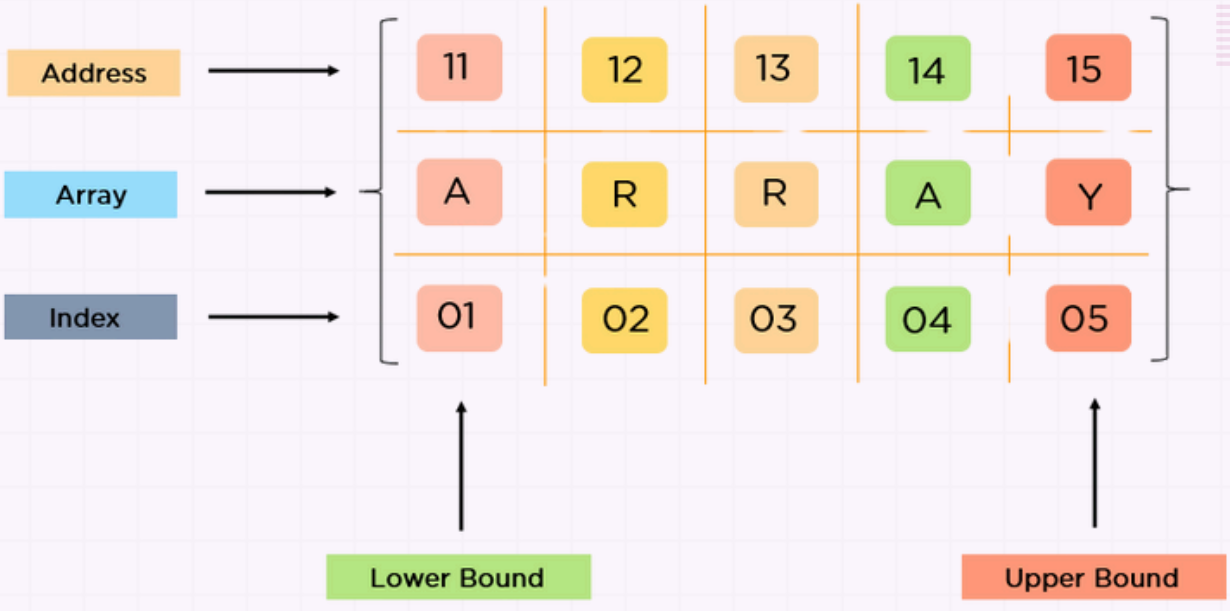
InterviewCafe
Where Learning Never Takes a Break!

Top Array Interview Questions & Explanations



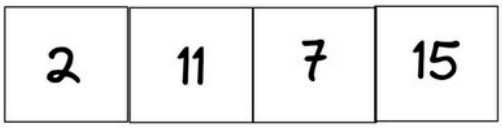
Santosh Kumar Mishra
Software Engineer at Microsoft • Author • Founder of InterviewCafe

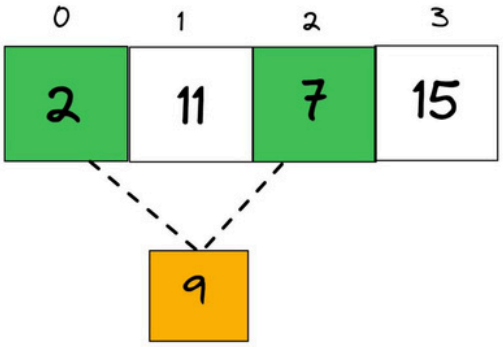
Table of Contents

Two Sum

Easy

→ Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.



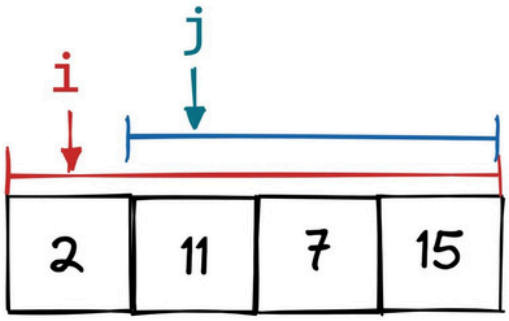


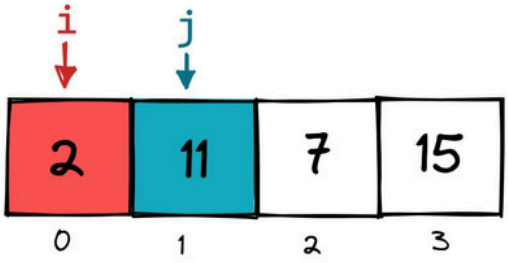
Target = 9

Output - [0,2]

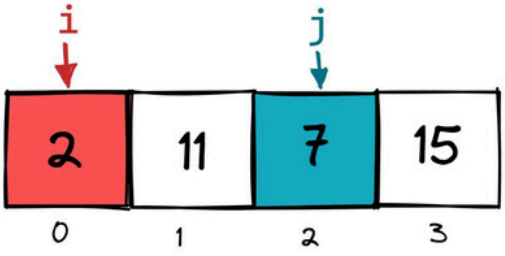
Approach-1

→ The brute force to solve this problem is by nested traversal. Loop through elements and check if arr[i] + arr[j] == target.





arr[i] + arr[j] != target ✗

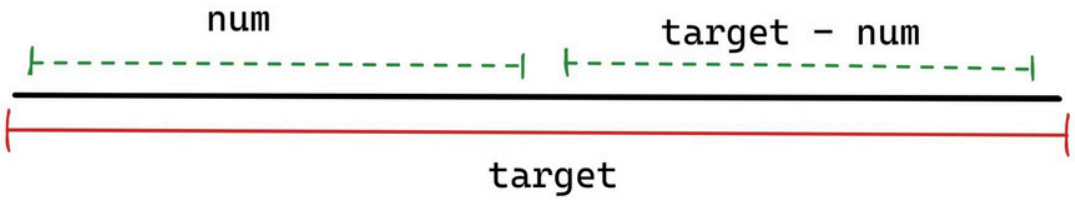


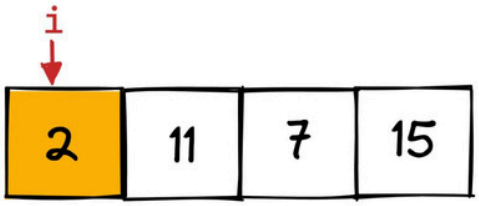
arr[i] + arr[j] == target ✓

in Santosh Kumar Mishra

Approach-2 (using HashMap)

→ Using HashMap, How we are gonna use hashmap Let's take a example :-
we have given target and num so, check if target-num is present in our hashmap or not.
if target-num is present is our hashmap we got those two element. otherwise not.



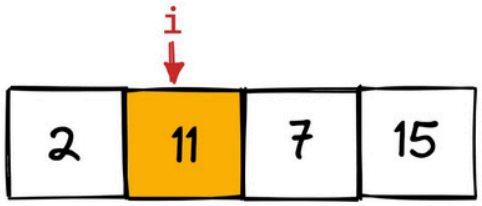


num , index

2 : 0

HashMap

target-num = 9-2 = 7 ✗

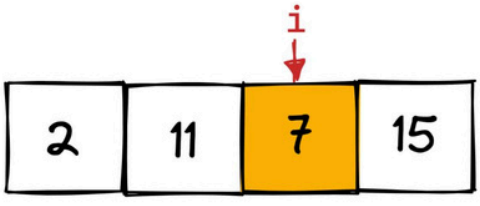


num , index

2 : 0
11 : 1

HashMap

target-num = 9-11 = -2 ✗



num , index

2 : 0
11 : 1

HashMap

target-num = 9-7 = 2 ✓

```
temp[0] = i
temp[1] = map.get(target-num)
```

target = num + x
x = target - num
x = 9 - 7 , x = 2

Approach-1

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {

        int[] arr = new int[2];
        for(int i  = 0;i<nums.length-1;i++){
            for(int j = i+1;j<nums.length;j++){

                if(nums[i] + nums[j] == target){
                    arr[0] = i;
                    arr[1] = j;

                    return arr;
                }
            }
        }
        return arr;

    }
}
```

T.C - O(n^2)
S.C - O(1)

Approach-2

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {

        int[] arr = {0,0};
        HashMap<Integer,Integer> map = new HashMap<>();

        for(int i = 0;i<nums.length;i++){

            if(map.containsKey(target-nums[i])){
                arr[0] = i;
                arr[1] = map.get(target-nums[i]);
                return arr;
            }
            else{
                map.put(nums[i],i);
            }
        }
        return arr;
    }
}
```

T.C - O(n)
S.C - O(n)

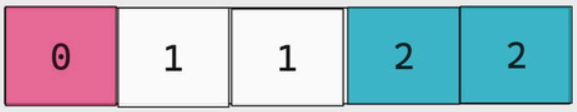
Santosh Kumar Mishra

Sort an array of 0s, 1s and 2s

Medium

→ Given an array of size N containing only 0s, 1s, and 2s. sort the array in ascending order.

Input : nums = [2,0,2,1,1]



Output : [0,1,1,2,2]

Brute-Force Approach : → Sort the given array

T.C ——— O(nlogn)

S.C ——— O(1)

Better Approach : → Count number of 0's,1's & 2's and insert into the array.

T.C ——— O(n)

S.C ——— O(1)

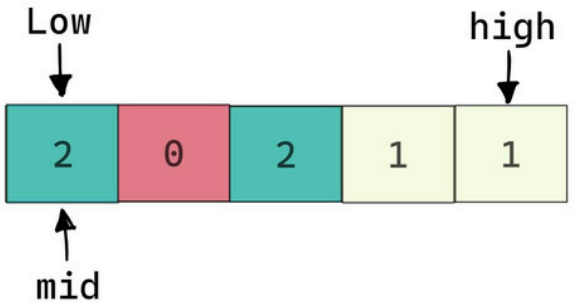
Optimal Approach :

→ Using "Dutch National flag algorithm". In-place Algorithm.

In this approach, we will be using 3 pointers named low, mid, and high.

The primary goal here is to move 0s to the left and 2s to the right of the array and at the same time all the 1s shall be in the middle region of the array and hence the array will be sorted.

in Santosh Kumar Mishra

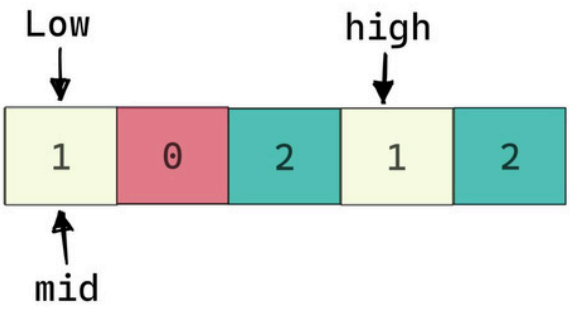


Rules:→ if arr[mid]

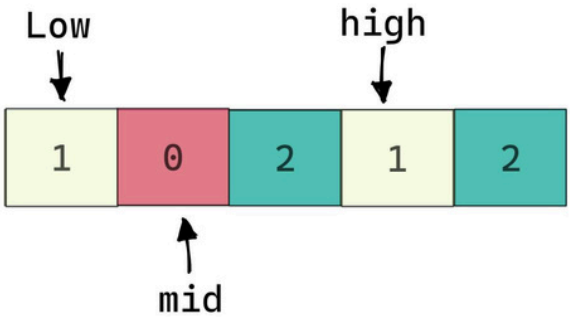
0 → swap(arr[low],arr[mid])
low++ , mid++

1 → mid++

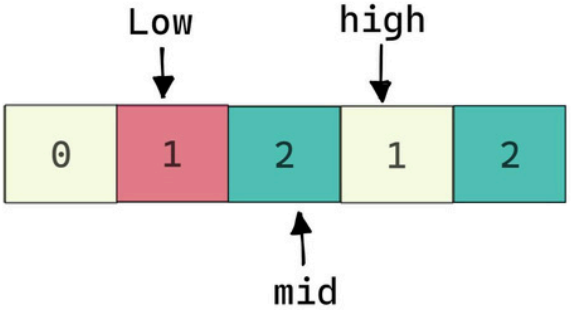
2 → swap(arr[mid],arr[high])
high--



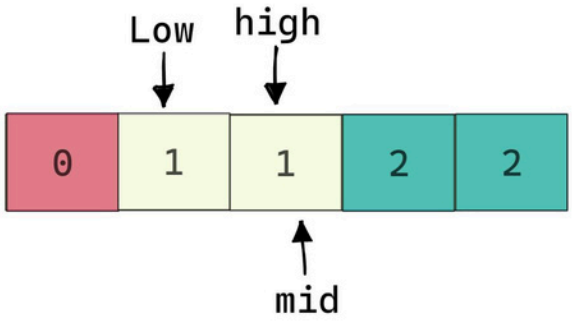
arr[mid] = 2
swap(arr[mid],arr[high])
high--



arr[mid] = 1
mid++



arr[mid] = 0
swap(arr[low],arr[mid])
low++ , mid++



arr[mid] = 2

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

high--

Code:-

```java
class Solution {
    public void sortColors(int[] nums) {
        int low = 0;
        int mid = 0;
        int high = nums.length-1;
        int temp = 0;

        while(mid<=high){

            if(nums[mid] == 0){
                temp = nums[mid];
                nums[mid] = nums[low];
                nums[low] = temp;
                mid++;
                low++;
            }
            else if(nums[mid] == 1){
                mid++;
            }
            else{
                temp = nums[mid];
                nums[mid] = nums[high];
                nums[high] = temp;
                high--;
            }
        }
    }
}
```

Time Complexity → O(n)

Space Complexity → O(1)

Optimal Approach

Santosh Kumar Mishra

Peak element

Medium

→ A peak element is an element that is strictly greater than its neighbors.

→ If the array contains multiple peaks, return the index to any of the peaks.

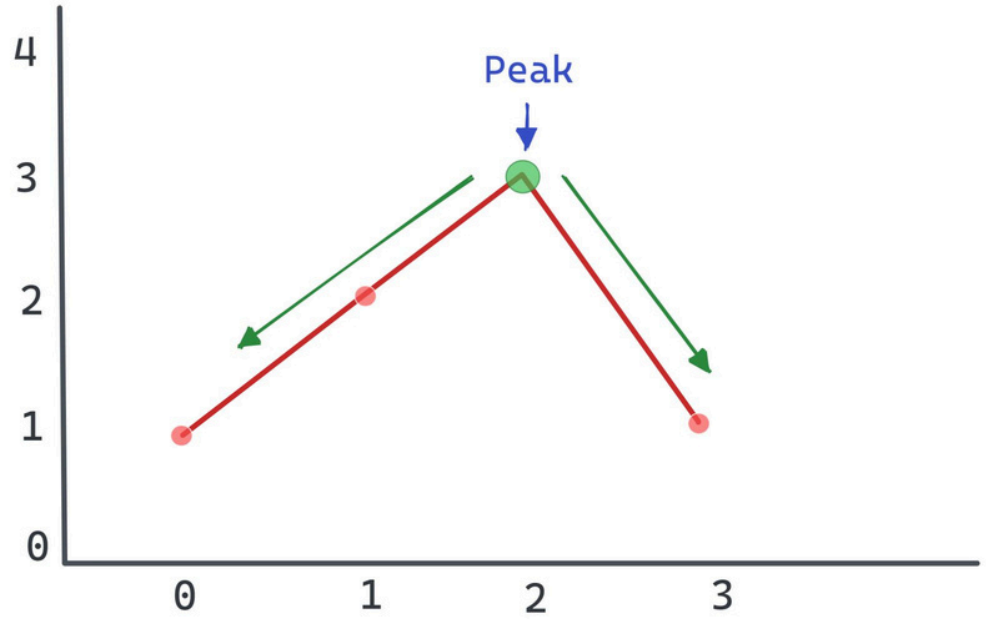
```
Input: nums = [1,2,3,1]

Output: 2

Explanation: 3 is a peak element and your
function should return the index number 2.
```

Approach 1 (Linear Search) :

→ We can just look at each and every element and compare the neighbors.

