**121. Best Time to Buy and Sell Stock**

You are given an array price 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.

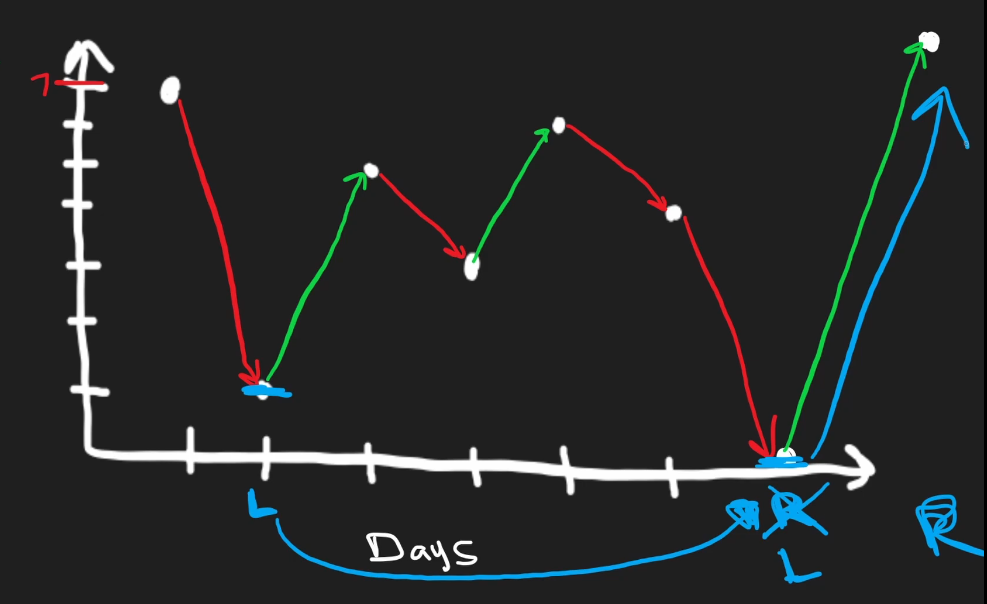
**Input:** prices = [7,1,5,3,6,4]

**Output:** 5

**Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

**Sol:**



Take two pointer i and j. start i from 0 and j from 1 index.

Now iterate the price.

If prices[i] < prices[j]….it means we got the profit of prices[j] – prices[i], find max of maxProfit and current profit.

Else …. I = j.. and increment the j counter.

Return max profit.

**1752. Check if Array Is Sorted and Rotated**

Given an array nums, return true*if the array was originally sorted in non-decreasing order, then rotated****some****number of positions (including zero)*. Otherwise, return false.

There may be **duplicates** in the original array.

**Note:** An array A rotated by x positions results in an array B of the same length such that A[i] == B[(i+x) % A.length], where % is the modulo operation.

**Input:** nums = [3,4,5,1,2]

**Output:** true

**Explanation:** [1,2,3,4,5] is the original sorted array.

You can rotate the array by x = 3 positions to begin on the element of value 3: [3,4,5,1,2].

**Sol:**

Here in the example, we can see that if array is sorted and rotated properly then we can have only point where arr[I] > arr[i+1].

So we will iterate the whole array and find out the count if arr[i] > arr[i+1]

If count > 1 then return false else return true

**2006. Count Number of Pairs With Absolute Difference K**

Given an integer array nums and an integer k, return *the number of pairs* (i, j) *where* i < j *such that* |nums[i] - nums[j]| == k.

The value of |x| is defined as:

* x if x >= 0.
* -x if x < 0.
* **Input:** nums = [1,2,2,1], k = 1
* **Output:** 4
* **Explanation:** The pairs with an absolute difference of 1 are:
* - [**1**,**2**,2,1]
* - [**1**,2,**2**,1]
* - [1,**2**,2,**1**]
* - [1,2,**2**,**1**]

**Sol:**

Take a map and put the frequency of each char in the map. Key will be num and its count will be value

Nums[i] - nums[j] = k or nums[i] – k = nums[j]

Nums[j] – nums[i] = k or nums[i] + k = nums[j]

