Q1**.** Given an integer array nums of 2n integers, group these integers into n pairs (a1, b1), (a2, b2),..., (an, bn) such that the sum of min(ai, bi) for all i is maximized. Return the maximized sum.

Example 1:

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

Output: 4

Explanation:All possible pairings (ignoring the ordering of elements) are:

1. (1, 4), (2, 3) -> min(1, 4) + min(2, 3) = 1 + 2 = 3

2. (1, 3), (2, 4) -> min(1, 3) + min(2, 4) = 1 + 2 = 3

3. (1, 2), (3, 4) -> min(1, 2) + min(3, 4) = 1 + 3 = 4

So the maximum possible sum is 4

**Solution**

Brutforce approach

2

3

4

1

Min(1,4)=1, Min(4,3)=3, Min(3,2)=2,

Min(1,2)=1, Min(1,3)=1,Min(4,2)=2

Sum=Min(1,4)+Min(4,3)=1+3=4 Sum=Min(3,2)+Min(4,2)=2+2=4

Min(1,2)+Min(1,3)=1+1=2

Maximum possible sum=4

n-1+n-2+…………………+1=n(n+1)/2=(n^2+n)/2=n^2/2+n/2

O(n^2)

**2nd solution**

1 4 3 2

Sorting:- 1 2 3 4

Return max sum

**3rd solution**

**2 sum problem**

**1 4 3 2 2 pointer approch**

**L L L R L**

R

L

R

**Min=(1,2)=1 Min(4,2)=2 sum=Min(1,2)+Min(4,2)=1+2=3**

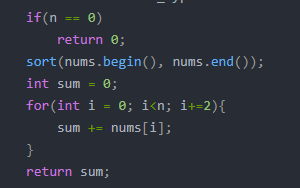
**Min(3,2)=2 Min(2,1)=1 sum=Min(3,2)+Min(2,1)=2+1=3**

**Min(1,3)=1 Min(3,4)=3 sum=Min(1,3)+Min(3,4)=1+3=4**

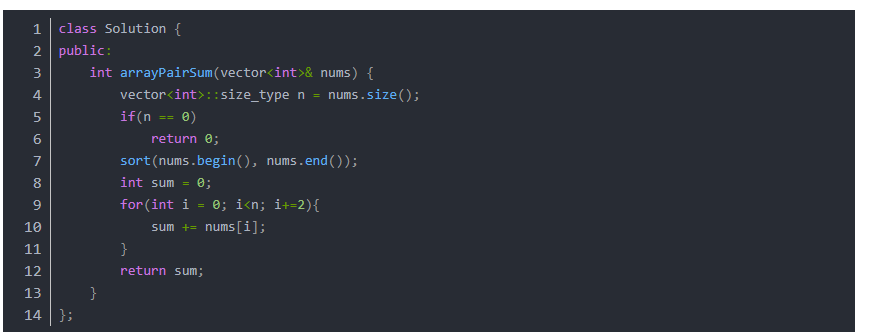
**Maximum possible sum=4**

**Algorithm:-**

* **Sort the input array**

****

**Java Code**

****

Python Solution

class Solution:

def arrayPairSum(self, nums: List[int]) -> int:

nums.sort()

r=0

for i in range(0,len(nums),2):

r+=min(nums[i],nums[i+1])

return(r)

Q2.

Alice has n candies, where the ith candy is of type candyType[i]. Alice noticed that she started to gain weight, so she visited a doctor.

The doctor advised Alice to only eat n / 2 of the candies she has (n is always even). Alice likes her candies very much, and she wants to eat the maximum number of different types of candies while still following the doctor's advice.

Given the integer array candyType of length n, return the maximum number of different types of candies she can eat if she only eats n / 2 of them.

Example 1:

Input: candyType = [1,1,2,2,3,3]

Output: 3

Explanation: Alice can only eat 6 / 2 = 3 candies. Since there are only 3 types, she can eat one of each type.

**Solution**

#### Idea:

In order to solve this problem, we need to identify how many unique types of candy there are. The easiest method to find unique values is with a **set**. If we convert our input array of candy types (**C**) to a set, then it will only contain unique values, and thus the size of the set will represent the number of different candy types.

The only other thing to remember is that we're constrained to at most **C.length / 2** pieces, per the instructions, so we need to use a **min()** boundary before we **return** our answer.

#### Implementation:

Java alone does not have an easy **set** constructor from an **int array**. Any such solution would have to invole boxing the primitive **int**s into **Integer**s before converting to a **HashSet**, so it's easier just to build the **HashSet** iteratively via a **for loop**.

