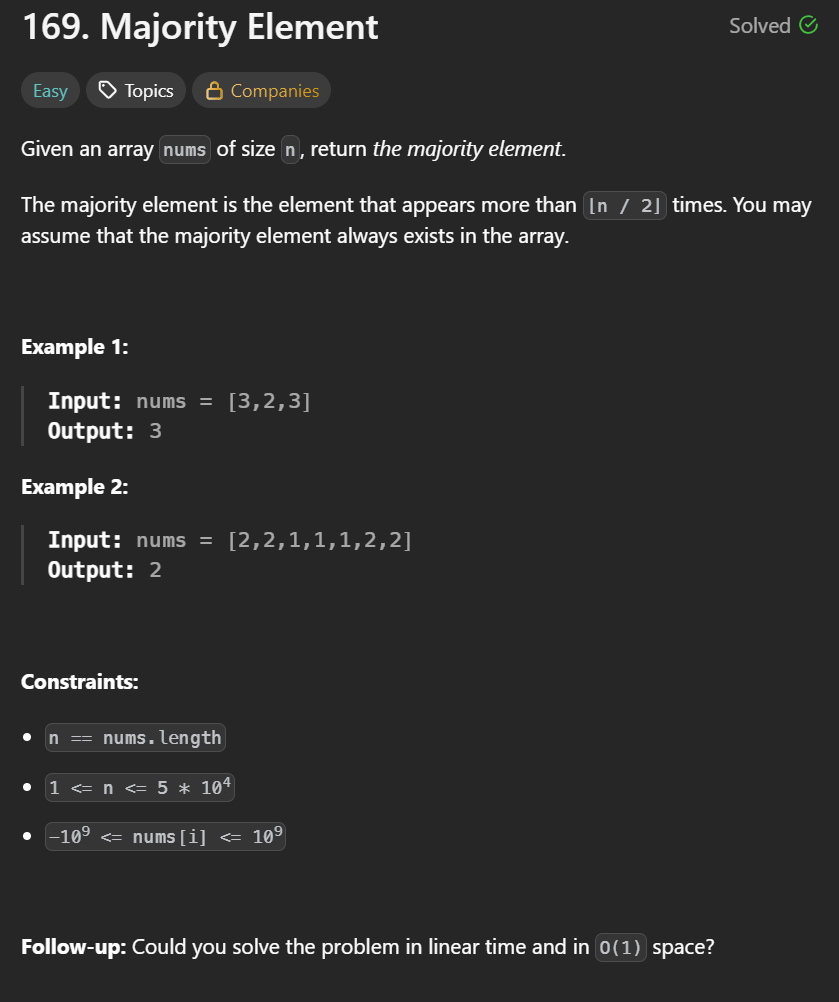
**Revision DSA**

**L.169 Majority Element;**

****

**Brute force approach: O(n2)**

Visiting every element in the array, and check how many times the element exists in the array, and keep a track using a **freq** variable. Then check freq with n/2.

code:

for(int val: nums){

for(int el: nums){

if(el == val){

freq++;

}

}

if(freq > n/2){

return freq;

}

**Optimized approach: O(nlogn):**

We first sort the array, the elements will be in continuous manner as it is a sorted array, we keep at track of elements using freq var. We will keep checking for an element whether it is greater than n/2;

If the element is changed and freq is not greater than n/2, we reset the freq to 0;

Code:  
**freq = 1; ans = nums[0];**

**for(int i=1; i<n; i++){**

**if(nums[i] == nums[i-1]{**

**Freq++;**

**}**

**else{**

**freq = 1;**

**ans = nums[i];**

**}**

**}**

**Moore’s approach:** **O(n)**

Initially we take two variables **freq =0** and **ans=0.**

**The idea is that the majority element in the array will always have a higher power(Count of freq). The freq of the majority element will always be greater than n/2.** Also when the freq becomes 0, we add the current nums[i] value into our ans variable. When the ans is equal to the current nums[i] value, we increment the freq, else if the ans and current value of nums is different, we decrement the freq variable.