→ Iterate over the array and check if that element is greater than it's neighbouring elements. Then its a peak element.

$$arr[i] \geq arr[i-1] \;\&\&\; arr[i] \geq arr[i+1]$$



Santosh Kumar Mishra

Brute Approach

Time Complexity → O(n)

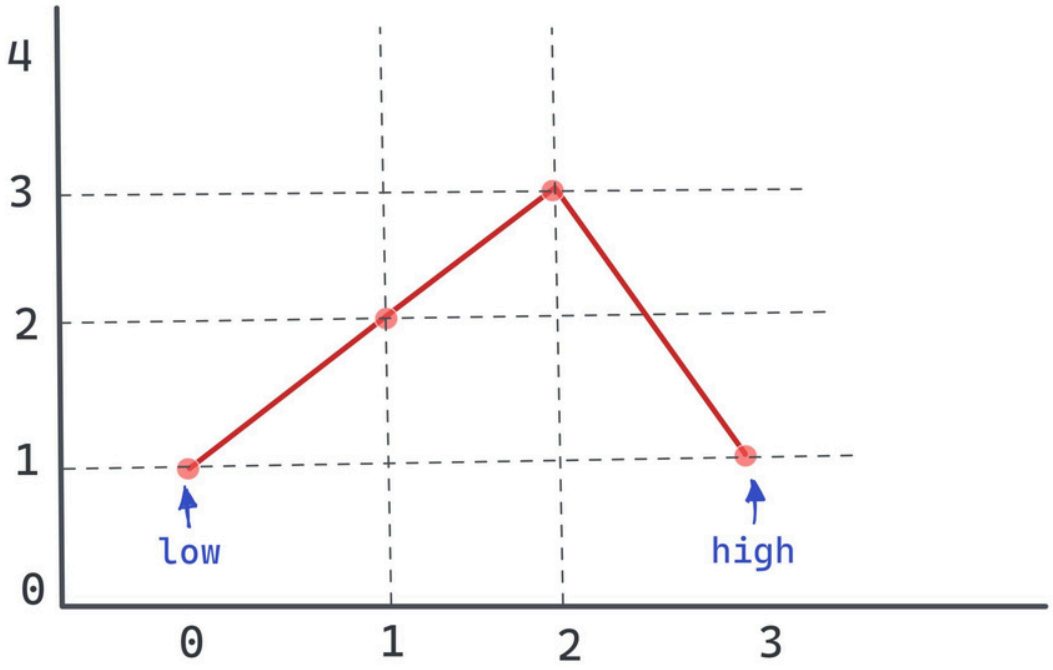
Space Complexity → O(1)

Approach 2 (Binary Search)

→ We are going to maintain two pointers low & high ,set low at 0 and high at nums.length-1.

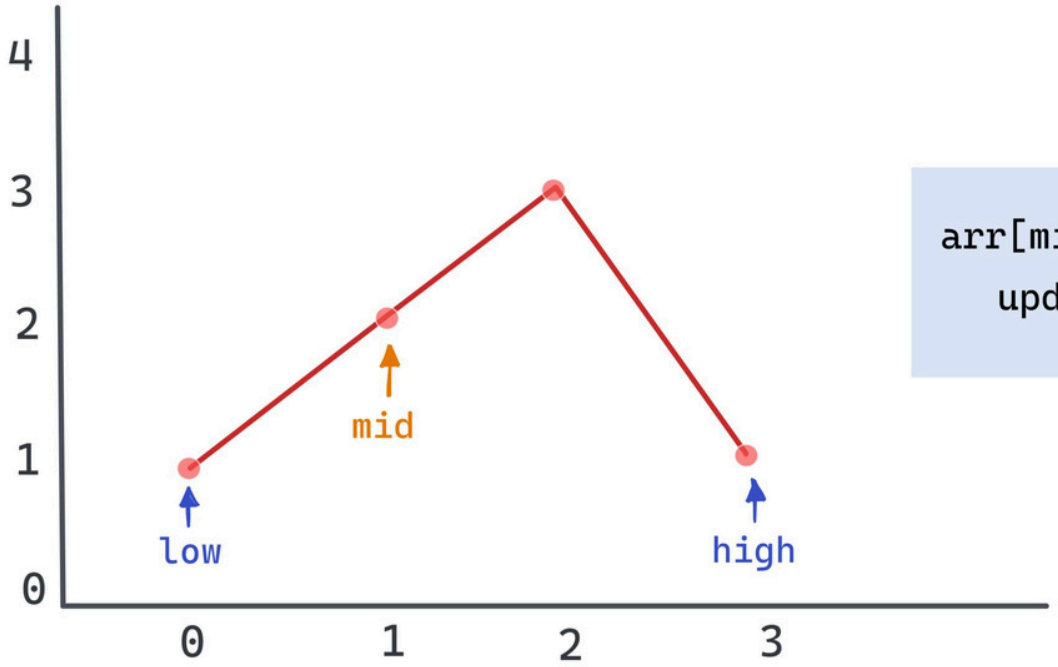
→ Using Binary Search, check if the middle element is the peak element or not,

- If the middle element the peak element terminate the while loop and print middle element.
- Else if check the element on the right side is greater than the middle element then there is always a peak element on the right side, update low = mid+1.
- Else if the element on the left side is greater than the middle element then there is always a peak element on the left side, update high = mid.

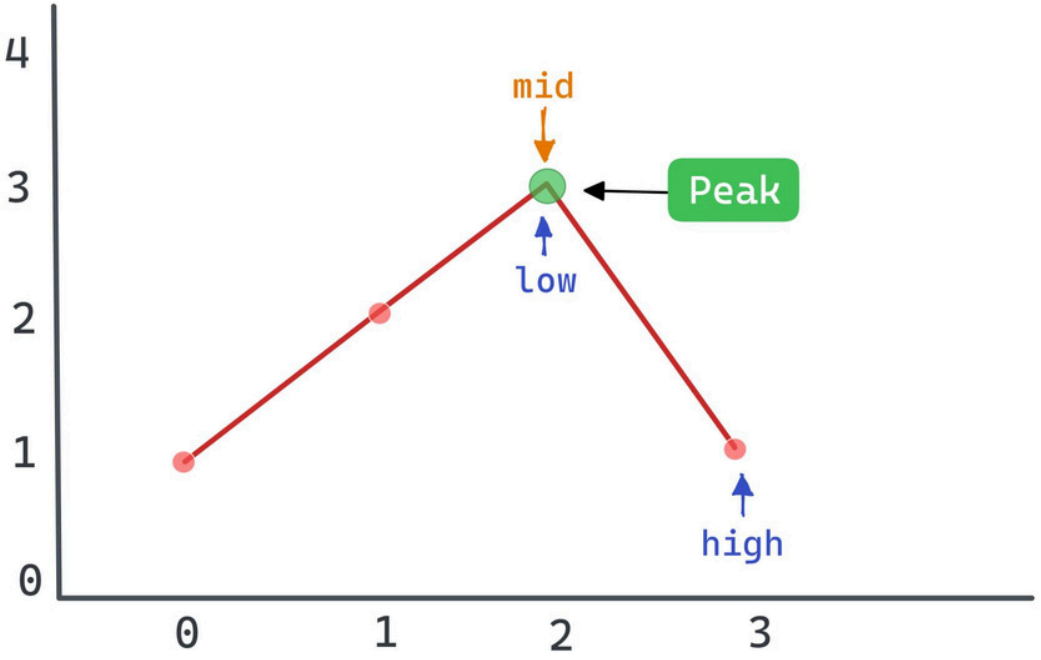


Following corner cases give better idea about the problem.

1. If input array is sorted in strictly increasing order, the last element is always a peak element.
   For example, 50 is peak element in {10, 20, 30, 40, 50}.
2. If input array is sorted in strictly decreasing order, the first element is always a peak
   element. 100 is the peak element in {100, 80, 60, 50, 20}.
3. If all elements of input array are same, every element is a peak element.

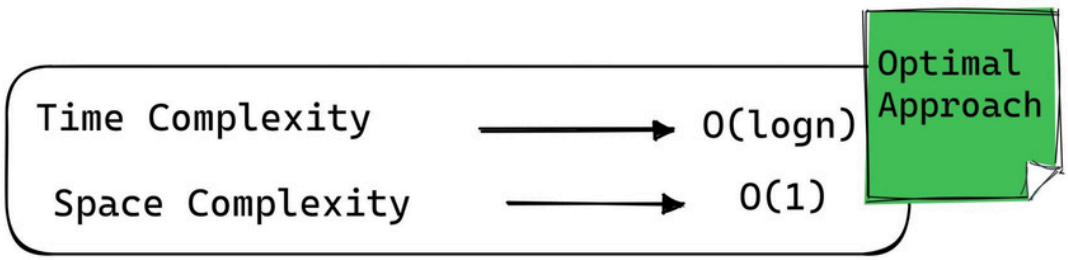



```
arr[mid] < arr[mid+1]
   update low = mid+1
```



Code:-

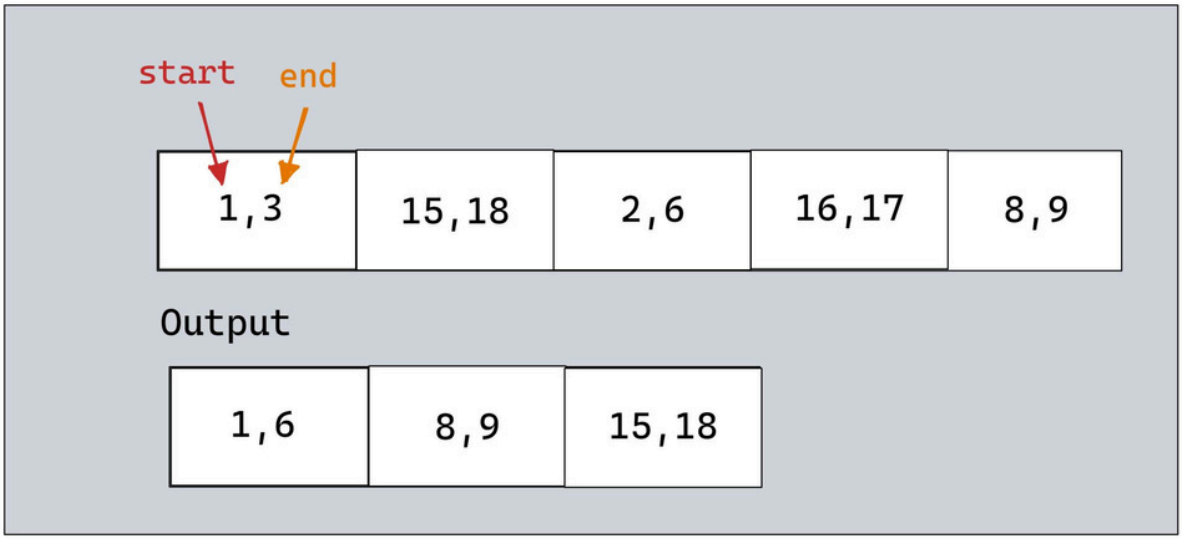
```java
class Solution {
    public int findPeakElement(int[] nums) {

        int low  = 0;
        int high = nums.length-1;

         while(low<high){
             int mid = (low+high)/2;

             if(nums[mid] > nums[mid+1]){
                 high = mid;
             }
             else{
                 low = mid+1;
             }
         }
         return low;
    }
}
```



Merge Intervals

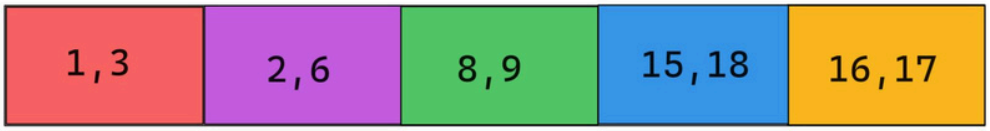
→ Given an array of intervals where intervals[i] = [starti, endi]

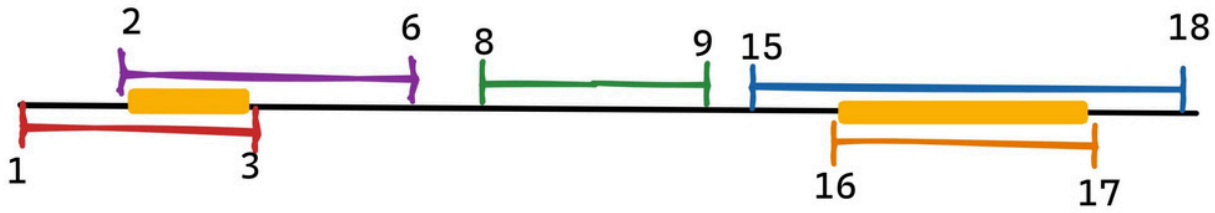
→ Merge all overlapping intervals, and return an array of the non - overlapping intervals



Approach

→ Firstly, sort the given array depending upon the start time.

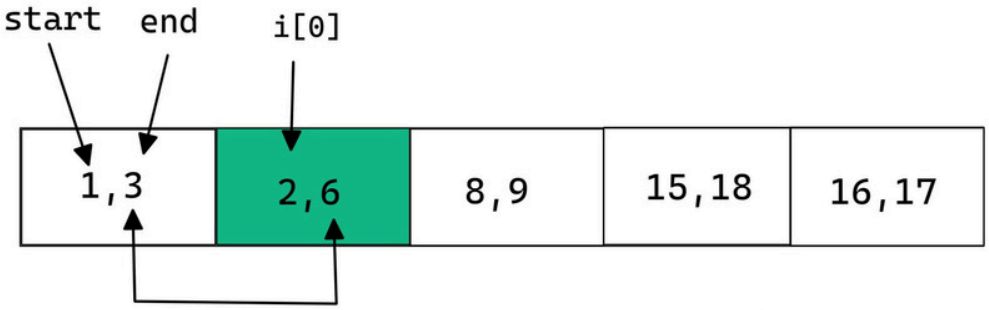




→ Now, we are going to check if start of new interval is less than or equal to end of previous interval, which means there is an possibility of merging.

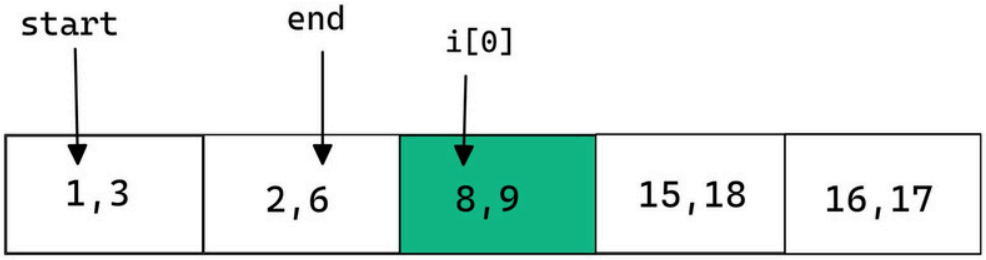
now, check if i[0] ≤ endTime

we can merge it.
so we have to set its end boundoury.



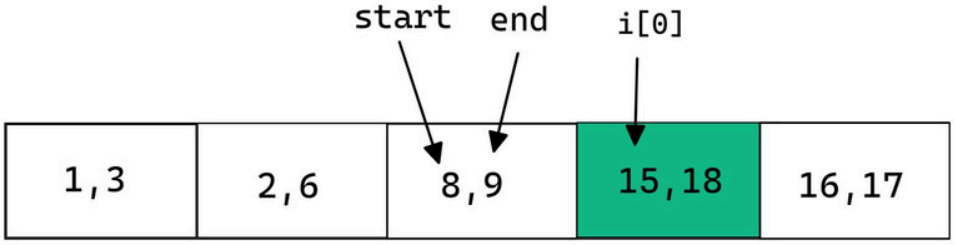
end = Max(end,i[1])

res = 1,6



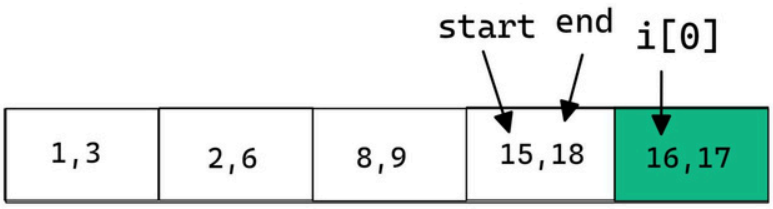
since, i[0] is not ≤ end,
That's mean we can't merge it.

going to set our start & end pointer.
start = i[0]
end = i[1]

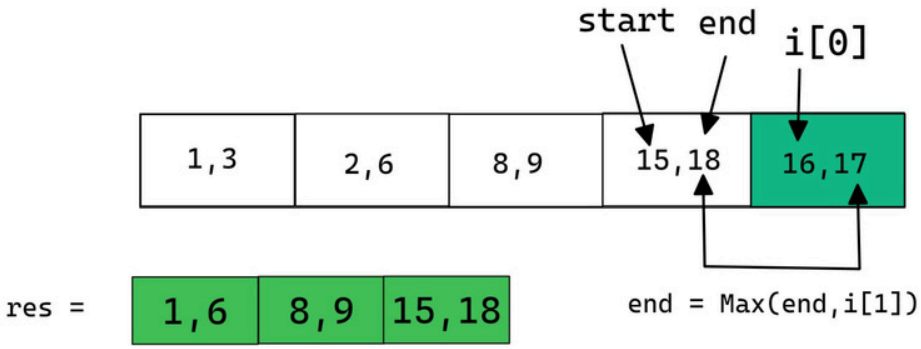


res = 1,6 8,9

i[0] > end,
we can't merge it.
update start & end.



i[0] ≤ end
so, we can merge it.
check its end or i[1] which have greater boundry.



