Striver's SDE Sheet

Day 3: Arrays Part-III

Search a 2D Matrix

You are given an m x n integer matrix matrix with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true *if* target *is in* matrix *or* false *otherwise*.

You must write a solution in O(log(m * n)) time complexity.

Ans:

Observation and approach: (process based question due to its condition its also pattern based question)

In first go we have to find element in 2d matrix so brute force approach is to travel whole array and find element

Takes O(N*M) time

Second approach strikes in my mind apply binary search on each row and travel through m columns which take O(M*lon N)

And third idea strike in my mind is to take out first and last element of each row apply binary search over it , give a range then go back to that range row and apply binary search over it and get the answer

It takes O(log n +log m) -> O(log (n*m)) answer 😎

Or we can this 2 d array into a 1 d array and get our answer (striver approach)

Code:

```
int m=mat.size();
        int n=mat[0].size();
        int start=0;
        int end=m*n-1;
        while(start<=end){
            int mid=(start+end)>>1;
            int row = mid / n;
            int col = mid % n;
            if(mat[row][col]==target){
                return true;
            }else if(mat[row][col]>target){
                end=mid-1;
            }else{
                start=mid+1;
            }
        }
        return false;
```

some time when you play with time complexities , strikes some approaches like in this case log n means give some hint about binary search

Pow(x, n)

Implement pow(x, n), which calculates x raised to the power n (i.e., x^n).

Ans:

Code:

```
class Solution {
public:
    double myPow(double x, int n) {
         long long nn=n;
        if (nn<0){
            nn=-1*nn;
        }
        double ans =1.0;
        while (nn) {
    if (nn % 2!=0) {
      ans = ans * x;
      nn = nn - 1;
    } else {
      x = x * x;
      nn = nn / 2;
    }
  }
        if(n<0){
            return 1/ans;
        }
        else{
            return ans;
        }
    }
};
```

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.

Ans:

Observation and approach :

In first go brute force approach says that sort the array and give the element takes n logn time

Second approach is to count frequency of all element return the answer takes n time and n space

Third approach find over the internet: MOORE'S VOTING ALGORITHM

Think of this array as a collection of votes from the voters for different political parties.
Now as we know the party which has >50% votes can form the government.

So , our above question is analogous to this situation.
Now , if we are certain that one party has > 50% votes . Then , if :

- We increment a count variable every time we see the vote from the majority party and decrement it whenever a vote from some other party is occured , we can guarantee that ,
  count>0.

Using the above logic ,

1. Create a count=0 and a majority variable that stores the current majority element.
2. Traverse the array , and if count =0 , then store the current element as the majority element and increment the count.
3. Else , if the current element is equal to the current majority element , increment count , else decrement it.
4. At the end return the majority element.

This approach works because ,

1. We simply know that the majority element has a frequency greater than half of the total elements.
2. So , the value of count > 0.
3. But whenever it becomes 0 , it means that till now the majority element has either not occured , or if it has then the its frequency is equal to the sum of frequency of rest of the others. So , the next element will be the majority element till now . But eventually by the end , the final answer will always be the majority element of the array , as when the "count" becomes 0 for the final time , the next element will be the majority element.

Code:

```
class Solution {

public:

    int majorityElement(vector<int>& nums) {

        int majority,count=0;

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

        {

            if(count)
```

```
            { count+=(nums[i]==majority ? 1 : -1); }
            else
            { majority=nums[i]; count=1; }
        }
        return majority;
    }
};
```

note some time over the internet we find bunch of different solution and algorithm to solve a particular problem and I think that I have to explore this type of algo because as you saw on day 2 we have a direct question from merger sort logic so to knows more algorithm means to have more approaches (not always but in 80% cases yes)

So lets explore

Majority Element II

Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

Ans:

Observation and approach:

In first go brute force approach says that apply two loops and check for element if condition followed then push to answer vector and return it

Second approach is to use hash map takes o(n) and o(n) space

Third approach is to sort the array in log n and travel to find element with condition take n time

And approach that I found over the internet: Boyer–Moore Voting Algorithm.

Code:

```
vector<int> majorityElement(vector<int>& nums) {
        int num1 = INT_MIN, num2 = INT_MIN;
        int count1 = 0, count2 = 0;
        for(auto element : nums){
            if(num1 == element){
                count1++; }
            else if(num2 == element){
                count2++; }
            else if(count1 == 0){
                num1 = element;
                count1 = 1;
            }
            else if(count2 == 0){
                num2 = element;
                count2 = 1;
            }
            else{
                count1--;
                count2--;
            }
        }
        vector<int> output;
        int countMajority = nums.size()/3;
        count1 = 0, count2 = 0;
        for(auto element : nums){
            if(num1 == element){
                count1++;
            }
            if(num2 == element){
                count2++;
            }
```

```
        }
        if(count1 > countMajority){
            output.push_back(num1);
        }
        if(count2 > countMajority){
            output.push_back(num2);
        }
        return output;
    }
```

Reverse Pairs

Given an integer array nums, return *the number of reverse pairs in the array*.