**Code:**

class Solution {

public:

int majorityElement(vector<int>& nums) {

int freq = 0, ans = 0;

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

if(freq == 0){

ans = nums[i];

}

if(ans == nums[i]){

freq++;

}else{

freq--;

}

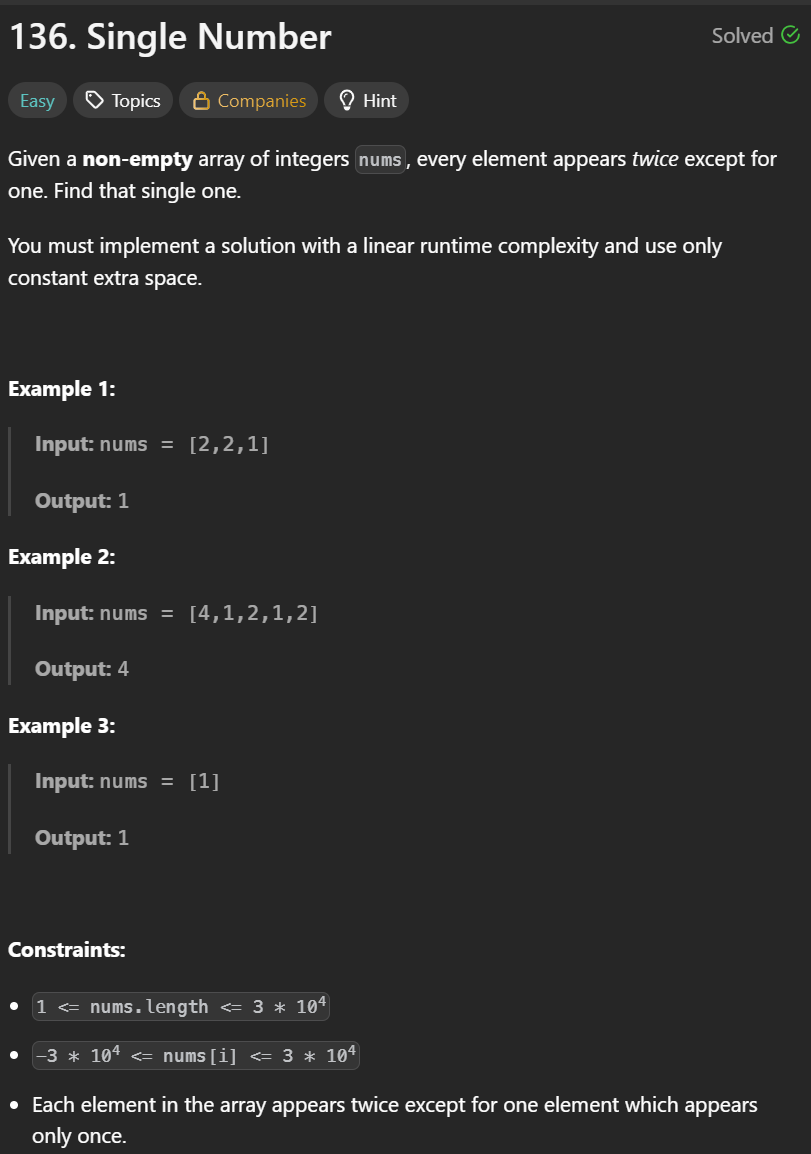
}

return ans;

}

};

**L.136 Single Number:**



Soln:

We use the concept of BIT manipulation, XOR operations. Here we have to find the single element in the array, every other element will have a duplicate.

Using XOR operations, we can find a single number in the array.

Note:

XOR operations (^):  
**0 ^ 0 = 0**

**1 ^ 1 = 0**

**0 ^ 1 = 1**

**1 ^ 0 = 1**

Similarly, if we XOR same element we get a 0:

n ^ n = 0

If we XOR any number with “0”, we get that number:

n ^ 0 = n

Time complexity: **O(n)**

Code:

class Solution {

public:

int singleNumber(vector<int>& nums) {

int ans=0;

for(int val: nums){

ans= ans^val;