Iterate the nums array

If map contains nums[i] – k then add the count of nums[i] – k into result

If map contains nums[j] – k then add the count of nums[j]-k into result

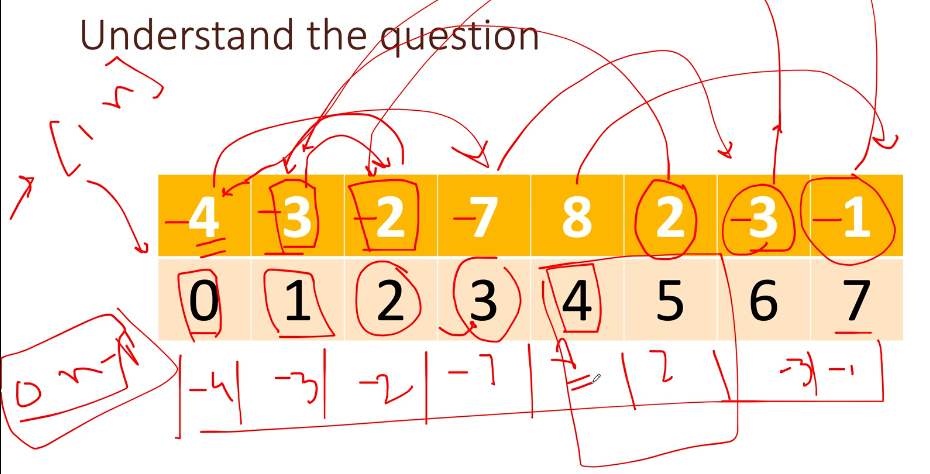
**448. Find All Numbers Disappeared in an Array**

Given an array nums of n integers where nums[i] is in the range [1, n], return *an array of all the integers in the range* [1, n] *that do not appear in* nums.

**Input:** nums = [4,3,2,7,8,2,3,1]

**Output:** [5,6]

**Sol:**



Iterate the nums and get the index by doing nums[i] – 1 and then go to that index and mark it -ve. After coming out of the loop we can see that only index 4, 5 are left which value is not -ve and its position will be 5,6 respectively.

**674. Longest Continuous Increasing Subsequence**

Given an unsorted array of integers nums, return *the length of the longest****continuous increasing subsequence****(i.e. subarray)*. The subsequence must be **strictly** increasing.

A **continuous increasing subsequence** is defined by two indices l and r (l < r) such that it is [nums[l], nums[l + 1], ..., nums[r - 1], nums[r]] and for each l <= i < r, nums[i] < nums[i + 1].

**Input:** nums = [1,3,5,4,7]

**Output:** 3

**Explanation:** The longest continuous increasing subsequence is [1,3,5] with length 3.

Even though [1,3,5,7] is an increasing subsequence, it is not continuous as elements 5 and 7 are separated by element 4.

**Sol:**

We can use variable size sliding window here.

We will check if arr[j-1] >= arr[j] it means we got the 1st decrementing element and we will assign I = j;

then will check the max of ans and window size j – I + 1 and increment the j++

**169. Majority Element**

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.

**Input:** nums = [3,2,3]

**Output:** 3

**Sol:**

Store frequency of the element into map.

Then if there is any element whose frequency is > n/2 return that element.

**643. Maximum Average Subarray I**

You are given an integer array nums consisting of n elements, and an integer k.

Find a contiguous subarray whose **length is equal to** k that has the maximum average value and return *this value*. Any answer with a calculation error less than 10-5 will be accepted.

**Input:** nums = [1,12,-5,-6,50,3], k = 4

**Output:** 12.75000

**Explanation:** Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75

**Sol:**

This is just same as find the maximum subarray of size k and once we get the result divide it by k to get the average.

**2016. Maximum Difference Between Increasing Elements**

Given a **0-indexed** integer array nums of size n, find the **maximum difference** between nums[i] and nums[j] (i.e., nums[j] - nums[i]), such that 0 <= i < j < n and nums[i] < nums[j].

Return *the****maximum difference****.*If no such i and j exists, return -1.

**Input:** nums = [7,**1**,**5**,4]

**Output:** 4

**Explanation:**

The maximum difference occurs with i = 1 and j = 2, nums[j] - nums[i] = 5 - 1 = 4.

Note that with i = 1 and j = 0, the difference nums[j] - nums[i] = 7 - 1 = 6, but i > j, so it is not valid.

**Sol:**

Iterate the array and find the minimum.

Once current element is greater than min then we need to find out maximum of maxDiff and nums[i] – min

**1200. Minimum Absolute Difference**

Given an array of **distinct** integers arr, find all pairs of elements with the minimum absolute difference of any two elements.