Time-Complexity → O(nlogn) +O(n)

Space-Complexity → O(1)

Code

```
class Solution {
    public int[][] merge(int[][] intervals) {
        List<int[]> res = new ArrayList<>();
        Arrays.sort(intervals,(a,b) -> a[0] - b[0]);

        int start = intervals[0][0];
        int end = intervals[0][1];

        for(int[] i : intervals){
            if(i[0] <= end){
                end = Math.max(end,i[1]);
            }
            else{
                res.add(new int[]{start,end});
                start = i[0];
                end = i[1];
            }
        }
        res.add(new int[]{start,end});
        return res.toArray(new int[0][]);
    }
}
```

3 Sum

Medium

→ Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that
i ≠ j, i ≠ k, and j ≠ k, and
nums[i] + nums[j] + nums[k] == 0.

→ Notice that the solution set must not contain duplicate triplets.

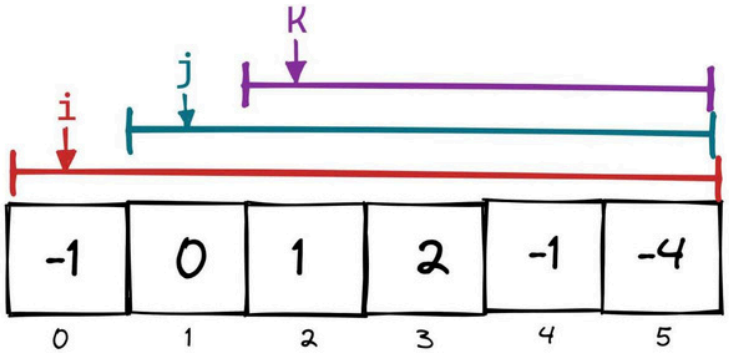
```
Input: nums = [-1,0,1,2,-1,-4]

Output: [[-1,-1,2],[-1,0,1]]

Explanation:
Out of all possible unique triplets possible,
[-1,-1,2] and [-1,0,1] satisfy the condition of
summing up to zero
```

Approach-1

→ Create three nested loop first loop runs from start to end (loop counter i), second loop runs from i+1 to end (loop counter j) and third loop runs from j+1 to end (loop counter k)

→ The counter of these loops represents the index of 3 elements of the triplets.

→ Find the sum of ith, jth and kth element. If the sum is equal to given 0.



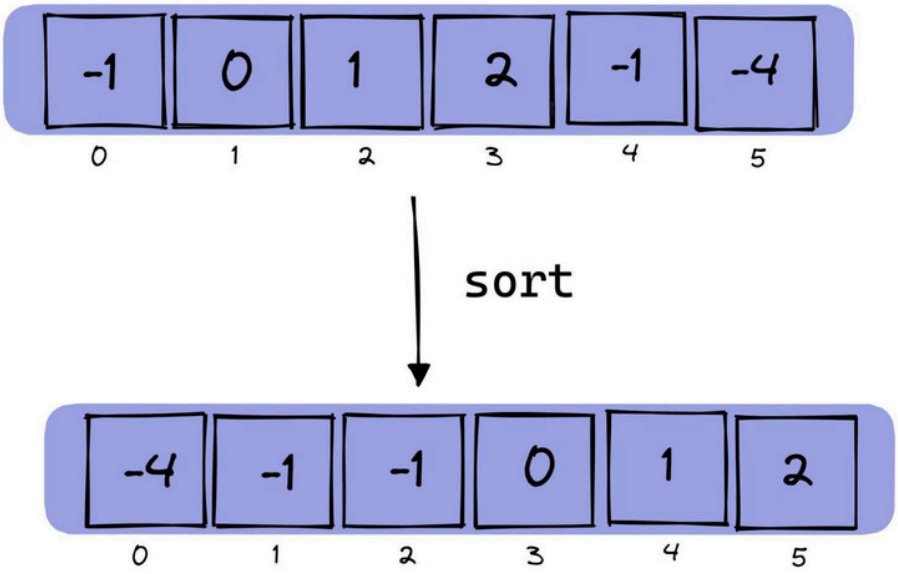
Time-Complexity → O(n^3)

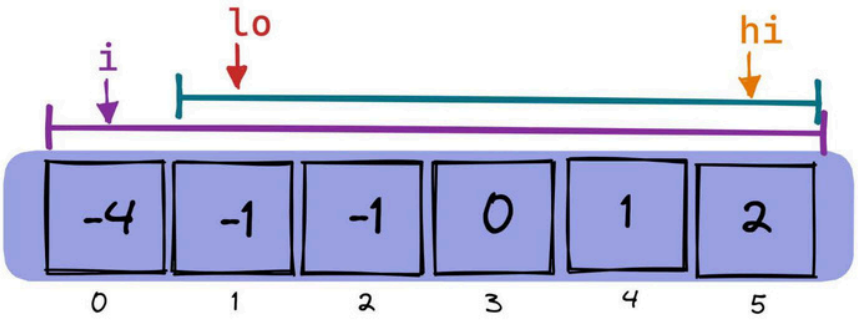
Space-Complexity → O(1)

Approach-2 (Two Pointers + Sorting)

→ We will follow the same two pointers pattern. It requires the array to be sorted, so we'll do that first.

- Sort the Array.
- Notice, that we are fixing the i pointer and then applying the traditional 2 pointer approach to check whether the sum of three numbers equals zero.
  - → If the sum is less than zero, it indicates our sum is probably too less and we need to increment our j pointer to get a larger sum.
  - → if our sum is more than zero, it indicates our sum is probably too large and we need to decrement the k pointer to reduce the sum and bring it closer to zero.






```java
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {

        List<List<Integer>> res = new ArrayList<>();
        Arrays.sort(nums);

        for (int i = 0; i < nums.length-2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }

            int j = i + 1, k = nums.length - 1;
            int target = -nums[i];

            while (j < k) {
                if (nums[j] + nums[k] == target) {
                    res.add(Arrays.asList(nums[i], nums[j], nums[k]));
                    j++;
                    k--;
                    while (j < k && nums[j] == nums[j - 1]) j++;
                    while (j < k && nums[k] == nums[k + 1]) k--;
                } else if (nums[j] + nums[k] > target) {
                    k--;
                } else {
                    j++;
                }
            }
        }
        return res;
    }
}
```

Time-Complexity ⟶ O(nlogn)

Space-Complexity ⟶ O(1)

Santosh Kumar Mishra

Find Duplicate elements

Medium

→ Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

→ There is only one repeated number in nums, return this repeated number.

```
Input: nums = [1,3,4,2,2]

Output: 2
```

Approach 1 (Using Sorting) :

→ Sort the input array (nums).

→ Iterate through the array, comparing the current number to the previous number

Time complexity : ——→ O(nlogn)

Space complexity : ——→ O(1)

Approach 2 (Using Set) :

→ Initialize a HashSet.

→ HashSet is used to store only non-duplicate elements only.

→ If an element is encounter while traversing, and is present in HashSet so, that a duplicate element.

Time complexity : ——→ O(n)

Space complexity : ——→ O(n)

in Santosh Kumar Mishra

Approach 3 (negative marking):

→ Use elements as Index and mark the visited places.

```
value = Math.abs(arr[i])

if(arr[value-1] is positive)
    make arr[value-1] negative.

if(arr[value-1] is negative)
    "Found Repeated element"
```

Time complexity :⟶ O(n)

Space complexity :⟶ O(1)

Code:-

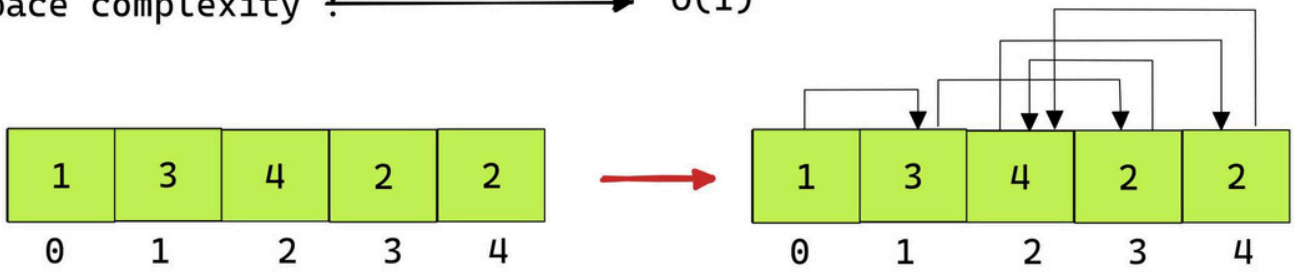
```java
class Solution {
    public int findDuplicate(int[] nums) {

        for(int i = 0;i<nums.length;i++){
            int value = Math.abs(nums[i]);

            if(nums[value-1] > 0){
                nums[value-1] = -nums[value-1];
            }
            else{
                return value;
            }
        }
        return -1;
    }
}
```

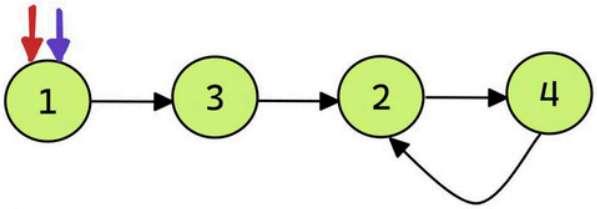
Approach 4 (Floyd's Tortoise and Hare) :

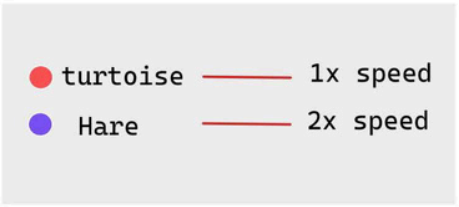
Since the hare goes fast, it would be the first to enter the cycle and run around the cycle.At some point, the tortoise enters the cycle as well, and since it's moving slower the hare catches up to the tortoise at some intersection point.

Time complexity :⟶ O(n)

Space complexity :⟶ O(1)







Code:-

```
class Solution {
public int findDuplicate(int[] arr) {
        int slow = arr[0];
        int fast = arr[arr[0]];

        while (fast != slow){
            slow = arr[slow];
            fast = arr[arr[fast]];
        }
        // loop to find entry
        fast = arr[0];
        while (slow != fast){
            slow = arr[slow];
            fast = arr[fast];
        }
        return slow;
    }
}
```

Maximum Subarray

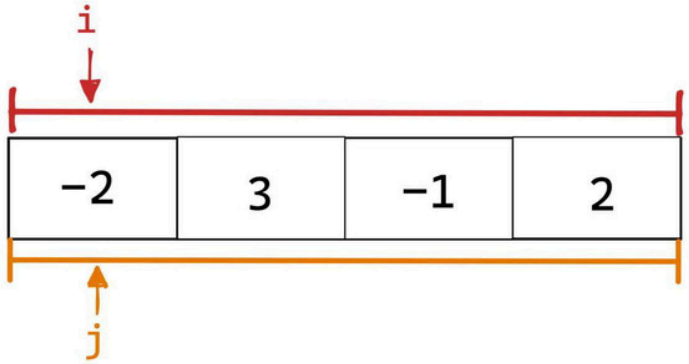
Medium

- Given an integer array nums, find the contiguous subarray which has the largest sum and return its sum.
- A subarray is a contiguous part of an array.

```
Arr[] = [-2,3,-1,2]

Output :- 4   [3,-1,2]
```

Approach-1

→ Using two for loops, outer loop from i=0 → n and inner loop from j=i → n.And keep a track of every current sum in each sub-array.




```
currSum += arr[j]
maxSum = Math.max(maxSum,currSum)
```

→ Declare two variable currSum which is inside outer loop to keep track of subarray sum.

→ And maxSum which is declare outside the outer loops to store the maximum sum which came from currSum.

Time Complexity	O(n^2)
Space Complexity	O(1)

Santosh Kumar Mishra

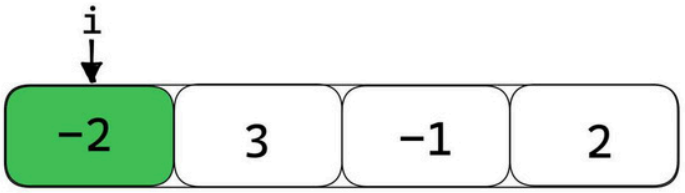
Kadane's Algorithm

→ The most optimal solution for obtaining the maximum sub-array is Kadane's algorithm; it uses two variables:

- currSum → To keep track of whether or not the value at the current index would increase the maximum sum.
- maxSum → To keep track of the overall maximum that is propagated along the array.

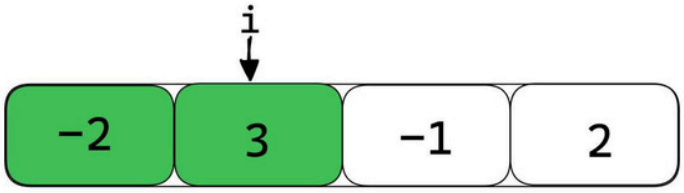
Steps :-

- Set both of the above-mentioned variables to the value at the first index, i.e., arr[0].
- For the next index i, store the maximum of arr[i] and currSum + arr[i] in currSum itself.
- Store the maximum of maxSum and currSum in maxSum.
- Repeat the above two steps for the remaining indices.
- Return the value of maxSum.



