Love Babbar's SDE Sheet

Approaches for

Array Section

With Time and Space Complexities.

By Bhargab Sarmah

215. Kth Largest Element in an Array

Given an integer array `nums` and an integer `k`, return *the* `k^th` *largest element in the array*. Note that it is the `k^th` largest element in the sorted order, not the `k^th` distinct element.

Example 1: **Input:** nums = [3,2,1,5,6,4], k = 2 **Output:** 5

Example 2: **Input:** nums = [3,2,3,1,2,4,5,5,6], k = 4 **Output:** 4

Approach 1: Sort the array and return arr[n-k] TC = O(nlogn) SC = O(1)

Approach 2: 1) use a min priority queue (min heap) 2)iterate over the array and insert in queue, after inserting check if the size of the queue is greater than k if yes pop the top element 3) return q.top()

TC = O(nlogk) SC = O(k)

```
int findKthLargest(vector<int>& nums, int k) {

    //min heap
    std::priority_queue<int,std::vector<int>,greater<int>>q;
    for(auto i:nums){
      q.push(i);
        if(q.size() >k){
            q.pop();
        }
    }
 } return q.top()
```

Approach 3: Using quick sort

75. Sort 0's 1's and 2's

Given an array of size N containing only 0s, 1s, and 2s; sort the array in ascending order without using sort function.

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

Approach: Using DNF sort

Steps: 1) take 3 pointers lo = 0, mid = 0 , hi = n-1

2) run a loop while mid<=hi

If we find 0 in middle swap arr[lo] and arr[mid] mid++,lo++

If we find 1 mid++

If 2 swap mid and hi hi--

TC = O(n) Sc = O(1)

Move all negative numbers to the beginning & positive numbers to the end

Input: nums = [12,-70,-2,1,-1,0] **Output:** [-70,-2,-1,12,1,0]

Approach:

Steps: 1) take 2 pointers i = 0, st = 0

2) run a loop while i<n

If arr[i] == –ve number swap arr[i] and arr[st] st++

TC = O(n) Sc = O(1)

189. Rotate Array

Given an array, rotate the array to the right by `k` steps, where `k` is non-negative.

Input: `nums = [1,2,3,4,5,6,7], k = 3` **Output:** `[5,6,7,1,2,3,4]`

Explanation:

```
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
```

Approach: Steps 1) reverse the array // [7,6,5,4,3,2,1] 2) reverse the 1st k elements // [5,6,7,4,3,2,1] 3) reverse the rest elements after k // `[5,6,7,1,2,3,4]`

TC = O(n) Sc = O(1)

53. Maximum Subarray (Kadane's Algo)

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

I/p: `nums = [-2,1,-3,4,-1,2,1,-5,4]` **O/p:** `6` **Expln:** `[4,-1,2,1] has the largest sum = 6.`

Approach1: `Use 2 for loops and find the largest subarray sum` **TC = O(n^2) Sc = O(1)**

Approach 2: Using Kadane's Algo

Steps: 1) Use two variables max_ans = 0 and curr_max = 0 curr_max gives us the subarray sum till that

2) if curr_max < 0 update curr_max = 0 index and update max_ans

TC = O(n) Sc = O(1)

910. Smallest Range II

You are given an integer array `nums` and an integer `k`. For each index `i` where `0 <= i < nums.length`, change `nums[i]` to be either `nums[i] + k` or `nums[i] - k`.The **score** of `nums` is the difference between the maximum and minimum elements in `nums`. Return *the minimum **score** of `nums` after changing the values at each index*.

Input: `nums = [1,3,6], k = 3` **Output:** `3`

Explanation: `Change nums to be [4, 6, 3]. The score is max(nums) - min(nums) = 6 - 3 = 3.`

Approach: 1) sort the arr and ans= arr[n-1] – arr[0]

2) the lowest value will be added by k and highest value will be subtracted by k

3) Now while adding k to lowest the 2nd lowest might become less thn the earlier lowest

4) similarly while subtracting k from highest the 2nd highest might get > earlier highest

5)iterate in the array and check if after adding k to lowest the ith num < lowest

If yes update minn = min(arr[0]+k,arr[i+1]-k)

Check for max now, maxx = max(arr[n-1]-k , arr[i]+k)

Then find ans = min(ans,maxx-minn)

TC = O(nlogn) Sc = O(1)

287. Find the Duplicate Number

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*.You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

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

Input: nums   = [2,2,2,2,2]            output : 2
```

Approach: Since the elements will be from [1,n] , therefore our answer lies within that range

Steps: 1) we will use binary search and find the number of elements in the i/p arr that are <= mid **st=1 en = n-1**

2) if cnt is <= mid than our answer lies in the 2nd half hence, **st = mid+1**

3) else **en = mid-1**

4) **return st**

TC = O(nlogn) Sc = O(1)

88. Merge Sorted Array

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.**Merge** `nums1` and `nums2` into a single array sorted in **non-decreasing order**. Do not use extra space store the values in nums1.