Return a list of pairs in ascending order(with respect to pairs), each pair [a, b] follows

* a, b are from arr
* a < b
* b - a equals to the minimum absolute difference of any two elements in arr
* **Input:** arr = [4,2,1,3]
* **Output:** [[1,2],[2,3],[3,4]]
* **Explanation:** The minimum absolute difference is 1. List all pairs with difference equal to 1 in ascending order.

**Sol:**

Sort the array. And then iterate the array to find the min diff of nums[i+1] – nums[i]. once we get the min diff

Iterate the array again and find out all nums[i+1] – nums[i] == min and add the pairs in ans.

**283. Move Zeroes**

Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

**Note** that you must do this in-place without making a copy of the array.

**Input:** nums = [0,1,0,3,12]

**Output:** [1,3,12,0,0]

**Sol:**

Iterate the array and if nums[i] != 0 then nums[index++] = nums[i].

Once we come out of the loop then we will have all non zero value at starting of the array. Then put another loop starting fromi= index to arrays length and do nums[i] = 0

**1299. Replace Elements with Greatest Element on Right Side**

Given an array arr, replace every element in that array with the greatest element among the elements to its right, and replace the last element with -1.

After doing so, return the array.

**Input:** arr = [17,18,5,4,6,1]

**Output:** [18,6,6,6,1,-1]

**Sol:**

Assign arr[n-1] = -1 as last element of the array will have no greater element to right.

And assign max\_from\_right = arr[n-1]

Iterate the array from end. Starting point will be n-2

Take arr[i] into temp

Assign arr[i] = max\_from\_right

And then update the max\_from\_right if it is > temp

max\_from\_right = Math.*max*(max\_from\_right, temp);

**35. Search Insert Position**

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

**Input:** nums = [1,3,5,6], target = 5

**Output:** 2

**Sol:**

Do the binary search on array and return the low. That will be the position to insert the value.

**136. Single Number**

Given a **non-empty** array of integers nums, every element appears *twice* except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

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

**Output:** 1

**Sol:**

Take the 1st element as ans and then perform XOR on rest of the element with 1st element.

XOR : 1 – 1 : 0, 0 – 1 : 1, 1 – 0 : 0

**1636. Sort Array by Increasing Frequency**

Given an array of integers nums, sort the array in **increasing** order based on the frequency of the values. If multiple values have the same frequency, sort them in **decreasing** order.

Return the *sorted array*.

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

**Output:** [3,1,1,2,2,2]

**Explanation:** '3' has a frequency of 1, '1' has a frequency of 2, and '2' has a frequency of 3.

**Sol:**

Take a map and store the count of each number.

Take a priority queue and put comparator into it.

**new** PriorityQueue<>((a, b) -> map.get(a) == map.get(b) ? b - a : map.get(a) - map.get(b));

if frequency is same then just sort it in descending order. Else sort it based on frequency.

Add all the key of the map to queue.

Then poll one by one and take a count of that number from map.

Add the same polled number that many times into result array.

**905. Sort Array By Parity**

Given an integer array nums, move all the even integers at the beginning of the array followed by all the odd integers.

Return ***any array****that satisfies this condition*.

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

**Output:** [2,4,3,1]

**Explanation:** The outputs [4,2,3,1], [2,4,1,3], and [4,2,1,3] would also be accepted.

**Sol:**

Take 2 pointer I and j starting from 0.

Iterate the array and if arr[j] % 2 != 0 then just increment j counter. Else swap arr[i] and ar[j] and increment i++, j++

**922. Sort Array By Parity II**

Given an array of integers nums, half of the integers in nums are **odd**, and the other half are **even**.

Sort the array so that whenever nums[i] is odd, i is **odd**, and whenever nums[i] is even, i is **even**.

Return *any answer array that satisfies this condition*.

**Input:** nums = [4,2,5,7]

**Output:** [4,5,2,7]

**Explanation:** [4,7,2,5], [2,5,4,7], [2,7,4,5] would also have been accepted.

**Sol:**

Take 2 pointer I and j starting I from 0 and j from 1

Iterate the array till I < n && j < n

While(nums[i] %2 == 0)

I += 2;

While(nums[j] % 2 == 1)

J += 2;

Once control comes here it means I points to odd value index and j points to even value index

So we will swap arr[i]] and arr[j]

**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.

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Sol:**

Take a map.

Iterate the array and check key = target – nums[i] is present in map

If yes then add the ans[0] = map.get(key) and ans[1] = i

Else add the key as key and index as value.