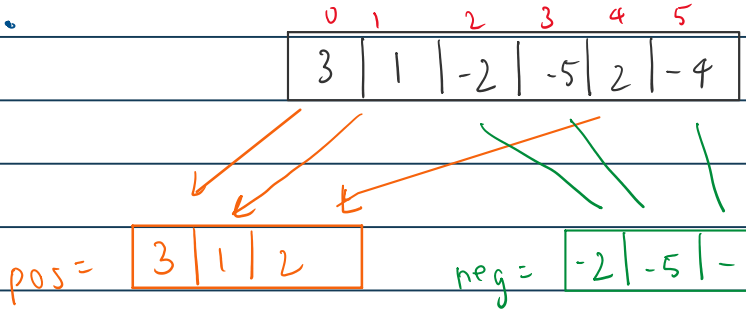
From <https://leetcode.com/problems/rearrange-array-elements-by-sign/>

[3, 1, -2, -5, 2, -4] // [3, -2, 1, -5, 2, -4]

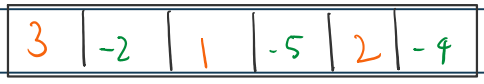
[1, -1] // [1, -1]

// Brute force : (using two extra vectors)

- create two empty vectors of size $\frac{n}{2}$ each to store positive & negative elements.



iterate over the array and insert +ve elements in pos[] & -ve elements in neg[].



- insert +ve elements at even index & -ve elements at odd index.

```
pos(n/2), neg[n/2];
for(i=0 ⟶ n){
    if(a[i] > 0) pos.push_back(a[i]);
    else( neg.push_back(a[i]);
}

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

Note: If no. of +ves & -ves are not equal, this approach optimal solution

time complexity : O(n)
space complexity : O(n)

```
        a[2*i+1] = neg[i];
    }
    return a;
```

Space complexity : O(n)

// Optimal Solution :

- instead of doing two passes, we will try to solve in a iteration.
- create an empty array ans[], iterate through the array and simultaneously insert elements in the ans array:

```
vector<int> ans;
negIndex = 1, posIndex = 0;
for(i=0 ⟶ n){
  if(nums[i] < 0){
        ans[negIndex] = nums[i];
    negIndex += 2;
  }
  else{  ans[posIndex] = nums[i];
        posIndex += 2;
  }
return ans;
```

time complexity : O(n)
space complexity : O(n)

#31. Next Permutation :

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of arr = [1,2,3] is [1,3,2].
- Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
- While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.

The replacement must be **in place** and use only constant extra memory.

From <https://leetcode.com/problems/next-permutation/>

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

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

[1,1,5] // [1,5,1]

// Brute force :

- find all the permutations of the given array and store them.
- Search given array among all permutations
- Print the next permutation present right after the given array.

// Optimal Solution : (In-built function)

- C++ provides a built-in function which provides the next permutation just after the given array.

```
next_permutation (nums.begin(), nums.end())
```

- included in the <algorithm> header file.

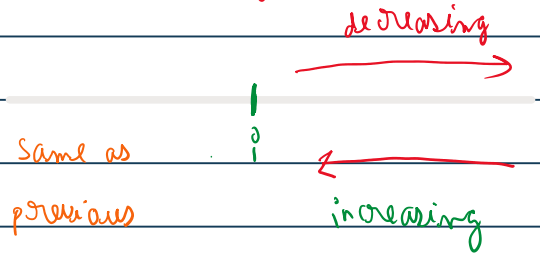
time complexity : O(n)

space complexity : O(1)

// Optimal Solution - II

- observe the following:

- 1. Take idea from the dictionary, "xaj, xax, xbx" these strings have a common prefix, ranking are done on the basis of differentiating character.
  - likewise: (1,2,3) } same prefix
    (1,3,2)
    (2,1,3)
    (2,3,1)
  - consider index 'i' being the breaking point (left part being same (right part decreasing)



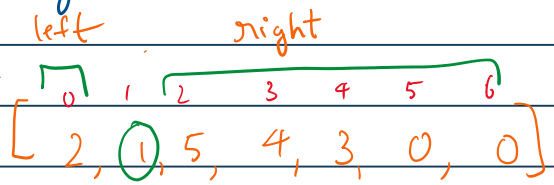
- find the breaking point by traversing from behind, break where $a[i] < a[i+1]$.

2. if there is no breaking point, it means the current array is the last permutation, we will return the smallest one as the next permutation,
   return array as reversed

3. to get the next permutation, find the next greater of a(break-point) from the right half, and swap it with current break-point.
   traverse from back side, the first element greater than the break-point, swap it with current break-point.

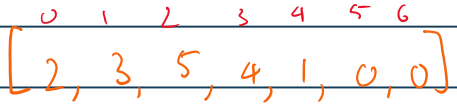
4. After swapping the break-point, our permutation > current permutation, in-order to return the next permutation, find the smallest permutation of the right half by reversing it.

// Dry Run:

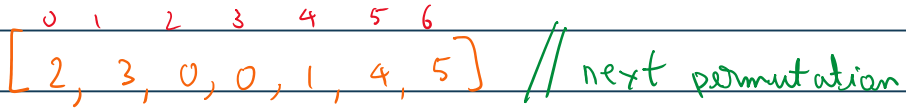


1. break-point will be index 1 = i=1
2. swap the break-point with next greater than break-point in right half

swap a[i] with a[4]



- reverse the right half of the break-point