#### Javascript Code:

const distributeCandies = C => Math.min((new Set(C)).size, C.length / 2)

#### Python Code:

class Solution:

def distributeCandies(self, C: List[int]) -> int:

return min(len(set(C)), len(C) // 2)

#### Java Code:

class Solution {

public int distributeCandies(int[] C) {

Set<Integer> cset = new HashSet<>();

for (int c : C) cset.add(c)

return Math.min(cset.size(), C.length / 2);

}

}

Q3. We define a harmonious array as an array where the difference between its maximum value and its minimum value is exactly 1.

Given an integer array nums, return the length of its longest harmonious subsequence among all its possible subsequences.

A subsequence of an array is a sequence that can be derived from the array by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: nums = [1,3,2,2,5,2,3,7]

Output: 5

Explanation: The longest harmonious subsequence is [3,2,2,2,3].

Ans:- class Solution {

public int findLHS(int[] nums) {

Arrays.sort(nums);

int prev = Integer.MIN\_VALUE;

int prevN = 0;

int curr = nums[0];

int currN = 0;

int max = 0;

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

if (nums[i] != curr) {

if (prev+1 == curr)

max = Math.max(prevN + currN, max);

prev = curr;

prevN = currN;

curr = nums[i];

currN = 1;

} else {

currN++;

}

}

if (prev+1 == curr)

max = Math.max(prevN + currN, max);

return max;

}

}

Algorithm

In this approach, we make use of a hashmap mapmap

which stores the number of times an element occurs in the array along with the element's value in the form (num, count\_num),

where num refers to an element in the array and count\_num refers to the number of times this num occurs in the numsnums array.

We traverse over the nums array and fill this map once.

After this, we traverse over the keys of the map created.

For every key of the map considered, say key, we find out if the map contains the key+1.

Such an element is found, since only such elements can be counted for the harmonic subsequence

if key is considered as one of the element of the harmonic subsequence.

We need not care about key−1, because if key is present in the harmonic subsequence,

at one time either key+1 or key−1 only could be included in the harmonic subsequence.

The case of key−1 being in the harmonic subsequence will automatically be considered, when key−1 is encountered as the current key.

Now, whenver we find that key+1 exists in the keys of mapmap,

we determine the count of the current harmonic subsequence as count\_{key} + count\_{key+1},

where count\_​i refers to the value corresponding to the key\_i in mapmap, which reprents the number of times i occurs in the array nums.

Complexity Analysis

Time complexity : O(n). One loop is required to fill map and one for traversing the map.

Space complexity : O(n). In worst case map size grows upto size n.

/\*

We define a harmonious array is an array where the difference between its maximum value and its minimum value is exactly 1.

Now, given an integer array, you need to find the length of its longest harmonious subsequence among all its possible subsequences.

Example 1:

Input: [1,3,2,2,5,2,3,7]

Output: 5

Explanation: The longest harmonious subsequence is [3,2,2,2,3].

Note: The length of the input array will not exceed 20,000.

\*/

class Solution {

public int findLHS(int[] nums) {

int rst = 0;

Map<Integer, Integer> map = new HashMap<>();

// put the number and its occurence in the map

for (int num : nums) {

map.put(num, map.getOrDefault(num, 0) + 1);

}

// Traverse the map to get the longest harmonious subsequence

for (int key : map.keySet()) {

if (map.containsKey(key + 1)) {

rst = Math.max(rst, map.get(key) + map.get(key + 1));

}

}

return rst;

}

}

Question 4

You have a long flowerbed in which some of the plots are planted, and some are not.However, flowers cannot be planted in adjacent plots.Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return true if n new flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule and false otherwise.

Example 1:

Input: flowerbed = [1,0,0,0,1], n = 1

Output: true

Ans:- **Note:**

* The input array won't violate no-adjacent-flowers rule.
* The input array size is in the range of [1, 20000].
* n is a non-negative integer which won't exceed the input array size.

**Solution**

Greedy Solution: O(n) time, O(1) space

public class Solution {

public boolean canPlaceFlowers(int[] flowerbed, int n) {

if (flowerbed == null || n > flowerbed.length / 2 + 1) return false;

int count = 0;

for (int i = 0; i < flowerbed.length && count < n; i++) {

if (flowerbed[i] == 0) {

int prev = i == 0 ? 0 : flowerbed[i - 1];

int next = i == flowerbed.length - 1 ? 0 : flowerbed[i + 1];

if (prev == 0 && next == 0) {

flowerbed[i] = 1;

count++;

}

}

}

return count == n;

}

}

**Analysis**

This solution literally checks each position in flowerbed and increase count if there is available place  
Therefore, we just need to return count == 0 at the end  
To check if position i is available, we should see its previous i-1 and next i+1 in given flowerbed  
If available, we first set flowerbed[i] = 1 then increment our count  
To terminate the loop earlier, we add count < n inside for loop heading

Let’s consider a very simple case where we have a bunch of 0s ending with 1.