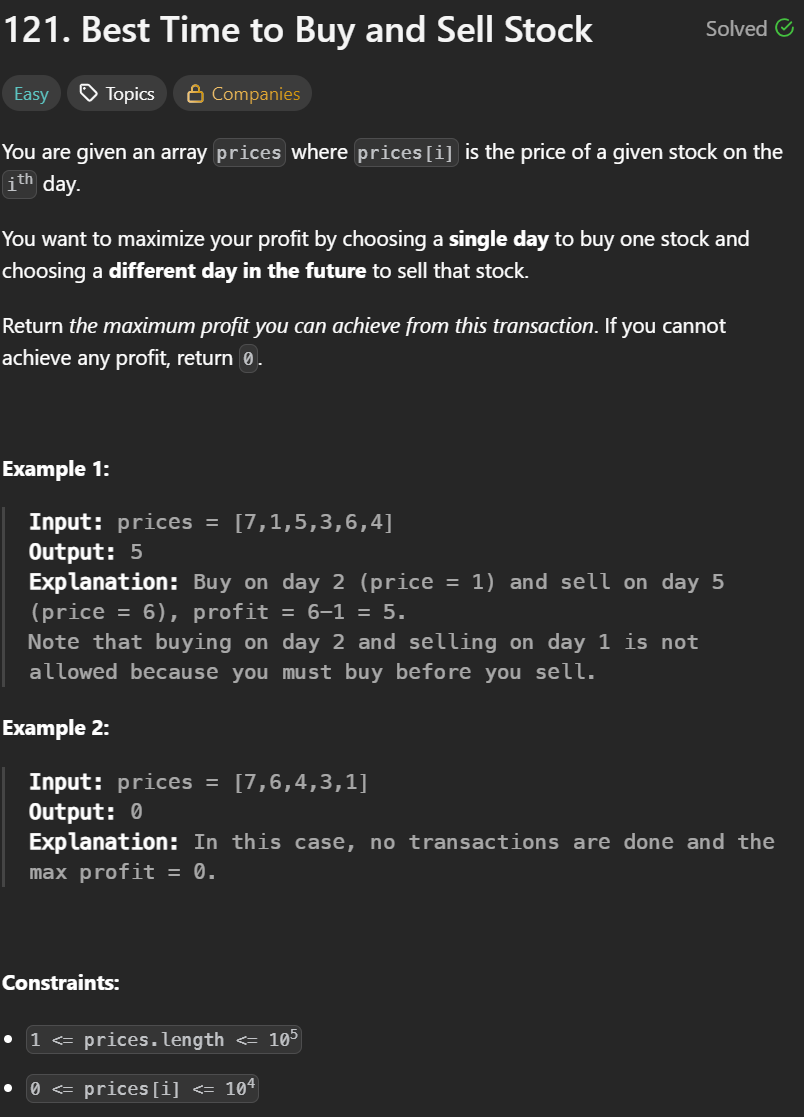
}

return ans;

}

};

**L.121. Best time to buy and sell stock: O(n)**



Soln: there is an array which contains the prices of each stock. The ith value of the array indicates the days. For eg. i = 1 in the array is the 1st day, i=2 in the array is the 2nd day.

We have to buy a stock first and then sell it. We have to find the maximum profit. Stocks cannot be bought and sold on the same day. If no profit we return -1.

In order to gain maximum profit, we have to find the minimum of buy and maximum of profit.

Code:

class Solution {

public:

int maxProfit(vector<int>& prices) {

int maxProfit=0, bestBuy = prices[0];

for(int i=1; i< prices.size(); i++){

if(prices[i]> bestBuy){

maxProfit = max(maxProfit, prices[i]-bestBuy);

}

bestBuy = min(bestBuy, prices[i]);