```
ind = -1;
for(i=n-2; i>=0; i++){
    if(a[i] > a[i+1]){
        ind = i;
        break;
    }
}
if(ind == -1){
 reverse(a.begin(), a.end());
   return a;
}
for(i=n-1; i>ind; i++){
    if(a[i] > a[ind])
      swap(a[i], a[ind]);
        break;
} }
```

```
    reverse( a.begin()+ind+1, a.end());

    return a;
}
```

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

space complexity: $O(1)$

Leaders in Array:

Given an array, print all the elements which are leaders. A Leader is an element that is greater than all of the elements on its right side in the array.

From <https://takeuforward.org/data-structure/leaders-in-an-array/>

[4, 7, 1, 0] // [7, 1, 0]

[10, 22, 12, 3, 0, 6] // [22, 12, 6]

// Brute force Solution

- We will check for each elements whether it is a leader or not using two loops
- Outer loop to select the element, inner loop to check whether it is greater than all elements on its right half side.
- If a element is a leader push it in the ans, if not move to next element.

```
vector<int> ans;
for(i=0 ⟶ n){
    flag = true;
    for(j=i+1; ⟶ n){
        if(a[j] > a[i]){
            flag = false;
            break;
        }
    }
    if(flag) ans.push(a[i]);
}
return ans;
```

time complexity: $O(n^2)$
space complexity: $O(1)$

// Optimal Solution :

- we will use only one loop to iterate a single time.
- start iterating from behind, make the element as maxi, push the last element in ans.
- start comparing the elements, if an element is greater than maxi, it is one of the leader, push it in ans and update the maxi.
- return the ans.

```
leader = a[n-1];
ans.push(leader);
for(i=n-2; i>=0; i--){
    if(a[i] > leader){
        leader = a[i];
        ans.push(leader);
    }
}
return ans;
}
```

time complexity: O(n)

space complexity: O(1)

#128. Longest Consecutive Sequence:

Given an unsorted array of integers nums, return *the length of the longest consecutive elements sequence.*

From <https://leetcode.com/problems/longest-consecutive-sequence/description/>

[100, 4, 200, 1, 3, 2] // 4

[0, 3, 7, 2, 5, 8, 4, 6, 0, 1] // 9

// Brute force:

- Iterate through each element, and for each element x = a[i], do a linear search for, x+1, x+2, .. consecutive numbers.
- If next consecutive increase the curr_len and keep updating the max of curr_len and max_len.

```
ans = INT_MIN;
for(i = 0 ⟶ n){
    x = a[i]
    curr_len = 1;
    while( LinearSearch (a, x+1)){
        x += 1;
        curr_len += 1;
    }
    ans = max(ans, curr_len);
}
return ans;
}
```

time complexity: $O(n^2)$

space complexity: $O(1)$

Optimal Solution - I (using sorting)

- Sort the array to bring the consecutive integers adjacent to each other.
- initialize a variable lastSmaller to keep track of the last integer of the current sequence.
- if current element is just next in the sequence, increase the len of current sequence, and make the current as the lastSmaller.
- if not make len=1, & lastSmaller = current element
- Keep storing the max len of sequence.

```
sort(nums);
lastSmall = INT_MIN;
cnt=0, longest=1;
for(i=0 ——→ n){
   if(a[i]-1 == lastSmaller){
      cnt+=1;
      lastSmall = nums[i];
   }
else if(nums[i] != lastSmaller)
      cnt=1;
      lastSmaller = a[i];
   }
   longest = max(longest, cnt);
}
return longest;
```

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

space complexity: O(1)

// Optimal Solution - II (using set)

- put all the elements in the unordered set.
- for any element x to be an starting point of a sequence, (x-1) should not be present in set.
- if 'x' is a starting point, cnt = 1, and find the consecutive numbers (x+1, x+2 ...) and keep increasing the cnt if consecutive no. are found,
- update the max_len accordingly.