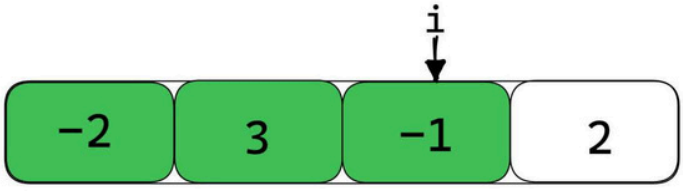
currSum = -2

maxSum = -2



currSum = max(3,-2+3) = 3

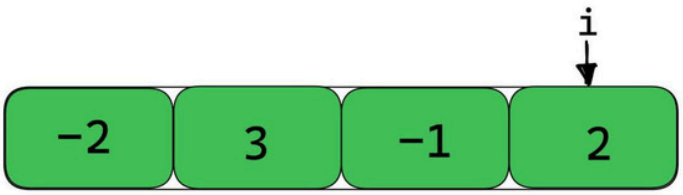
maxSum = max(3,-2) = 3



currSum = max(-1,3-(-1)) = 4

maxSum = max(3,4) = 4

Santosh Kumar Mishra



currSum = max(2,4-2) = 2

maxSum = max(4,2) = 4

Time Complexity ⟶ O(n)

Space Complexity ⟶ O(1)

Optimal Code

```
class Solution {
    public int maxSubArray(int[] nums) {

        int curr_sum = 0;
        int sum = nums[0];

        for(int i = 0;i<nums.length;i++){
            curr_sum += nums[i];

            if(curr_sum > sum){
                sum = curr_sum;
            }
            if(curr_sum < 0){
                curr_sum = 0;
            }
        }
        return sum;
    }
}
```

Kth Largest Element in an Array

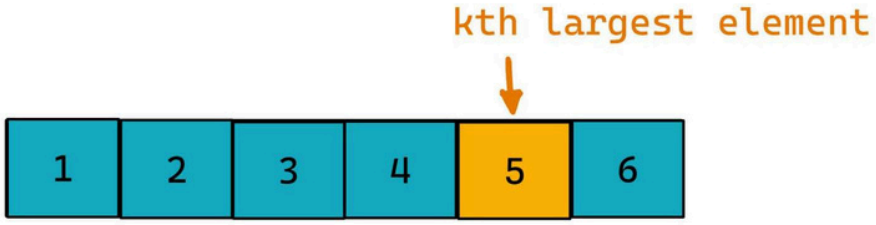
Medium

- Given an integer array nums and an integer k, return the kth largest element in the array.
- Note that it is the kth largest element in the sorted order, not the kth distinct element.

```
Input: nums = [3,2,1,5,6,4], k = 2

Output: 5
```

Brute-Force Approach :

- The naive solution would be to sort an array first and then return kth element from the end




```
Arrays.sort(nums);
return nums[nums.length-k];
```

Time Complexity → O(nlogn)

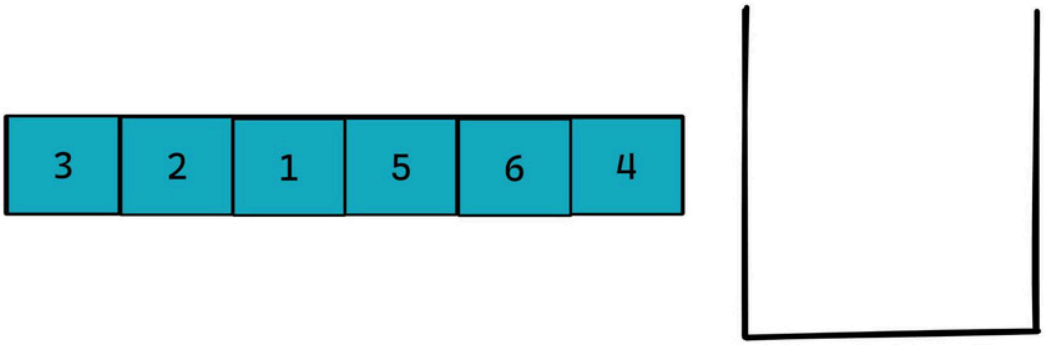
Space Complexity → O(1)

Brute Approach

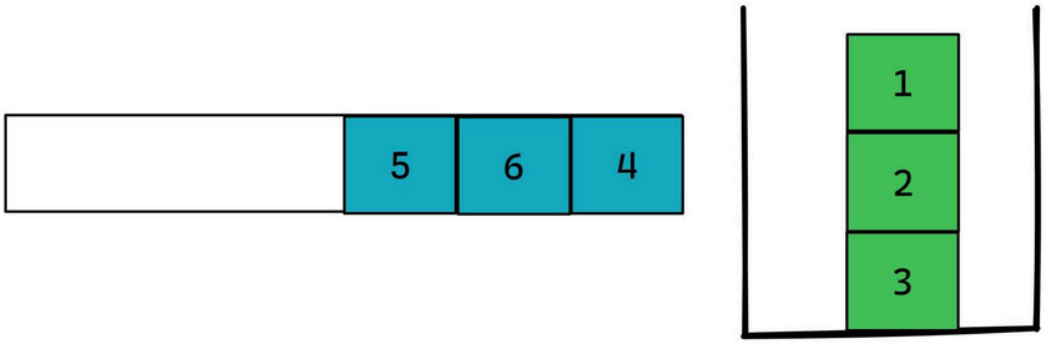
Santosh Kumar Mishra

Optimal Approach :

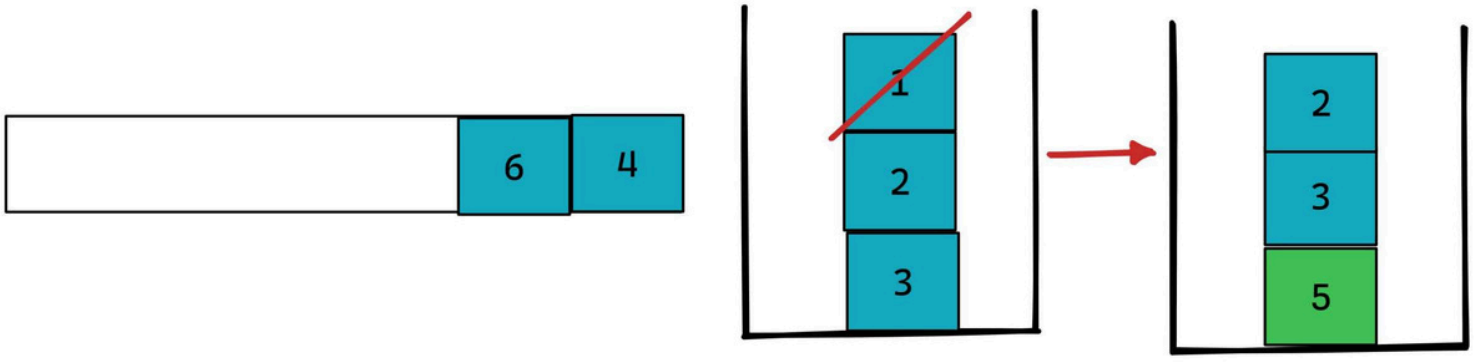
→ The idea is to init a heap "the smallest element first", and add all elements from the array into this heap one by one keeping the size of the heap always less or equal to k.



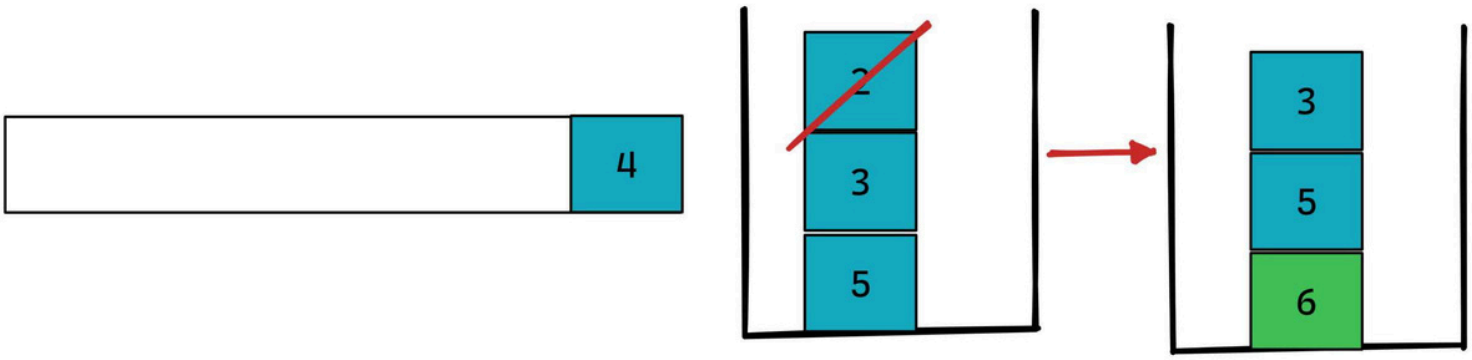
→ fill the values until heap.size() ≤ k,



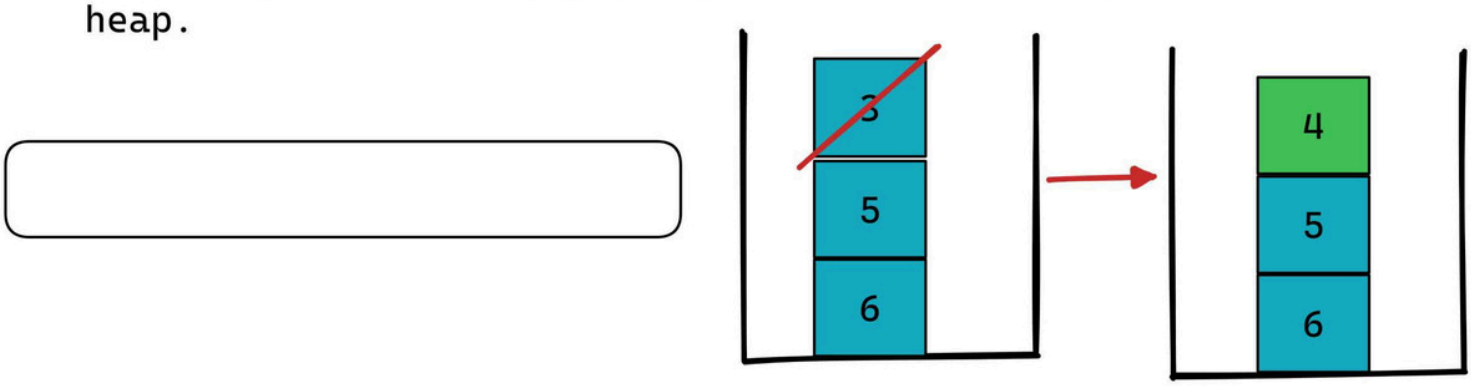
→ Now, heap.size() > K, pop top element from heap. And add node 5 in heap.



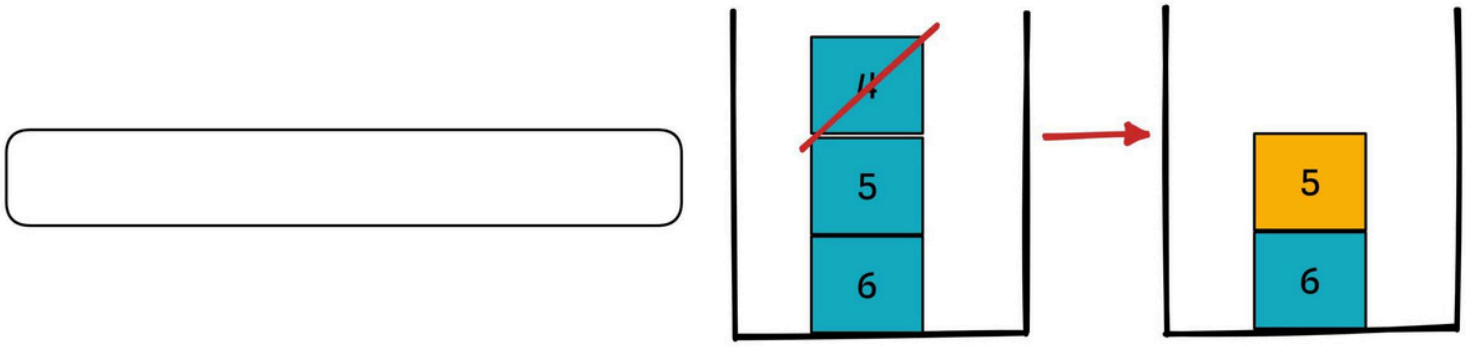
→ Now, heap.size() > K, pop top element from heap.And add node 6 in heap.



→ Now, heap.size() > K, pop top element from heap.And add node 4 in heap.



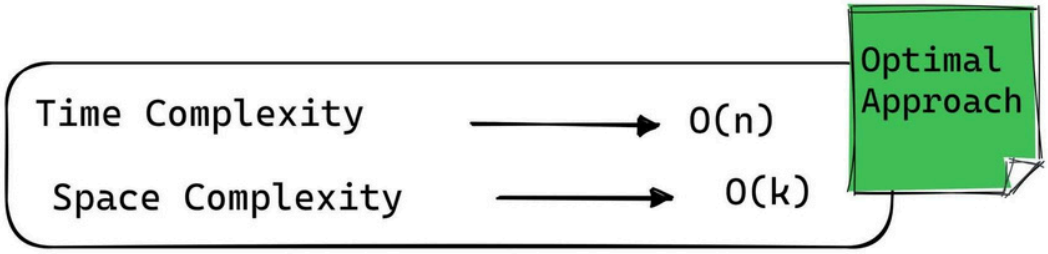
→ Now, heap.size() > K, pop top element from heap.



◆ Just Return the top element in heap(5), heap.poll(). That our kth largest element.

Code:-

```java
class Solution {
    public int findKthLargest(int[] nums, int k) {

        PriorityQueue<Integer> heap = new
        PriorityQueue<Integer>((n1,n2) -> n1- n2);

        for(int n : nums){
            heap.add(n);

            if(heap.size() > k){
                heap.poll();
            }
        }

        return heap.poll();
    }
}
```



Increasing Triplet Subsequence

Medium

- Given an integer array nums, return true if there exists a triple of indices (i, j, k) such that i < j < k and nums[i] < nums[j] < nums[k].
- If no such indices exists, return false.

Input:- | 5 | 0 | 4 | 6 |

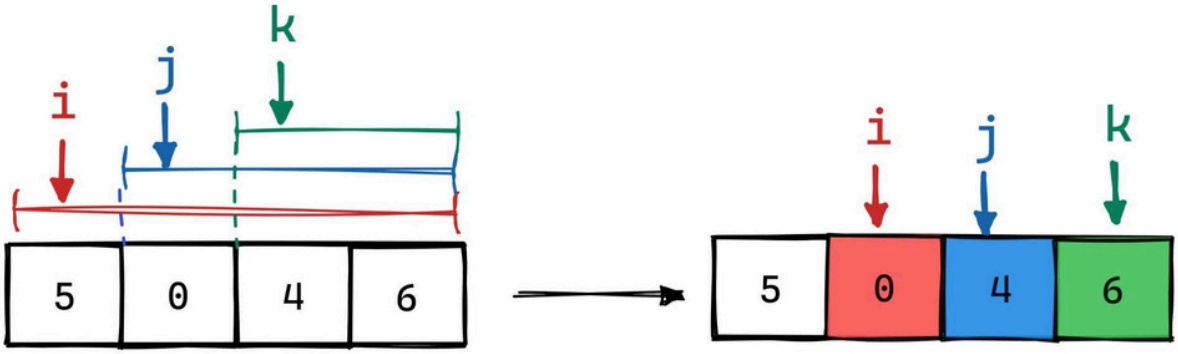
Output:- True

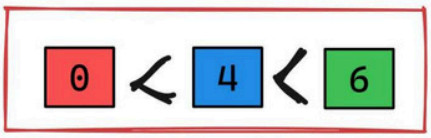
Approach-1

→ Using Three for loops i,j & k. Where i is going to start from 0 , j = i+1 & k = j+1.

→ we Have to check these conditions

- i<j<k
- nums[i]<nums[j] && nums[i]<nums[k] && nums[j]<nums[k]





Time-Complexity ——→ O(n^3)

Space-Complexity ——→ O(1)

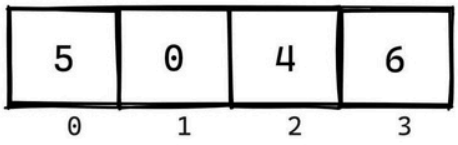
in Santosh Kumar Mishra

Approach-2

→ What we are going to do is firstly we are going to find first smallest & secound smallest value in a given Subsequence.

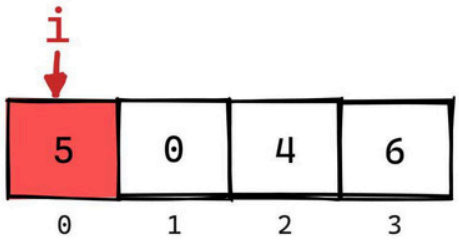
→ Then we are going to check if there is a value which is bigger than first & secound smallest value in given sequence.

→ Then we can say Its a Increasing triplet subsequence.



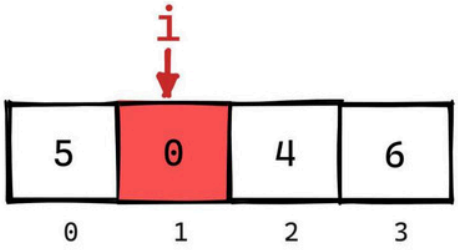
firstNum = Integer.MAX_VALUE

secoundNum = Integer.MAX_VALUE



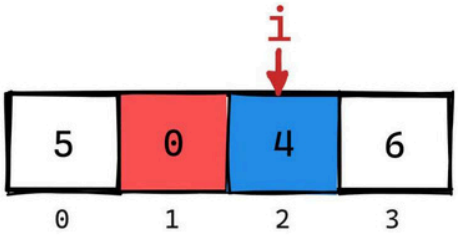
firstNum = 5

secoundNum = Integer.MAX_VALUE



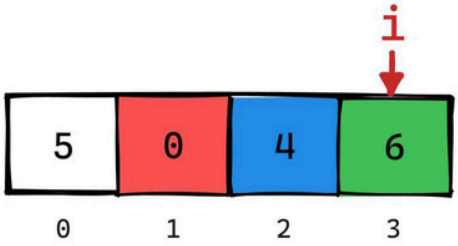
firstNum = 0

secoundNum = Integer.MAX_VALUE



firstNum = 0

secoundNum = 4



firstNum = 0

secoundNum = 4

0 < 4 < 6

Note: when nums[i]>firstNum && nums[i]>secoundNum , which means there is a increasing triplet subsequence.