A reverse pair is a pair (i, j) where:

- 0 <= i < j < nums.length and
- nums[i] > 2 * nums[j].

Ans:

Not able to figure out the answer 🤯

From leetcode:

Intuition

Having Solved Count Inversions Problem where we find the Pairs i<j and a[i]>a[j] we used Merge Sort to count them by the Logic that the element in first half in merging having greater value than any element in second half is going to add inversion pairs accordingly.
Having Said That We need to Find The Pairs with exact same Logic BUT This Time The Condition Switches to a[i]>(2*a[j]).
Thus We Need to Find The Very Last Index in Second Half where The Condition satisfies for any element in first half.

Approach

We Do Divide The Array into Subparts and Conquer Them Using Merge Sort and The Very Fact that Element in First Half has to be greater than twice of the element in second array.
So taking advantage that first and second halves are sorted, we apply Binary Search Instead of Linear To Optimize and avoid TLE.
While Merging itself we find the index for that required condition and apply math accordingly as to how many pairs will be added in the count.

Code:

```
int inversions=0;

    int find_index_where_just_greater(int left,int right,vector<int>&arr,int target)

    {

        while(left<=right)

        {

            int mid=(left+right)/2;

            long long x=arr[mid];

            x*=2;

            if(x>=target)

            {

                right=mid-1;

            }

            else

            {

                left=mid+1;

            }

        }

        return (left-1);
```

```
}
void merge(vector<int>&arr,int l,int m,int r)
{
    vector<int>Arr(r-l+1);
    int ptr1=l,ptr2=m+1,ptr3=0;
    while(ptr3<=(r-l))
    {
        if(ptr1==(m+1))
        {
            Arr[ptr3]=arr[ptr2];
            ptr2++;
            ptr3++;
            continue;
        }
        if(ptr2==(r+1))
        {
            int idx=find_index_where_just_greater(m+1,r,arr,arr[ptr1]);
            inversions+=idx-m;
            Arr[ptr3]=arr[ptr1];
            ptr1++;
            ptr3++;
            continue;
        }
        if(arr[ptr1]<=arr[ptr2])
        {
            int idx=find_index_where_just_greater(m+1,r,arr,arr[ptr1]);
            inversions+=idx-m;
            Arr[ptr3]=arr[ptr1];
            ptr1++;
            ptr3++;
        }
        else
        {
            Arr[ptr3]=arr[ptr2];
```

```
            ptr2++;
            ptr3++;
        }
    }
    for(int i=l;i<=r;i++)
    {
        arr[i]=Arr[i-l];
    }
    return;
}
void mergeSort(vector<int>&arr,int l,int r)
{
    if(l==r)
    {
        return;
    }
    int mid=(l+r)/2;
    mergeSort(arr,l,mid);
    mergeSort(arr,mid+1,r);
    merge(arr,l,mid,r);
    return;
}
int reversePairs(vector<int>&nums)
{
    mergeSort(nums,0,nums.size()-1);
    return inversions;
}
```

Unique Paths

There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to $2 * 10^9$

Ans:

Observation and approach :

In first go this question resemble to dp or recursion problem In which we have to find all possibility and for unique path just have to maintain a global set hence we get our answer

So brute force will be :

Code:

```
int uniquePaths(int m, int n, int i = 0, int j = 0) {
        if(i >= m || j >= n) return 0;
        if(i == m-1 && j == n-1) return 1;
        return uniquePaths(m, n, i+1, j) + uniquePaths(m, n, i, j+1);
    }
```

With DP :

```
int dp[101][101]{};
    int uniquePaths(int m, int n, int i = 0, int j = 0) {
        if(i >= m || j >= n) return 0;
        if(i == m-1 && j == n-1) return 1;
        if(dp[i][j]) return dp[i][j];
        return dp[i][j] = uniquePaths(m, n, i+1, j) + uniquePaths(m, n, i, j+1);
    }
```

with maths : (best solution)

```
 int uniquePaths(int m, int n) {
        long ans = 1;
        for(int i = m+n-2, j = 1; i >= max(m, n); i--, j++)
            ans = (ans * i) / j;
        return ans; }
```