```
unordered_set<int> st;
longest = 1;
if(n==0) return 0;
for( i=0 ⟶ n)
    st.insert (a[i]);
for(auto it : st){
    if ( st.find (it - 1) == st.end()) {
        cnt = 1;
        x = it;
        while ( st.find(x+1) != st.end() ){
            x += 1;
            cnt += 1;
        }
    }
    longest = max(longest, max);
}
return longest;
}
```

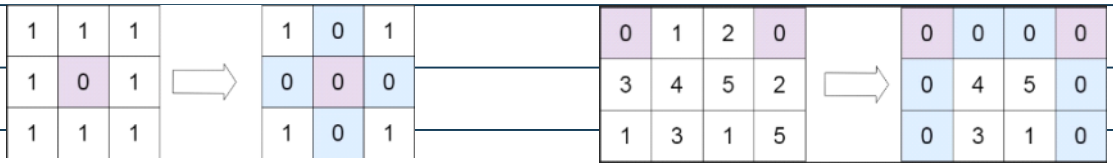
time complexity : O(n)

space complexity : O(n)

#73. Set Matrix Zeroes

Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's.
You must do it in place.

From <https://leetcode.com/problems/set-matrix-zeroes/description/>



// Brute force :

- iterate over the matrix, if you encounter an zero, mark the whole row and whole col.
- How to : markRow : take the ith row and traverse the mark cols and mark that

  markCol : take the jth col and traverse the row & mark that

- Again traverse the matrix, if you encounter an marked element make it zero.

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

```
markRow(i){
  for(j=0 ⟶ n){
    if(a[i][j] != 0)
      a[i][j] = flag
  }
}
markCol(j){
  for(i=0 ⟶ m){
    if(a[i][j] != 0)
      a[i][j] = flag
  }
}
```

```
for(i=0 ⟶ m){
  for(j=0 ⟶ n){
    if(a[i][j] == flag)
      a[i][j] = 0;
  }
}
```

time complexity:
$O(m*n*(m+n)) + O(m*n)$

Space complexity:
$O(1)$

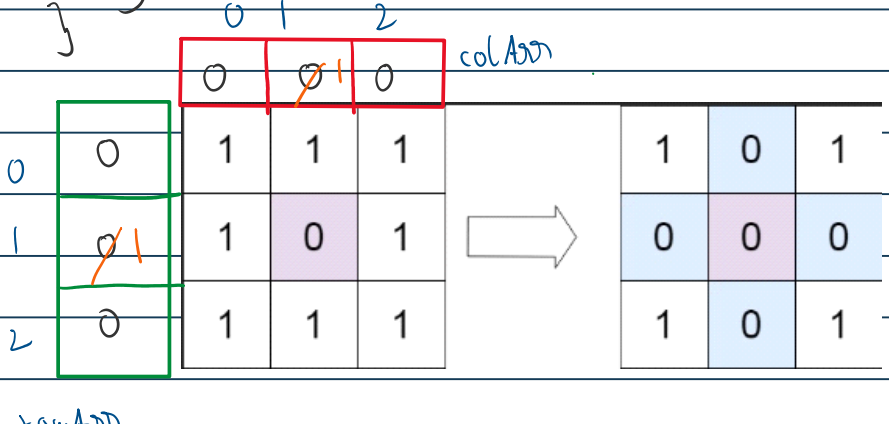
// Better Solution : (using extra space)

- initializing two arrays : rowArr & colArr, iterate over the matrix and as you find a zero, i.e a[i][j]==0
  - mark ith index of rowArr
  - mark jth index of colArr.
- Iterate over the matrix again, for each cell a[i][j], check if (rowArr[i] == flag || colArr[j] == flag) mark that cell as zero