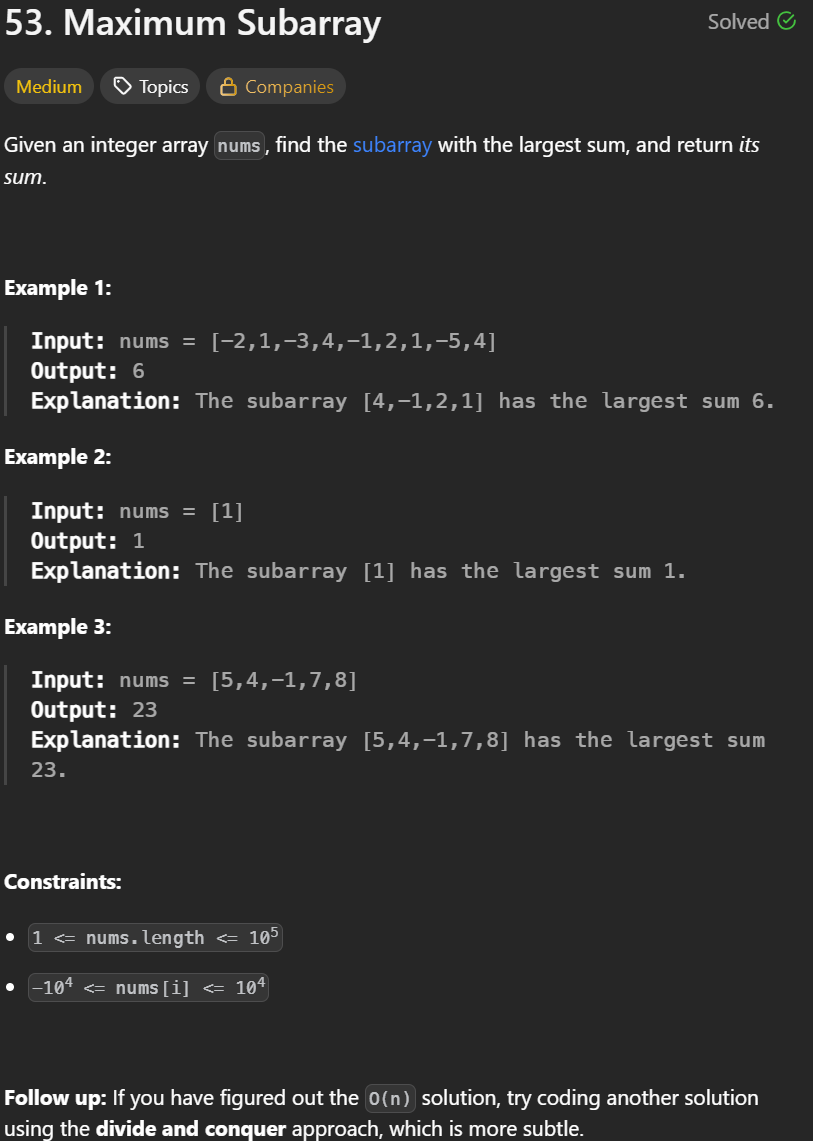
}

return maxProfit;

}

};

**L.52 Maximum Subarray (Kadane algorithm):**



Soln:

Big(+ve) + Big(+ve) = +ve

small(-ve) + Big(+ve) = +ve

small(+ve) + Big(-ve) = -ve

Take two vars CurrSum, MaxSum -> this will store our ans;  
Works such that if we add a big pos with any small whether positive or negative, we get a positive.  
Although if we get negative number in our currSum, then we reset the currentSum to 0;

I.e if our currSum becomes negative, we reset it to 0. Because subarray is a continuous entity, the negative value will only reduce the overall maxSUm, hence we reset the currentSum to 0;

Code:

class Solution {

public:

int maxSubArray(vector<int>& nums) {

int currSum = 0, maxSum= INT\_MIN;

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

currSum = currSum + nums[i];

maxSum = max(currSum, maxSum);

if(currSum < 0){

currSum =0;