Example:

01

001

0001

00001

We can plant only in the l = arr.size() – 2 zeros. A total number of plants that can be planted in such a section = Math.ceil(l/2D) Now we can imagine the whole array composed of such partitions.

Example

[1,0,0,0,1]

[1,0] [0,0,1]

0001

00001

We can plant only in the l = arr.size() – 2 zeros. A total number of plants that can be planted in such a section = Math.ceil(l/2D) Now we can imagine the whole array composed of such partitions.

Example

[1,0,0,0,1]

[1,0] [0,0,1]

**Complexity Analysis for Can Place Flowers Solution**

[**Time Complexity**](https://en.wikipedia.org/wiki/Time_complexity)**:** O(n) as we are just iterating the array once.

[**Space Complexity**](https://en.wikipedia.org/wiki/Space_complexity)**:** O(1) as we are not using any extra space.

Q5

Given an integer array nums, find three numbers whose product is maximum and return the maximum product.

Example 1:

Input: nums = [1,2,3]

Output: 6

Ans:- class Solution {

public int maximumProduct(int[] nums) {

int n=nums.length;

Arrays.sort(nums);

return Math.max(nums[0]\*nums[1]\*nums[n-1],nums[n-1]\*nums[n-2]\*nums[n-3]);

// return max\_product;

}

}

Approach :

Scan the array and compute Maximum, second maximum and third maximum element present in the array.

Scan the array and compute Minimum and second minimum element present in the array.

Return the maximum of product of Maximum, second maximum and third maximum and product of Minimum, second minimum and Maximum element.

Complexity :

Time complexity : O(n)

Space complexity : O(1)

Note:

Step 1 and Step 2 can be done in a single traversal

of the array.

Code (Explained in Comments)

class Solution

{

public int maximumProduct(int[] nums)

{

// Initialize Maximum, second maximum

// and third maximum element

int maxA=Integer.MIN\_VALUE;

int maxB=Integer.MIN\_VALUE;

int maxC=Integer.MIN\_VALUE;

// Initialize Minimum and

// second minimum element

int minA=Integer.MAX\_VALUE;

int minB=Integer.MAX\_VALUE;

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

{

// Update Maximum, second maximum

// and third maximum element

if(nums[i]>maxA)

{

maxC=maxB;

maxB=maxA;

maxA=nums[i];

}

// Update second maximum and

// third maximum element

else if(nums[i]>maxB)

{

maxC=maxB;

maxB=nums[i];

}

// Update third maximum element

else if(nums[i]>maxC)

{

maxC=nums[i];

}

// Update Minimum and second

// minimum element

if(nums[i]<minA)

{

minB=minA;

minA=nums[i];

}

// Update second minimum element

else if(nums[i]<minB)

{

minB=nums[i];

}

}

return Math.max(maxA\*maxB\*maxC,minA\*minB\*maxA);

}

}

Q 6

Given an array of integers nums which is sorted in ascending order, and an integer target,

write a function to search target in nums. If target exists, then return its index. Otherwise,

return -1.You must write an algorithm with O(log n) runtime complexity.

Input: nums = [-1,0,3,5,9,12], target = 9

Output: 4

Explanation: 9 exists in nums and its index is 4

Ans:-

|  |
| --- |
| class Solution { |
|  | public int search(int[] nums, int target) { |
|  | int low = 0, high = nums.length -1, mid; |
|  | while(low <= high){ |
|  | mid = (low + high)/2; |
|  | if (nums[mid] == target) |
|  | return mid; |
|  | if(target < nums[mid]){ |
|  | high = mid-1; |
|  | }else |
|  | low = mid+1; |
|  | } |
|  | return -1; |
|  | } |
|  | } |

**Solution:**

Binary search is an efficient algorithm with O(log n) runtime complexity used for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half.

Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

class Solution {  
 public int search(int[] nums, int target) {  
 int n = nums.length;  
 int l = 0, h = n-1;  
 int p = 0;  
 while(l<=h){  
 p = l+(h-l)/2;  
 if(nums[p]==target)  
 return p;  
 if(target<nums[p])  
 h = p-1;  
 else  
 l = p+1;  
 }  
 return -1;  
 }  
}

Question 7

An array is monotonic if it is either monotone increasing or monotone decreasing.An array nums is monotone increasing if for all i <= j, nums[i] <= nums[j]. An array nums is monotone decreasing if for all i <= j, nums[i] >= nums[j].Given an integer array nums, return true if the given array is monotonic, or false otherwise.

Example 1:

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

Output: true

Ans:- Approach

This code is for a method called isMonotonic that checks whether an array of integers nums is monotonic. A monotonic array is one that is either entirely non-increasing or non-decreasing. Let's break down the code step by step:

Initialize two boolean variables increasing and decreasing to false. These variables will be used to track whether the array is increasing or decreasing.

Start a loop that iterates over the elements of the array nums, starting from the second element (i = 1) and comparing it with the previous element.

Inside the loop, check if the current element nums[i] is greater than the previous element nums[i-1]. If it is, it means the array is increasing at this point.

If the current element is not greater than the previous element, check if it is less than the previous element. If it is, it means the array is decreasing at this point.

After examining all the elements in the array, the boolean variables increasing and decreasing will indicate whether the array is increasing, decreasing, or neither.

Calculate the final result by checking if both increasing and decreasing are true. If they are, it means the array has both increasing and decreasing elements, which makes it not monotonic. In this case, assign false to the result variable. Otherwise, assign true to result.

Return the result variable, which indicates whether the array nums is monotonic or not.

In summary, the code checks each pair of adjacent elements in the array to determine whether the array is increasing or decreasing at any point. If there are elements that cause the array to both increase and decrease, the array is not monotonic. Otherwise, the array is monotonic.

Code

class Solution {

public boolean isMonotonic(int[] nums) {

boolean increasing = false, decreasing = false;

for (int i=1; i< nums.length; i++){

if (nums[i]>nums[i-1]){

increasing=true;

} else if (nums[i]<nums[i-1]) {

decreasing=true;

}

}

boolean result = (increasing && decreasing) ? false:true;

return result;

}

}

Q 8

You are given an integer array nums and an integer k.In one operation, you can choose any index i where 0 <= i < nums.length and change nums[i] to nums[i] + x where x is an integer from the range [-k, k]. You can apply this operation at most once for each index i.The score of nums is the difference between the maximum and minimum elements in nums.Return the minimum score of nums after applying the mentioned operation at most once for each index in it.

Example 1:

Input: nums = [1], k = 0

Output: 0

Explanation: The score is max(nums) - min(nums) = 1 - 1 = 0.

Ans:- O(n) | using Max and Min element in array

class Solution {

public int smallestRangeI(int[] nums, int k) {

int min = nums[0];

int max = nums[0];

// Find the maximum and minimum element of array

for(int i = 1; i < nums.length; ++i) {

min = Math.min(nums[i], min);

max = Math.max(nums[i], max);

}

// The answer can be either the distance between maximum element and minimum element after decreasing and increasing their value respectively by k or 0 if the can be made same by some X where -k <= X <= k

int ans = max - min - 2 \* k;

return ans > 0 ? ans : 0;

}

}

**Logic**

if min(nums) + k < max(nums) - k, then return max - min -(2\*k);  
if min(nums) + k >= max(nums) - k, then return 0

Complexity

Time complexity: O(n)

Space complexity: O(1)

Approach

The first step is to sort the array and we know that the first index will have the min value and the last index will have the max value.If the first condition is true then min value will never surpass max value so we can take the max value of k and add the k value to the min value and subract that k value from the max value and if the above condition is false then we will check which value should get subtracted from max value, that is why we are using this int fl=nums[nums.size()-1]-nums[0]+k line to find the k value that needs to be subracted from max value and with that we can calculate the difference between max and min value.

For example:

arr=[1,2] k=5

#1nd condition

2-5>=1+5

the above condition turns out to be false, therefore we will go in the else statement

#2rd condition

fl=2-(1+5)

diff=2-(-4)-(1+5)

So diff=0

Code

class Solution {

public:

int smallestRangeI(vector<int>& nums, int k) {

sort(nums.begin(),nums.end());

int diff;

if(nums[nums.size()-1]-k>=nums[0]+k){

diff=(nums[nums.size()-1]-k)-(nums[0]+k);

}

else{

int fl=nums[nums.size()-1]-nums[0]+k;

diff=nums[nums.size()-1]-fl-nums[0]+k;

}

return diff;

}

};