Time-Complexity ——→ O(n)

Space-Complexity ——→ O(1)

- Code

```
class Solution {
    public boolean increasingTriplet(int[] nums)
{       int firstNum = Integer.MAX_VALUE;
        int secoundNum = Integer.MAX_VALUE;

        for(int i = 0;i<nums.length;i++){
            if(nums[i] <= firstNum){
                firstNum = nums[i];
            }
            else if(nums[i] <= secoundNum){
                secoundNum = nums[i];
            }
            else{
                return true;
            }
        }
        return false;
    }
}
```

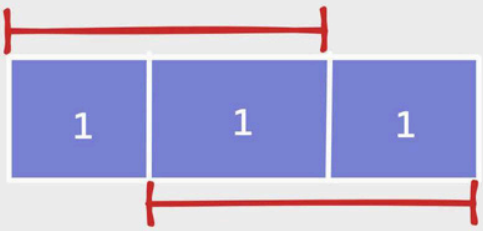
Subarray Sum Equals K

Medium

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

→ A subarray is a contiguous non-empty sequence of elements within an array.

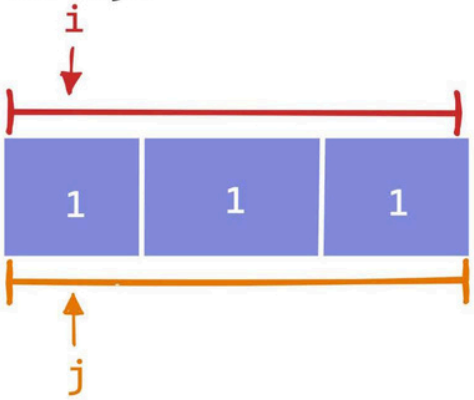
Input: nums = [1,1,1] , k = 2



Output: 2

Brute-Force Approach :

→ A simple solution is to traverse all the subarrays and calculate their sum.If the sum is equal to the required sum, then increment the count of subarrays



Time Complexity → O(n^2)

Space Complexity → O(1)

Brute Approach

Santosh Kumar Mishra

Optimal Approach : (Using HashMap)

→ we make use of a hashmap which is used to store the cumulative sum up to all the indices possible along with the number of times the same sum occurs.

HashMap(prefixSum , occurences of prefixSum)

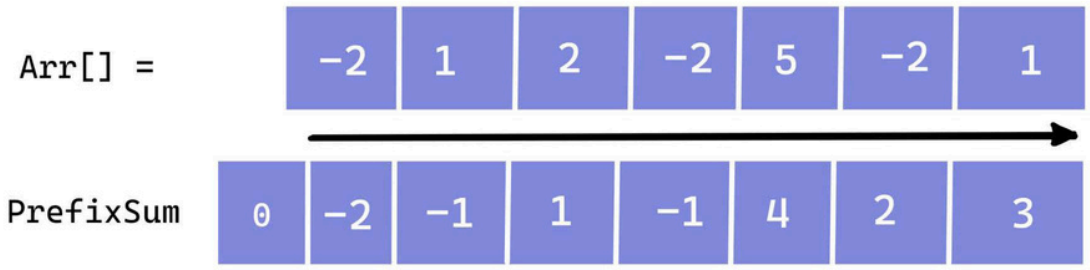
We traverse over the array nums and keep on finding the cumulative sum.

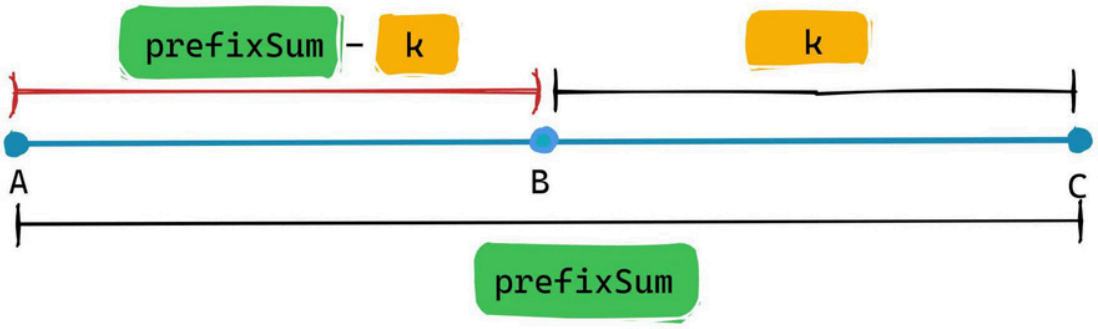
→ Every time we encounter a new sum, we make a new entry in the hashmap corresponding to that sum.

If the same sum occurs again, we increment the count corresponding to that sum in the hashmap.

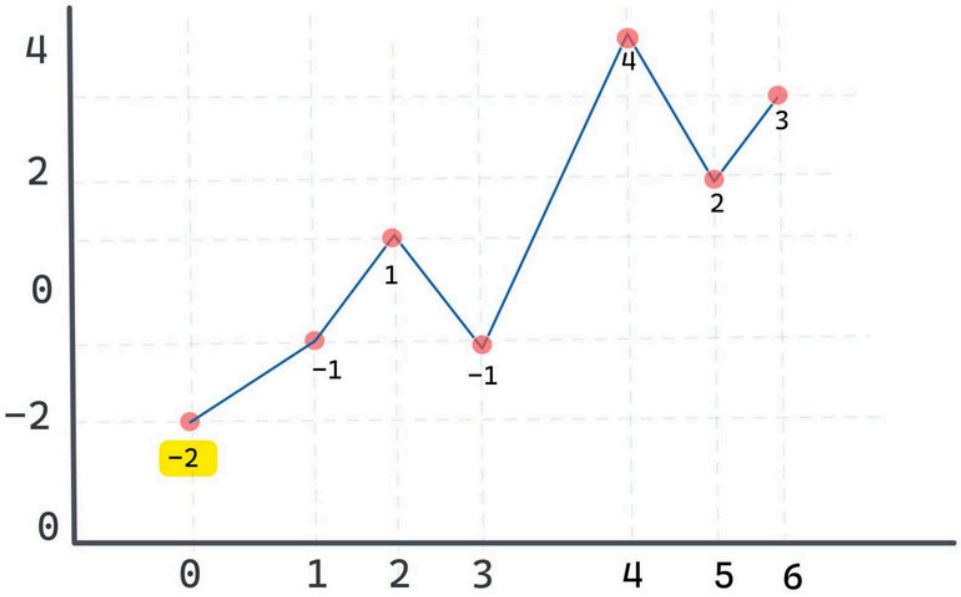
we are going to keep a track of prefixSum-k count every time we encounter it in our hashmap.

→ After the complete array has been traversed, the count gives the required result.





Initially we are going to initilize our Hashmap with (0,1)



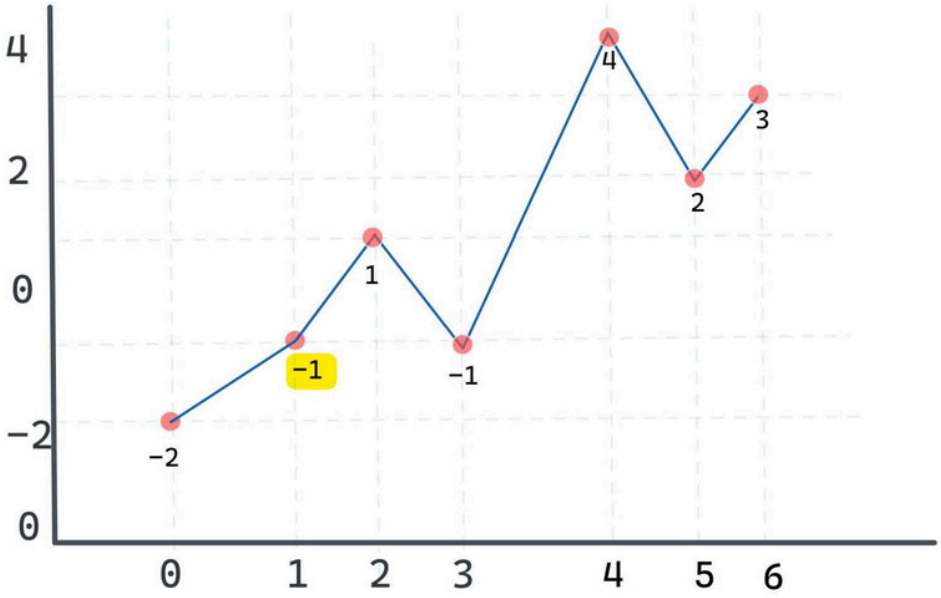
prefixSum - k

-2 - 3 = -5 ✗

-2 : 1
0 : 1

ans = 0

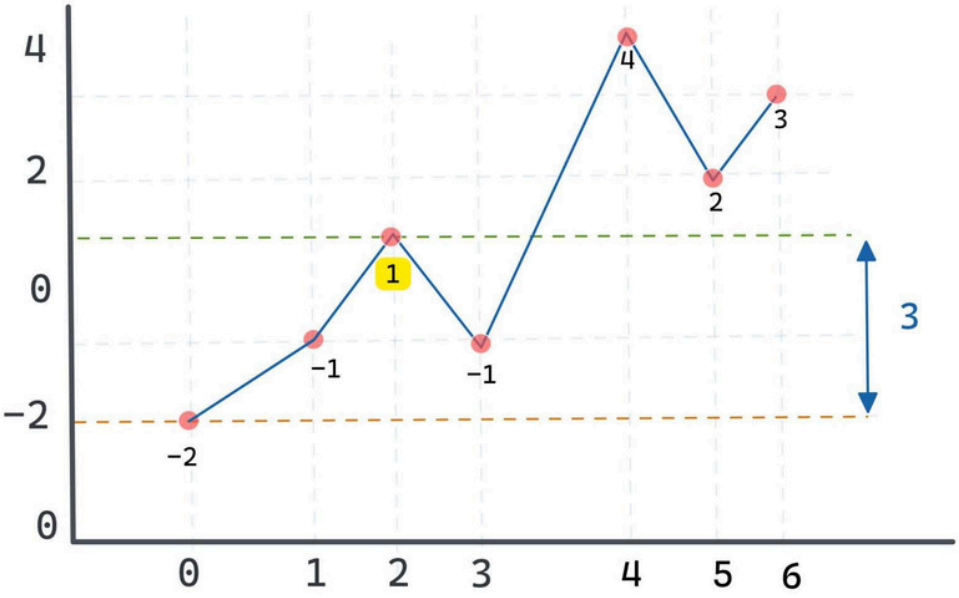
since, -5 is not present in our Hashmap. put prefixSum and its occurrence till now.



-1-3 = -4 ✗

-1 : 1
-2 : 1
0 : 1

ans = 0

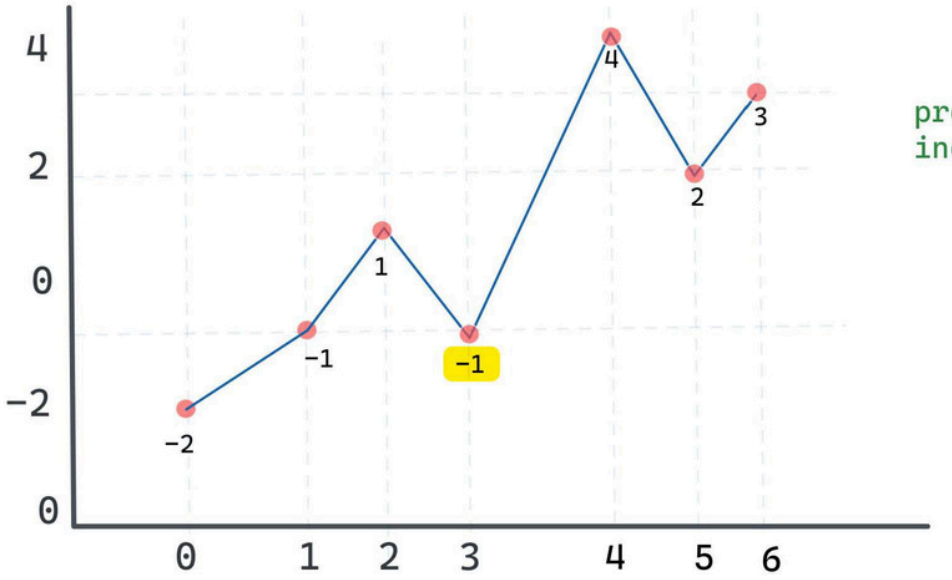


1-3 = -2 ✓

1 : 1
-1 : 1
-2 : 1
0 : 1

ans = 1

in Santosh Kumar Mishra

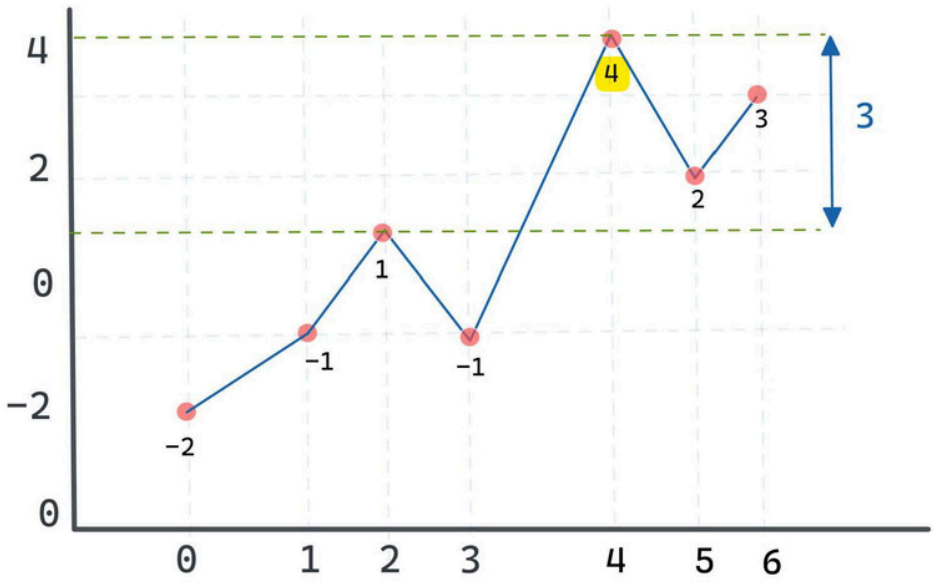


-1-3 = -4 ✗

prefixSum present in our map increase its count.