```
rowArr(m,0), colArr(n,0);
for(i=0 ⟶ m){
   for(j=0 ⟶ n){
      if(a[i][j]==0){
         rowArr[i] = 1
         colArr[j] = 1
      }
   }
}
```

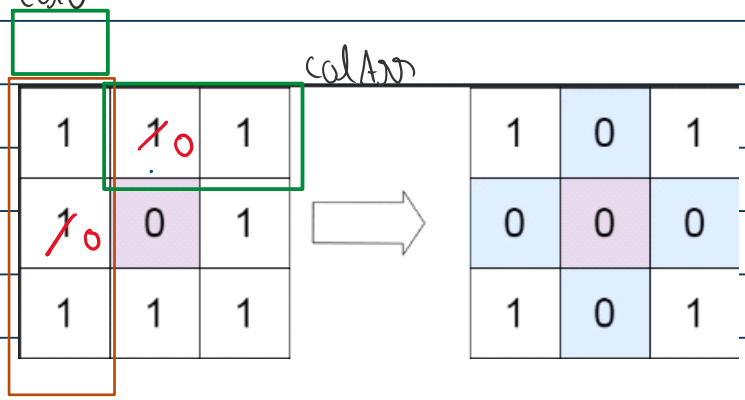
time complexity: O(m*n) + O(m*n)
space complexity: O(m+n)

```
for(i=0 ⟶ m){
   for(j=0 ⟶ n){
      if(rowArr[i]==1 || colArr[j]==1)
         a[i][j] = 0;
   }
}
```



Optimal Solution: (no extra space)

- We will take the idea from the previous solution, try to avoid the extra rowArr and colArr.
- Make col[0] as rowArr and row[0] as colArr. , but a[0][0] will be overlapping so, we will include a[0][0] in the rowArr, and initializing a variable col0 to store the marked value.
- col0



- now traverse the matrix, if you encounter any zero mark the rowArr and colArr accordingly, but if j == 0 for any zero cell make col0 = 0 not the colArr.
- start from matrix[1,1] upto matrix[m-1, n-1] and start making the cells zero with rowArr & colArr have zeroes for ith & jth indexes.
- check for 0th row by checking matrix[0][0] == 0, and for 0th col by checking col0 == 0;
- Step1: - mark the rowArr and colArr (a[i][j] == 0)
  { a[i][0] = 0 (if j != 0 a[0][j] = 0 else col0 = 0)}
  - start from [1][1] → [m-1][n-1] to make cells zero by checking (a[i][0] == 0 || a[0][j] == 0)
  - for 0th row (a[0][0] == 0) & for 0th col (col0 == 0)

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

time complexity: $O(m*n) + O(m*n) + O(m) + O(n)$

space complexity: $O(1)$

48. Rotate Image

You are given an n x n 2D matrix representing an image, rotate the image by **90** degrees (clockwise).
You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

From <https://leetcode.com/problems/rotate-image/description/>

1	2	3
4	5	6
7	8	9

⇒

7	4	1
8	5	2
9	6	3

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

⇒

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

// Brute force (using extra space)

- make an empty matrix of size n×n, and try try to place elements of given matrix into empty matrix at their rotated position.

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

[i][j] → [j][(n-1)-i]

```
mat[n][n];
for(i=0 ⟶ n){
  for(j=0 ⟶ n){
    mat[j][(n-1)-i] = matrix[i][j];
  }
}
```

time complexity: $O(n^2)$
space complexity: $O(n^2)$

// Better Solution: (in place)

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

⇒

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

- observe that each col of the matrix is converted to row in reverse order in the rotated matrix.

- This means, we can first transpose the matrix and then reverse each row.

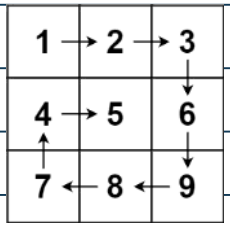
```
for(i=0 ⟶ n){
    for(j=0 ⟶ m){
        swap(a[i][j], a[j][i])
    }
}

for(i=0 ⟶ n){
    reverse(a[i].begin(), a[i].end());
}
```

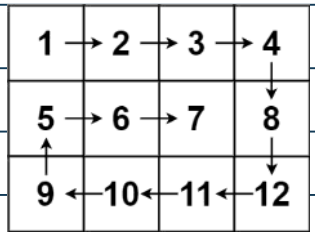
time complexity : $O(n^2)$
space complexity : $O(1)$

54. Spiral Matrix

Given an m x n matrix, return all elements of the matrix in spiral order.

From <https://leetcode.com/problems/spiral-matrix/description/>





// [[1,2,3],[4,5,6],[7,8,9]]

[1,2,3,6,9,8,7,4,5]

[[1,2,3,4],[5,6,7,8],[9,10,11,12]]

// [1,2,3,4,8,9,11,10,9,5,6,7]

- if we observe carefully: the traversal goes like

  right → bottom → left → top
  first → last → last → first col
  row col row

- we will have four variables storing the starting & ending points of our movements, a total of m x n movements will be there

  startingRow = 0
  startingCol = 0
  endingRow = m - 1
  endingCol = n - 1

- traverse from startingCol → endingCol, cnt++
  increment startingRow

- traverse from startingRow → endingRow
  decrement endingCol, cnt++

cnt < total (m x n)

- traverse from : endingCol ⟶ startingCol, cnt ++
  decrement endingRow
- traverse from : endingRow ⟶ startingRow, cnt ++
  increment startingCol

} cnt < total

- keep on doing these operations until cnt < total.