}

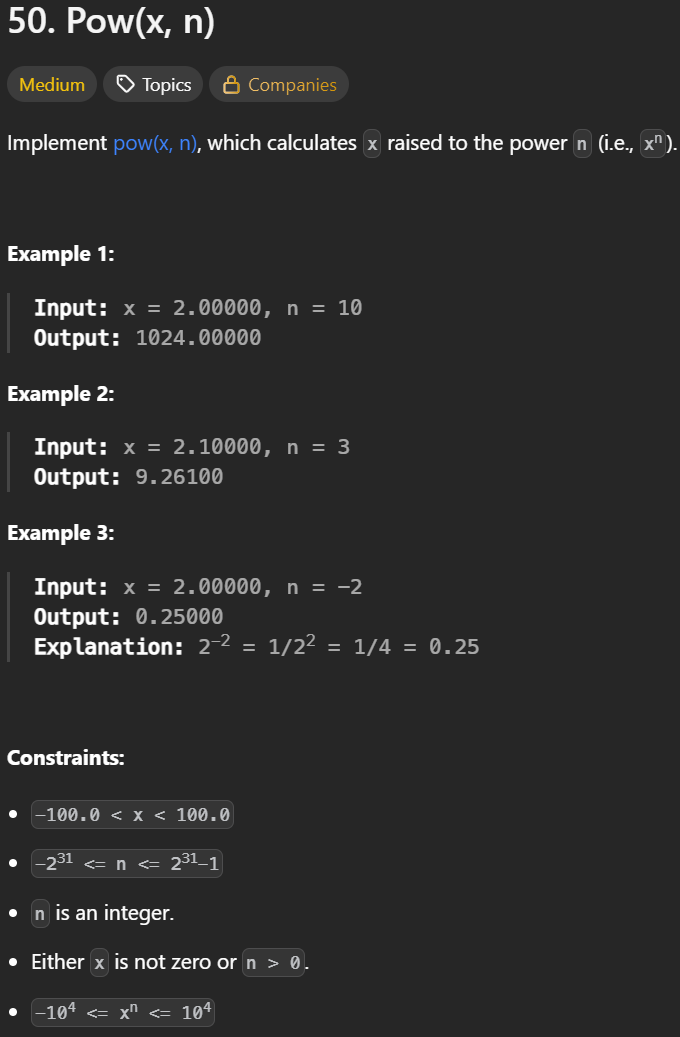
}

return maxSum;

}

};

L.50:



Calculating xn.

For eg: 35 -> 3 x 3 x 3 x 3 x 3

Here n = 5;

This logic has a time complexity of O(n), n can have power of 231.

But in time complexity, at most 108 operations can be performed, if it goes over that we get a TLE error. Since 231 is far greater than 108.

Hence we cannot use the linear approach.

n -> log2n + 1 binary digits exist

i.e in a decimal n, at most log2n + 1 binary digits exist.

For eg: n = 8 -> 1000 binary form

Log28+1 -> 3+1 = 4 hence 4 digits in binary.

If we apply a loop on a decimal form, here in case of 28, we apply loop 8 times. However if we apply loop on its binary form (1000), we apply loop 4 times.

For eg. We do not apply loop on its power decimal form. We apply on that power’s binary form.

Xn -> 35

Here we apply loop on 5’s binary form and not on 5 decimal form.

101

Here we apply loop on binary form where we will run loop 3 times.

This approach will give a Time complexity of **O(logn)**

A blackboard with math equations

AI-generated content may be incorrect.

We also consider the case where, if the value of n is negative, then we do reciprocal of x.

For eg. 3-5 then we do (1/3)5.

Hence if n<0 then x= 1/x and binForm = -binForm.

Code:

A screenshot of a computer program

AI-generated content may be incorrect.

L.11:

A screenshot of a graph

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Soln: Brute force approach, we take left boundary right boundary,

We compary every pair of left and right boundary, calculate the area using width = j-I, and

Height using min(arr[i]. arr[j]), and multiplying width x height. We will create a maxArea=0 var which will later calculate max of maxArea and area.

A blackboard with math formulas and graphs

AI-generated content may be incorrect.

This approach will give TLE error since max constraints is 105.

This approach has O(n2) hence in worst case it will go (105)2 -> 1010.   
only 108 operations are valid in these case.  
  
Hence we need an optimized approach,

We use a two pointer approach, where we take two vars l -> left and r -> right.

We will initialize l -> 0(start) and r -> n-1(end).

Now area = width x height

Width = r-l;

Height = min (height[i], height[j]);

Why we take min? because the area of container will be determined by the shorter height, due to container storing water in a rectangular form, the width (on x axis) and opposite width i.e the top surface of the water will be same. So the height smaller will determine the area.

So main logic here is if the height of left (l) is minimum then we do l++

If the height of right (r) is minimum then we do r--.

A blackboard with a graph and numbers

AI-generated content may be incorrect.

Code:  
A computer screen shot of a program code

AI-generated content may be incorrect.