1	1
-1	~~1~~ 2
-2	1
0	1

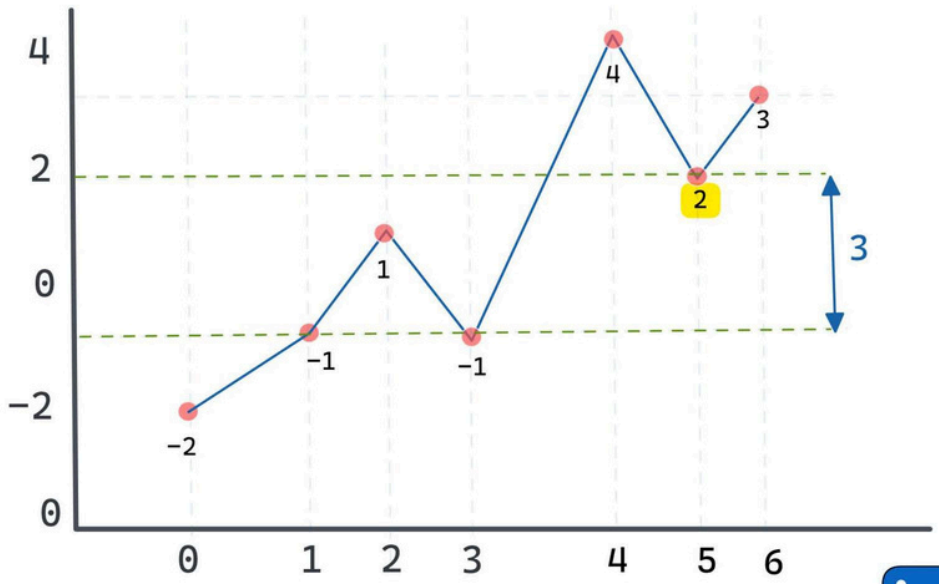
ans = 1



4-3 = 1 ✓

ans = 2

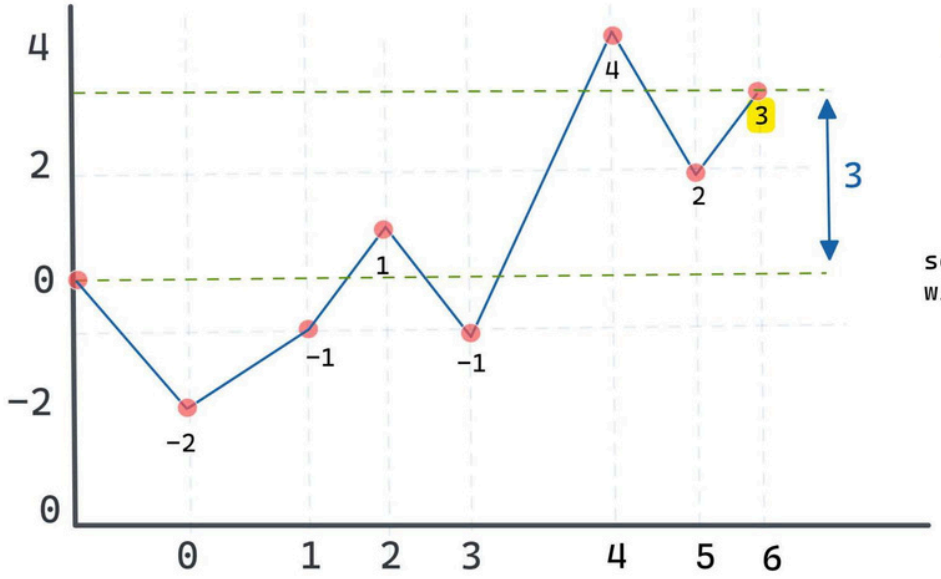
4	1
1	1
-1	2
-2	1
0	1



2-3 = -1 ✓

ans = 4

2	1
4	1
1	1
-1	2
-2	1
0	1



Note → Base case

so, we have to init. hasmap with 0 prefixSum.

ans = 5

Code:-

```java
public class Solution {
    public int subarraySum(int[] nums, int k) {

        int ans = 0, prefixSum = 0;
        HashMap < Integer, Integer > map = new HashMap < > ();
        map.put(0, 1);

        for (int i = 0; i < nums.length; i++) {
            prefixSum += nums[i];
            if (map.containsKey(prefixSum - k)){
              ans += map.get(prefixSum - k);
            }
            map.put(prefixSum, map.getOrDefault(prefixSum, 0) + 1);
        }
        return ans;
    }
}
```

Time Complexity → O(n)

Space Complexity → O(n)

Optimal Approach

Santosh Kumar Mishra

Valid Anagram

Easy

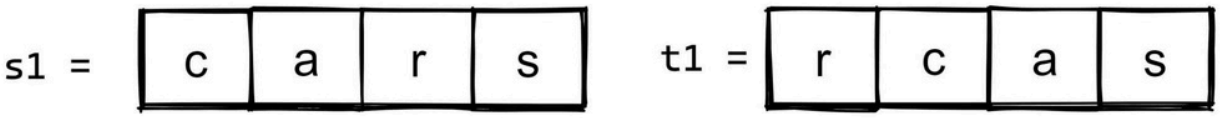
- Given two strings s and t, return true if t is an anagram of s, and false otherwise.
- An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

```
Input:-   s = "cars" , t = "rcas"

Output:-  True
```

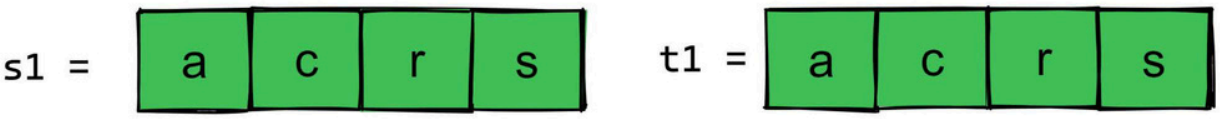
Approach-1 (Sorting)

→ we have to convert both the string into character Array.

s = "cars" , t = "rcas"



→ After that sort both the char Array & check if they are equal or not.



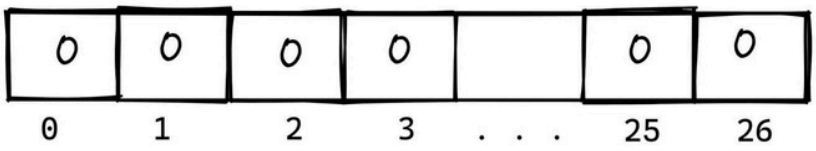
since,
both char Array are equal,it means string s & t is Anagram.

Time Complexity	O(nlogn)
Space Complexity	O(1)

Santosh Kumar Mishra

Approach-2 (temp array)

→ For This Approach we have to create an Array of size 26 for each letter in alphabet.



→ We are going to use String s to increment the value at that particular index with respect to its alphabet.

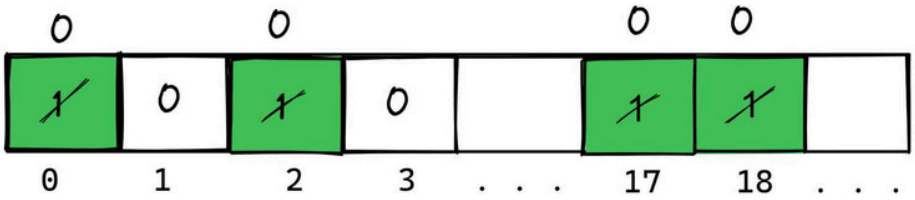
```
s = "cars"
```

```
arr[s.charAt(i) - 'a']++
```

→ We are going to use String t to decrement the value at that particular index with respect to its alphabet.

```
t = "rcas"
```

```
arr[t.charAt(i) - 'a']--
```



→ Now check if that Array became empty or not. if its empty so, string s & t are Anagram.

Time Complexity	O(n)
Space Complexity	O(n)

Note:- Also try HashMap Approach Same as Array Approach.

Approach-1 Code

```
class Solution {
    public boolean isAnagram(String s, String t) {

        char[] s1 = s.toCharArray();
        char[] t1 = t.toCharArray();

        Arrays.sort(s1);
        Arrays.sort(t1);

        return Arrays.equals(s1,t1);
    }
}
```

Approach-2 Code

```
class Solution {
    public boolean isAnagram(String s, String t)
{     int[] arr = new int[26];

        for(int i = 0;i<s.length();i++){
            arr[s.charAt(i) - 'a']++;
        }
        for(int i = 0;i<t.length();i++){
            arr[t.charAt(i) - 'a']--;
        }
        for(int i : arr){
            if(i != 0) return false;
        }
        return true;
    }
}
```

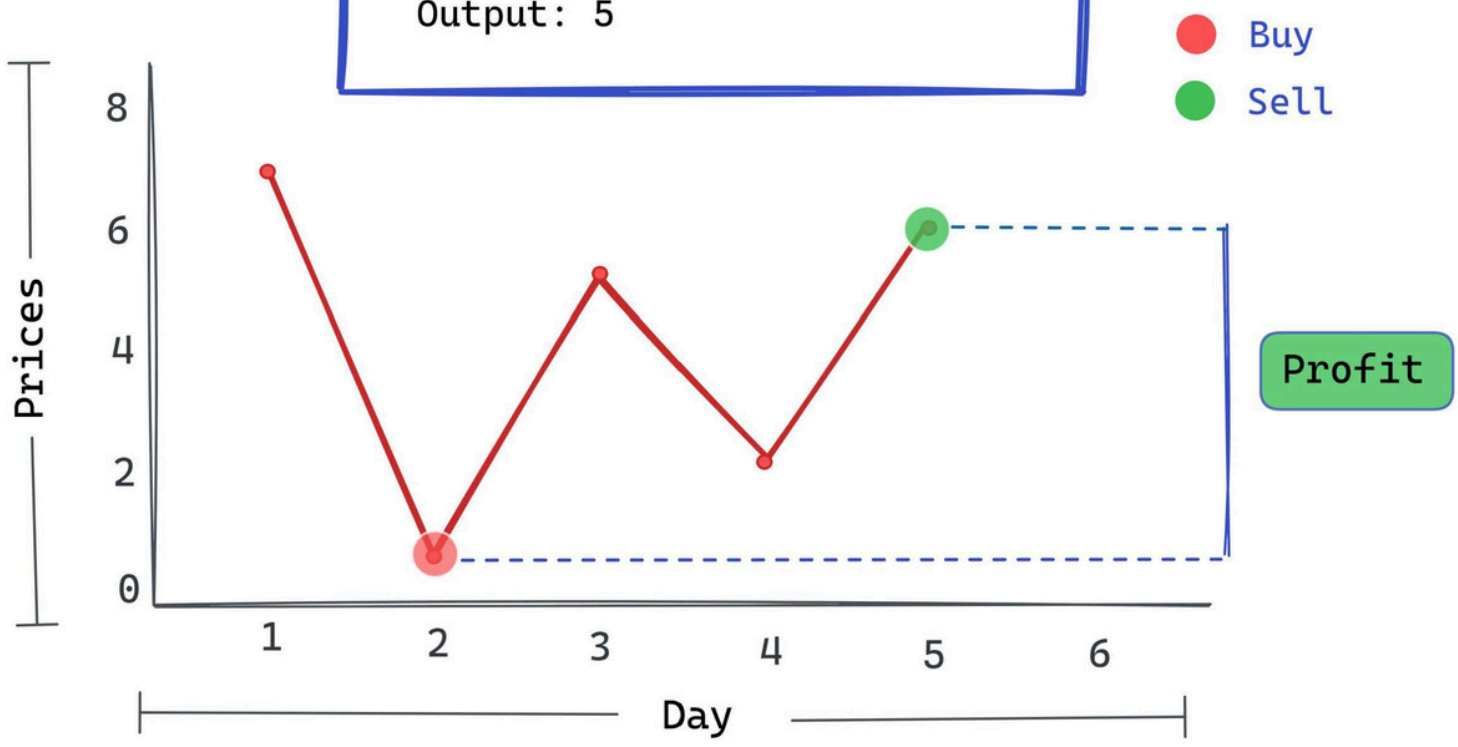
Santosh Kumar Mishra

Best Time to Buy and 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

Example 1 :

```
Input: prices = [7,1,5,3,6]

Output: 5
```



Brute-Force Approach :

- We need to find out the maximum difference between two numbers in the given array.
- Also, the second number (selling price) must be larger than the first one (buying price).

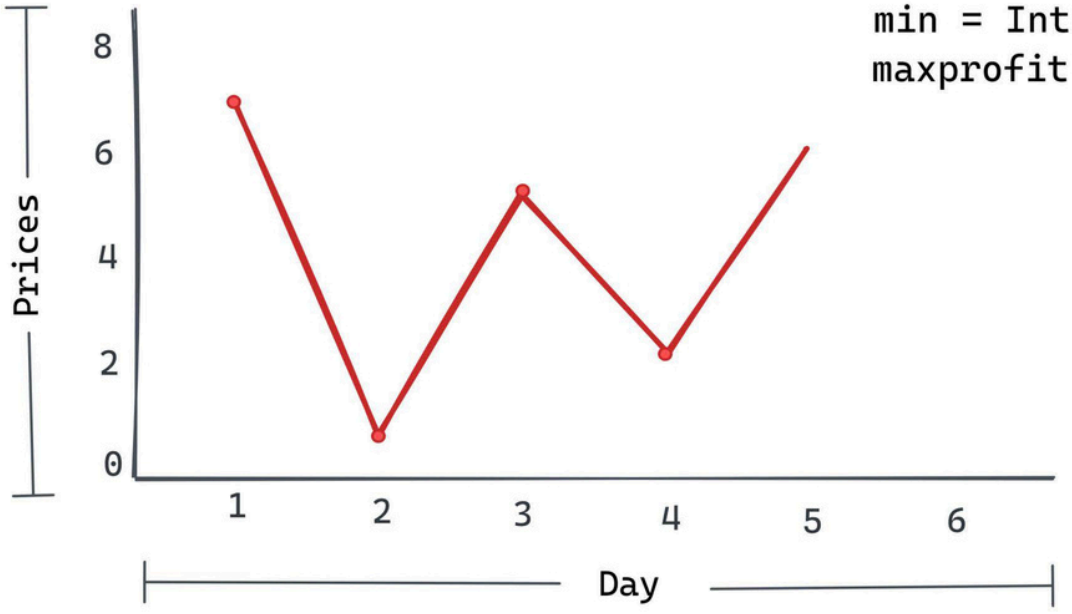
Code:-

max(prices[j] - prices[i])

```java
public class Solution {
    public int maxProfit(int prices[]) {

        int maxprofit = 0;

        for (int i = 0; i < prices.length - 1; i++) {
            for (int j = i + 1; j < prices.length; j++) {

                int profit = prices[j] - prices[i];
                if (profit > maxprofit)
                    maxprofit = profit;
            }
        }
        return maxprofit;
    }
}
```

Optimal Approach

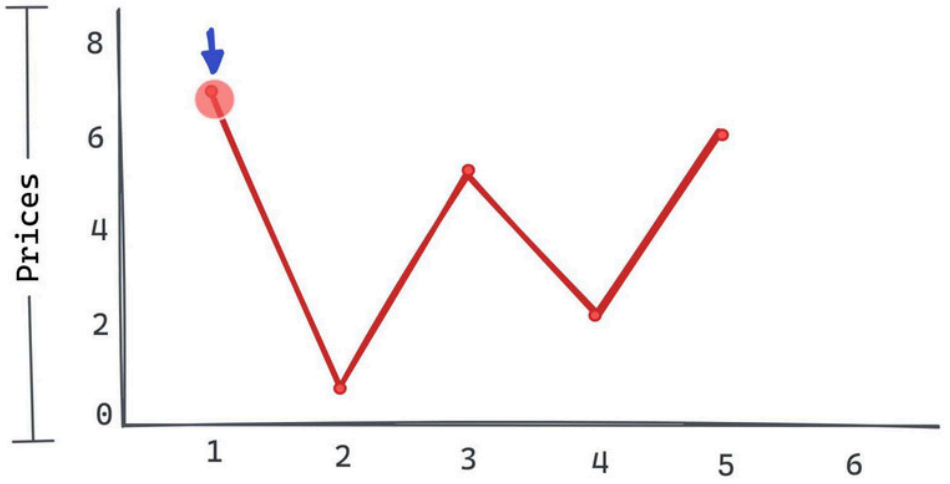
- we are going to initilize two variables min & max.
- And if the difference between maxprofit - min > maxprofit.Then we are going to update our maxprofit.