```
vector<> ans;  sR = sC = 0, eR = m-1, eC = n-1, cnt = 0, total = m x n;
while ( cnt < total) {
  for(i = sC ⟶ eC) ans.push(mat[sR][i]) cnt++;
   sR++;

for(i = sR ⟶ eR) ans.push(mat[i][eC]) cnt++
  eC--;

for(i = eC ⟶ sC) ans.push[mat[eR][i] cnt++;
 eR--;

for(i = eR ⟶ sR) ans.push[mat[i][sC] cnt++
 sC++;

  }
return ans;

}
```

time complexity: O(m x n)

space complexity: O(1)

#560. Subarray Sum Equals K :

Given an array of integers nums and an integer k, return the total number of subarrays whose sum equals to k.

From <https://leetcode.com/problems/subarray-sum-equals-k/description/>

[1,1,1] , k=2 // 2 [1,1], [1,1]

[1,2,3] , k=3 // 2 [1,2], [3]

// Brute force: (using three loops)

- initializing a variable cnt = 0;
- find all the possible subarrays of the given array, find the sum of each subarray, if it is == k, increment the cnt variable;
- Run a loop i from 0 ⟶ n, indicating the starting point of a subarray,
- run another loop(j) inside loop(i), j = i ⟶ n, indicating the end point of a subarray,
- run another loop(l) l = i ⟶ j, to calculate the sum, if sum == k, cnt++
- Outside the loop(i) return cnt

```
cnt = 0,
for(i=0 ⟶ n){
    for(j=i ⟶ n){
        sum = 0;
        for(l=i ⟶ j){
            sum += a[l];
        }
        if(sum == k) cnt++;
    }
}
return cnt;
```

time complexity: ≈O(n³)

space complexity: O(1)

// Better Solution : (using two loops)

- use the same idea of prev solution, but instead of using a third loop, calculate the sum in the second loop itself and compare it with k, if sum == k, cnt++.

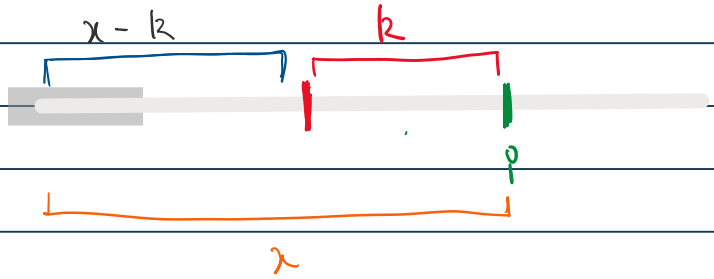
```
cnt = 0;
for(i=0 ⟶ n){
    sum = 0;
    for(j=i ⟶ n){
        sum += a[j];
        if(sum == k)
            cnt++;
    }
}
return cnt;
```

time complexity: $O(n^2)$

space complexity: $O(1)$

// Optimal Solution : (using prefix sum)

- use the concept of prefix sum, assume the following.



- we will look for subarray with sum k upto index i, but instead of that we will look for sum = x - k

1. initialize a map<int, int> mp, to store (prefixSum, Occurrence)
2. set the value of 0 as 1.
3. iterate over the array

3. iterate over the array
4. for each element i

- calculate prefixSum (x)
- find remove : (x - k)
- add occurrence of (x-k) to cnt
  update occurrence of (x-k) in map

[1, 2, 3]

sum = 1
remove = 1 - 3 = -2
cnt = mp[-2] = 0
mp[1]++

} i = 0

(6,1)
(3,1)
(1,1)
(0,1)

sum = 3
remove 3-3 = 0
cnt = mp(0) = 1
mp(3)++

} i = 1

sum = 3+3 = 6
remove: 6-3 = 3
cnt += mp(3) = 1+1 = 2
mp(6)++;

map<int, int>, cnt = 0, mp(0)++; preSum = 0;
for (i = 0 ⟶ n) {
  preSum += a(i);
  cnt += mp[preSum - k]
  mp(preSum)++;
}

return cnt;

time complexity : O(n)
space complexity : O(n)