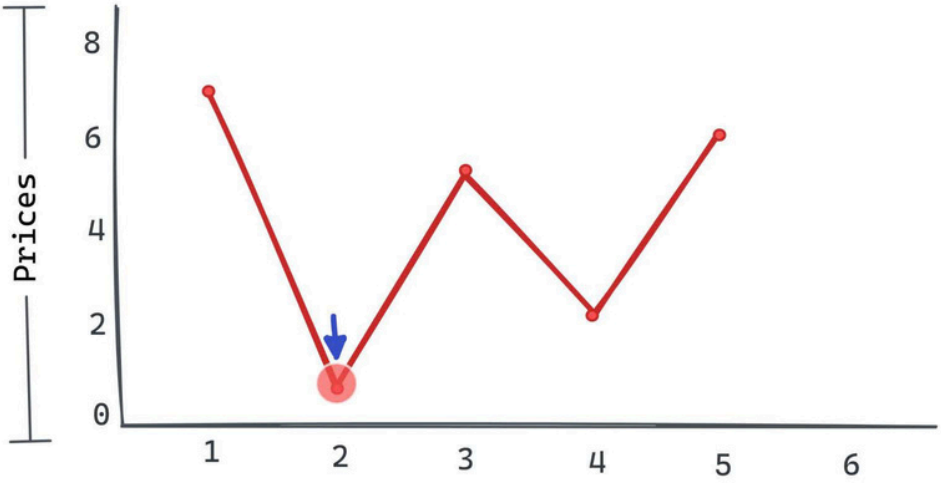
min = Integer.MAX_VALUE
maxprofit = 0

Buy
Sell

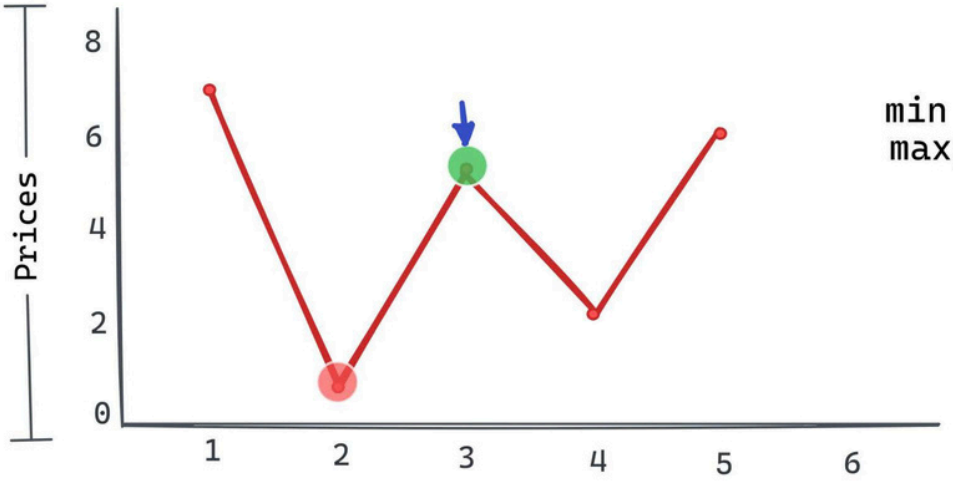
in Santosh Mishra



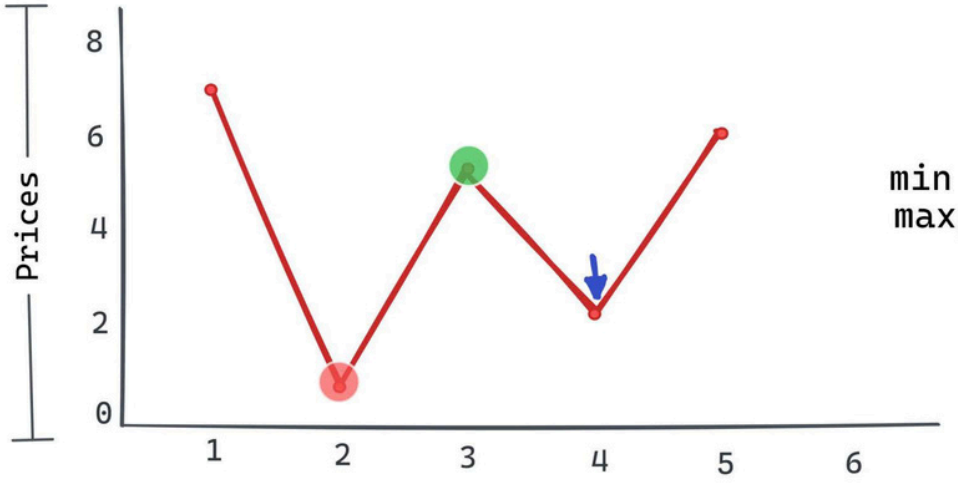
min = 7
maxprofit = 0



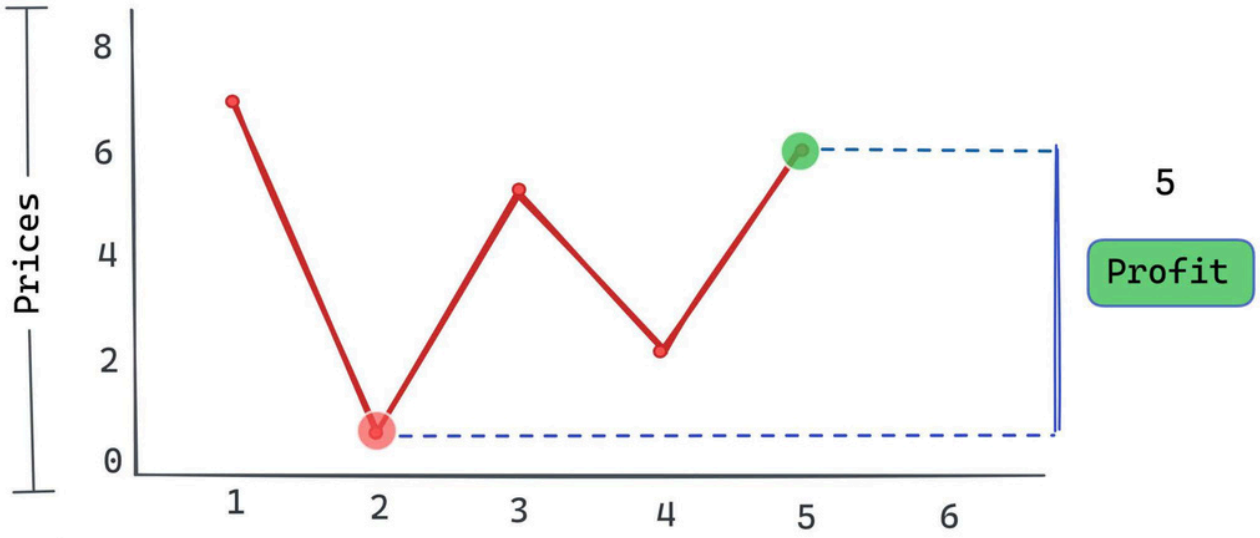
min = 1
maxprofit = 0



min = 1
maxprofit = 5-1 = 4



min = 1
maxprofit = 5-1 = 4



Code:-

```java
public int maxProfit(int prices[]) {

        int minprice = Integer.MAX_VALUE;
        int maxprofit = 0;

        for (int i = 0; i < prices.length; i++) {

            if (prices[i] < minprice)
                minprice = prices[i];

            else if (prices[i] - minprice > maxprofit)
                maxprofit = prices[i] - minprice;

        }
        return maxprofit;
    }
```

Time Complexity → O(n*n)

Space Complexity → O(1)

Brute Approach

Time Complexity → O(n)

Space Complexity → O(1)

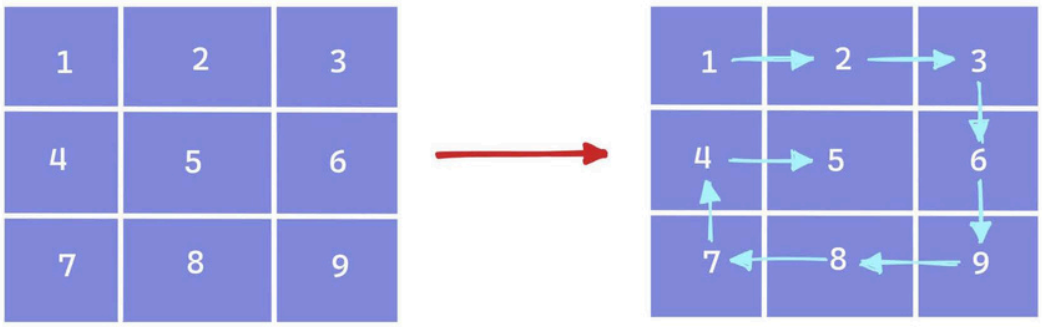
Optimal Approach

Spiral Matrix

Medium

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

Example 1 :




```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]

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

Approach 1: (Set Up Boundaries)

→ First, four variables containing the indices for the corner points of the array are initialized.

→ The algorithm starts from the top left corner of the array, and traverses the first row from left to right. Once it traverses the whole row it does not need to revisit it, thus, it increments the top corner index.

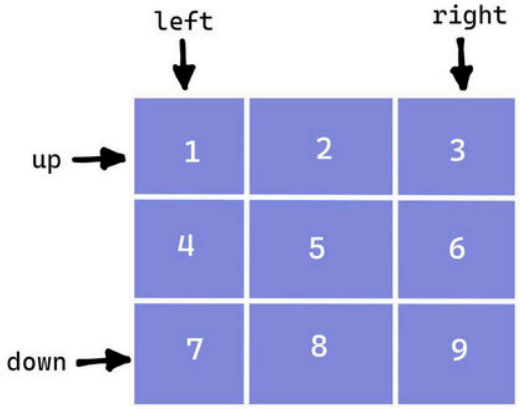
in Santosh Kumar Mishra

→ Once complete, it traverses the rightmost column top to bottom. Again, once this completes, there is no need to revisit the rightmost column, thus, it decrements the right corner index.

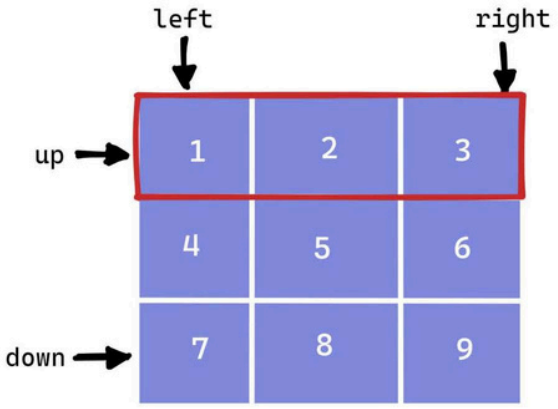
→ Next, the algorithm traverses the bottom most row and decrements the bottom corner index afterward.

→ Lastly, the algorithm traverses the leftmost column, incrementing the left corner index once it's done.

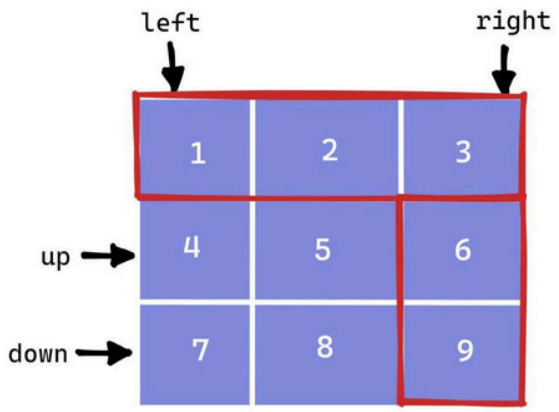
This continues until the left index is greater than the right index, and the top index is greater than the bottom index.



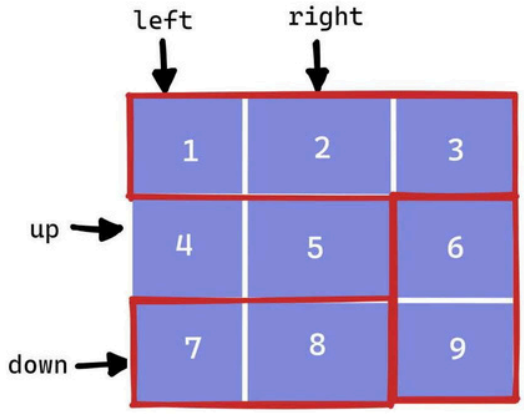

```
left ≤ right && up ≤ bottom
```



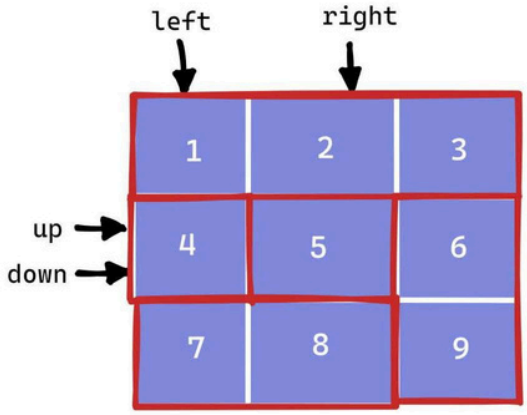

```
left ⟶ right
  up++
```



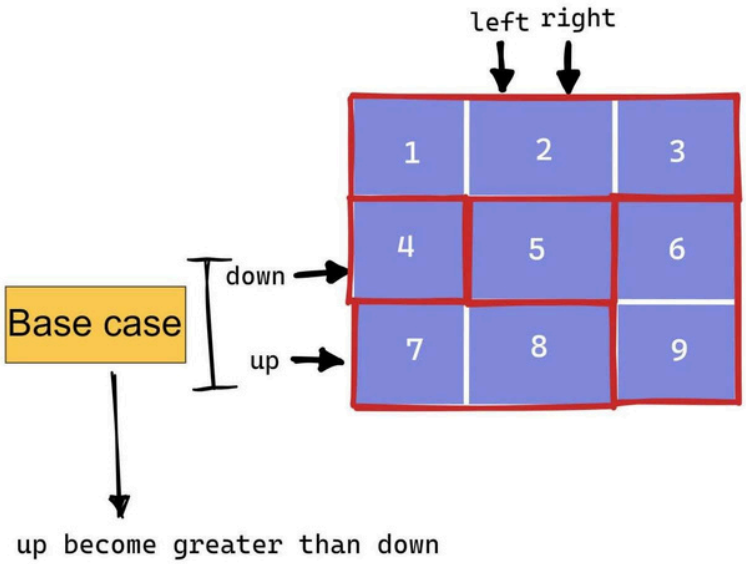

```
up ⟶ down
  right--
```




```
up ≤ bottom

right ⟶ left
   down--
```




```
left ≤ right

down ⟶ up
left++
```



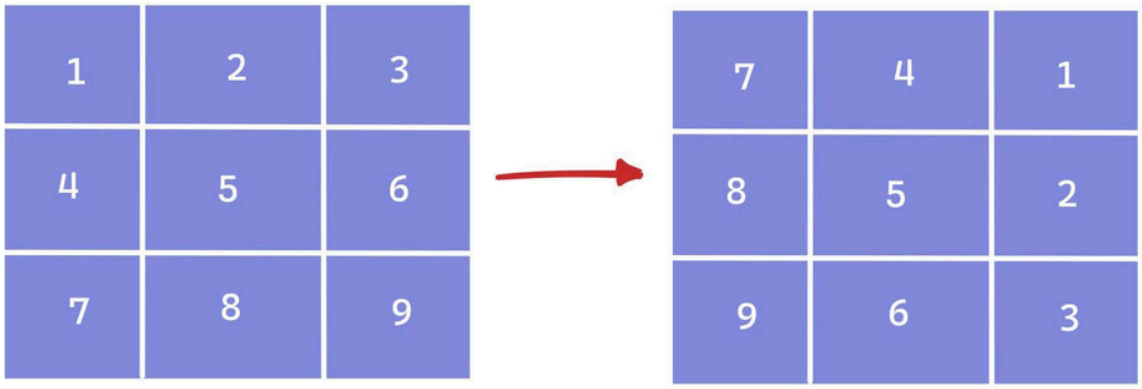

```
left ⟶ right
   up++
```

up become greater than down

Rotate Image

→ You are given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise).

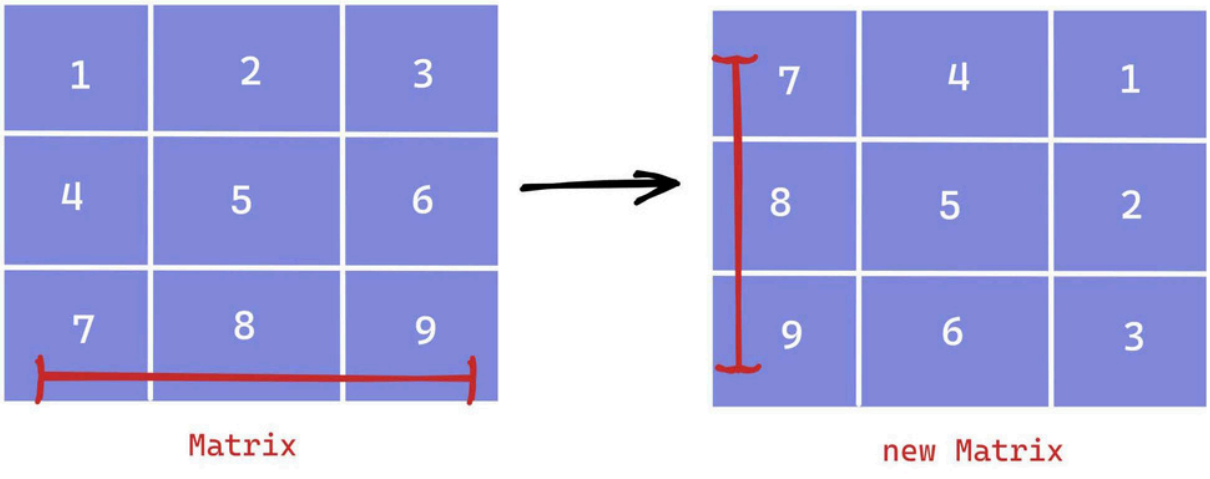
Example 1 :



Brute-Force Approach :

→ Create empty 2D matrix of size nxn , take last row from the given matrix and put it into the first column of new Matrix and so on.

```
ans[i][j] = matrix[(n-1) - j][i];
```

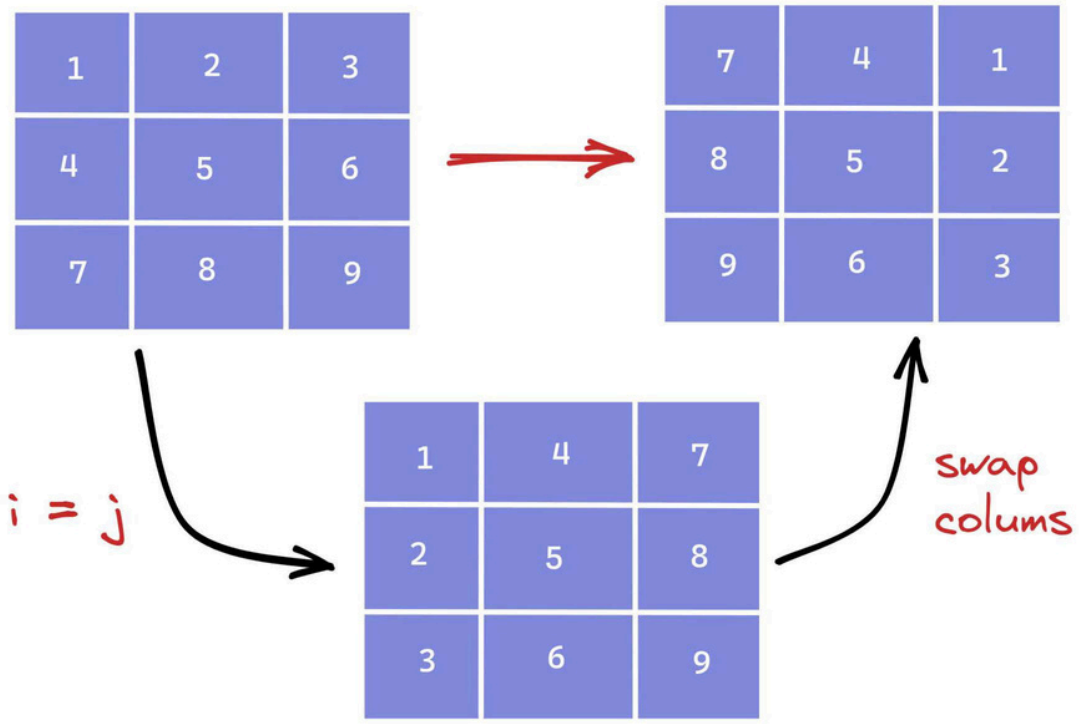


Code:-

```java
class Solution {
    public void rotate(int[][] matrix) {

        int[][] ans = new int[matrix.length][matrix[0].length];
        int n = matrix.length;

        for(int i = 0;i<n;i++){
            for(int j = 0;j<n;j++){

                ans[i][j] = matrix[(n-1) - j][i];

            }
        }
    }
}
```

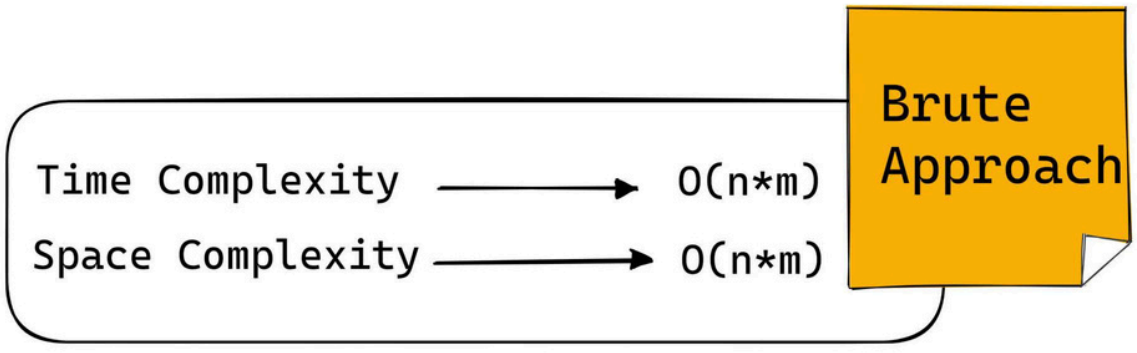
Optimal Approach

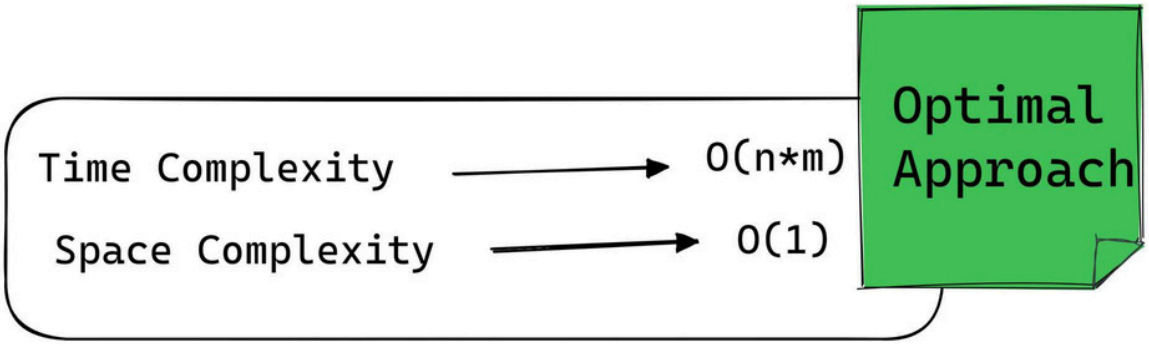
- Take transpose of given matrix (i = j).
- After that, swap first column with last column till matrix.length/2.



Code:-

```java
public void rotate(int[][] matrix) {

        for(int i = 0;i<matrix.length;i++){
            for(int j = i+1;j<matrix[0].length;j++){

                int temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }

        for(int i = 0;i<matrix.length;i++){
            for(int j = 0;j<matrix[0].length/2;j++){

                int temp = matrix[i][j];
                matrix[i][j] = matrix[i][n-j-1];
                matrix[i][n-j-1] = temp;
            }
        }
    }
```





Code:-

```java
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {

        List<Integer> list =new ArrayList<>();
        int left = 0;
        int right = matrix[0].length-1;
        int up = 0;
        int down = matrix.length-1;

        while(left<=right && up<=down){

            for(int col = left;col<=right;col++){
                list.add(matrix[left][col]);
            }
            up++;

            for(int row = up;row<=down;row++){
                 list.add(matrix[row][right]);
            }
            right--;

            for(int col=right;up<=down && col>=left;col--){
                list.add(matrix[down][col]);
            }
            down--;

            for(int row = down ; left<=right && row>=up;row--){
              list.add(matrix[row][left]);
            }
            left++;
        }
        return list;
    }
}
```

Time Complexity → O(m*n)

Space Complexity → O(1)

Santosh Kumar Mishra

The Art Of Data Structures and Algorithms

Visual Aids and Detailed Dry Runs for Easy Understanding

The Art of Structures Algorithms

ate Quick Reference Guide

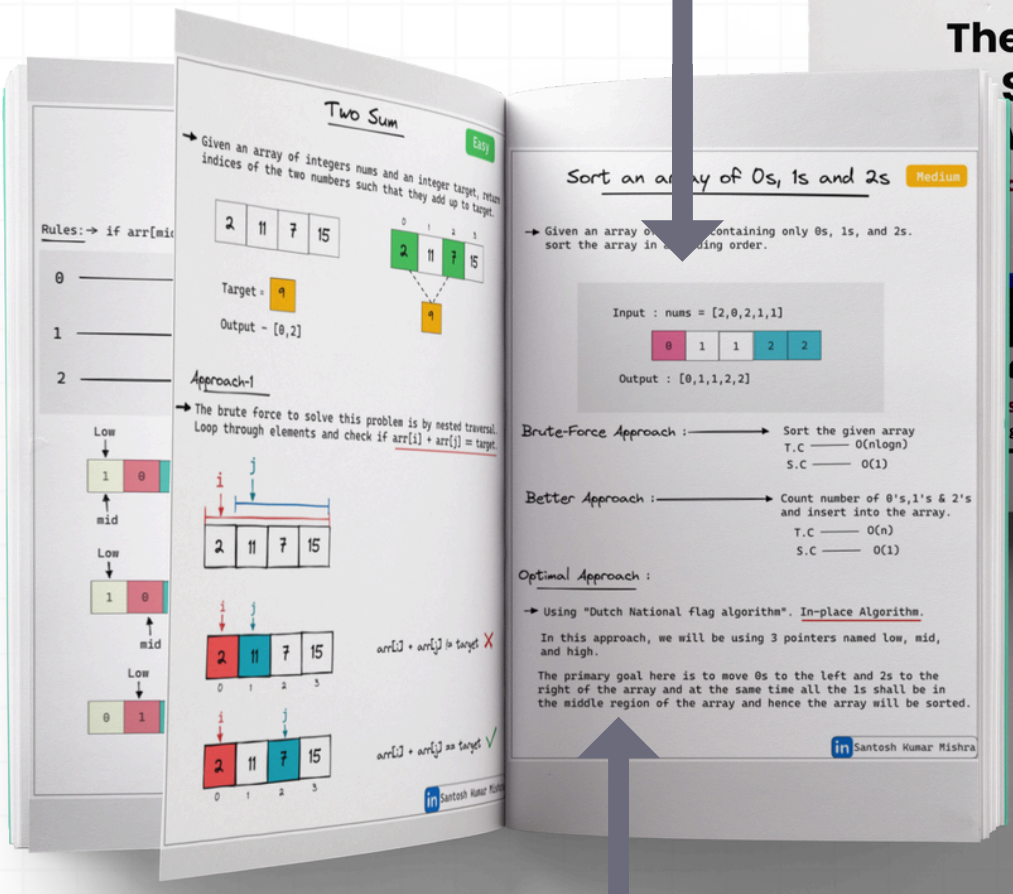
₹399 ~~₹999~~

sh Kumar Mishra

gineer at Microsoft, Author

@iamsantoshmishra

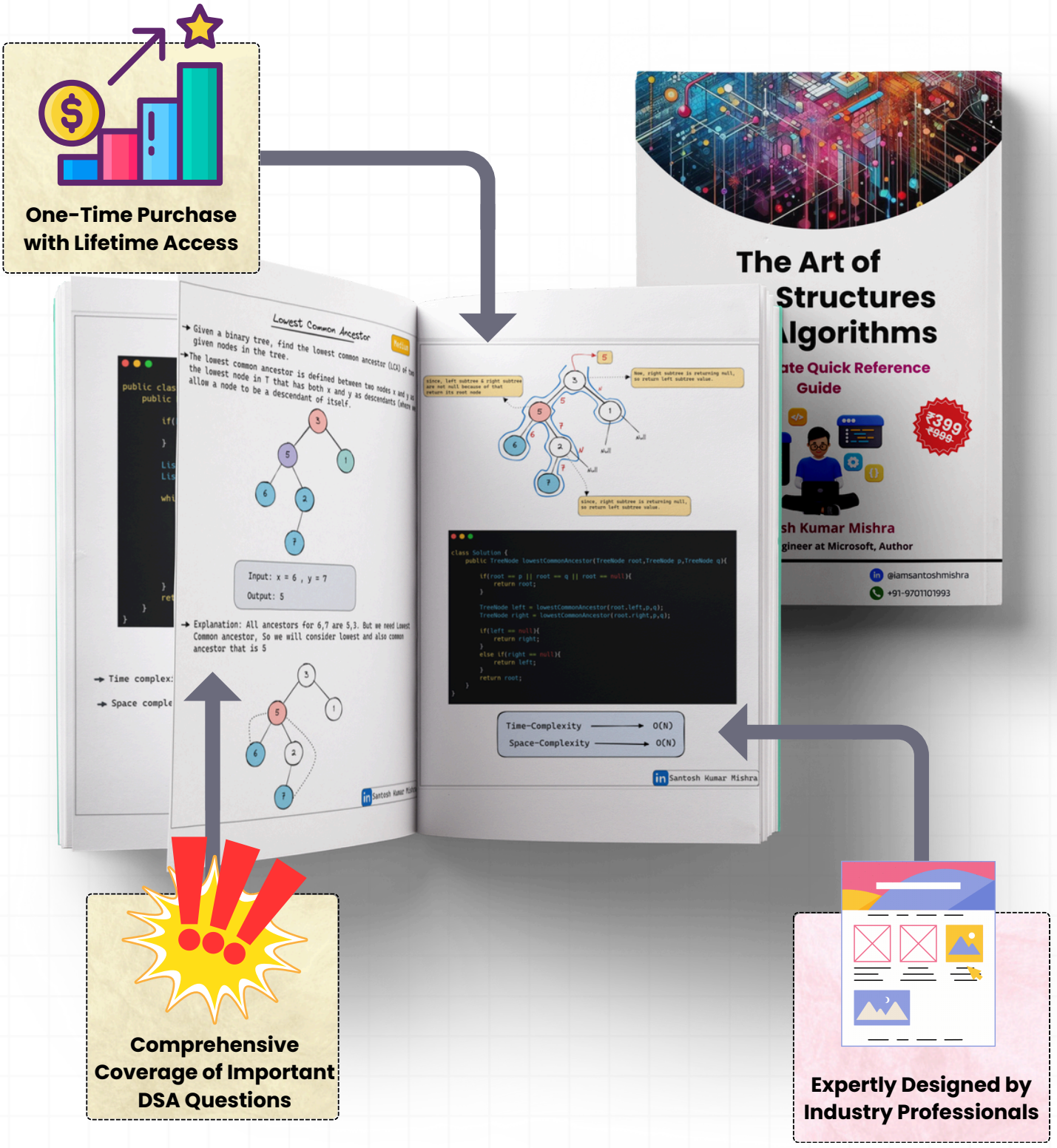
+91-9701101993



Questions Explained from Brute Force to Optimized Solutions

Perfect for Beginners to Advanced Learners

The Art Of Data Structures and Algorithms



The Art of Data Structures and Algorithms
Readers' Reviews and Insights

Prem Kumar
Software Engineer

I've never seen a book that covers DSA questions so thoroughly! Santosh breaks down each problem from the basic brute-force solution to the most optimized version, using proper diagrams and dry runs. It's a fantastic resource if you struggle with understanding complex algorithms.

Saumya Awasthi
Software Engineer

If you're preparing for interviews, this is the book you need. It contains all the key DSA problems explained with easy-to-follow graphics. The step-by-step approach, from brute force to optimized solutions, along with well-done dry runs, makes difficult concepts very digestible.

Archy Gupta
Software Engineer

Thank you, Santosh, for sending me a copy of this book. It's truly amazing! The book contains all the important DSA questions, and each one is explained from brute-force to optimized solutions with very clear dry runs and diagrams. It's the perfect resource for mastering DSA concepts and acing coding interviews.

The Art of Data Structures and Algorithms
Readers' Reviews and Insights

Aishwarya Tripathi
Software Engineer

"This book was a game-changer for my Amazon interview preparation. The collection of important DSA problems and their step-by-step solutions from brute-force to optimized helped me build a solid understanding. The graphical dry runs made complex concepts easy to follow. I highly recommend this book if you're aiming for top tech companies!

Parth Chaturvedi
Software Engineer

"The structured explanations in this book played a huge role in my Amazon interview success. The dry runs and graphical representations provided a clear understanding of complex problems, and the progression from brute-force to optimal solutions was exactly what I needed. This book is a must-read for serious interview prep.

Namrata Tiwari
Software Engineer

"This book's strength lies in its ability to break down complex problems into simple, understandable steps. The questions are explained starting from brute-force solutions and are then optimized with clear, graphical dry runs. It's an essential guide for interview prep and mastering DSA.

And Many More Success Stories from Satisfied Readers!

The Art of Data Structures and Algorithms

The Ultimate Quick Reference Guide

BUY NOW

Buy From Gumroad **Buy From Topmate**