```
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
             Output: [1,2,2,3,5,6]

Explanation: The arrays we are merging are [1,2,3] and [2,5,6].

The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1
```

Solution: corner cases: 1) m=0 , nums1 = nums2 return;

2) n =0, return

Steps: 1)iterate from the end of nums1.

2)Check **if m>=0** and the values of nums1[m-1] and nums2[n-1] whichever is greater insert at the ith index and decrement either m or n accordingly. Check if n==0 break;

56. Merge Intervals

Given an array of `intervals` where `intervals[i] = [start_i, end_i]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]          Output:
[[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

```
Solution:
1)sort the vector according to the start time
2) initialize, st = v[0][0] , en = v[0][1]
3) iterate over the vector, if start of each element is <= en, update en = max(en ,
v[i][1])
                        Else   insert {st,en} in our ans vector and update st= v[i][0],
en = v[i][1]
4)after iterarting insert the {st,en} in ans vector , return ans;
```

31. Next Permutation

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

For example, for `arr = [1,2,3]`, the following are considered permutations of `arr`: `[1,2,3]`, `[1,3,2]`, `[3,1,2]`, `[2,3,1]`.

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of `arr = [1,2,3]` is `[1,3,2]`.
- Similarly, the next permutation of `arr = [2,3,1]` is `[3,1,2]`.
- While the next permutation of `arr = [3,2,1]` is `[1,2,3]` because `[3,2,1]` does not have a lexicographical larger rearrangement.

Given an array of integers `nums`, *find the next permutation of* `nums`.

Eg: **input: 1 2 3 4 2 6 7 18 12 9 0** **Output 1 2 3 4 2 6 9 0 7 12 18**

Approach: Steps: 1) if arr in decreasing order return reverse of the arr ie. If **3,2,1** return **1,2,3**

2) if not, iterate from 2nd last index and check **if(arr[i] < arr[i+1])** if yes **break;** this means we found a no. that can be interchanged to make next permutation. Here **i = 6** (arr[i] = 7)

3) now iterate from the last index till the ith index (i we get from step 2) **if(arr[j]>arr[i]) break;** here j = 9 (arr[j]=9)

4) swap arr[i] & arr[j] //**1 2 3 4 2 6 9 18 12 7 0**

5) reverse the arr from i+1 th index to last //**1 2 3 4 2 6 9 0 7 12 18**

TC = O(n) Sc = O(1)

121. 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 `i`th 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.
```

Approach: Steps: 1) Take 3 vars , ans =0, curr_pro = 0, low_price = INT_MAX

2) now iterate over the arr,

low_price = min(low_price,arr[i])

curr_pro = arr[i] – low_price

ans = std::max(ans,curr_pro)

3) return ans

TC = O(n) Sc = O(1)

Count pairs with given sum

Given an array of **N** integers, and an integer **K**, find the number of pairs of elements in the array whose sum is equal to **K**.

```
N = 4, K = 6          arr[] = {1, 5, 7, 1}          Output: 2
```

Approach : 1) Use an unordered_map<int,int>mp, ans =0

2) Iterate over the arr and check

if (mp[arr[i]]) ans+= mp[arr[i]]

mp[k – arr[i]] ++

3) Return ans

TC = O(n) Sc = O(n)

Common elements

Given three arrays sorted in increasing order. Find the elements that are common in all three arrays.

Note: can you take care of the duplicates without using any additional Data Structure?

```
Input: n1 = 6; A = {1, 5, 10, 20, 40, 80}     n2 = 5; B = {6, 7, 20, 80, 100}
n3 = 8; C = {3, 4, 15, 20, 30, 70, 80, 120}
Output: 20 80
n1 = 3; A = {3,3,3}    n2 = 3; B = {3,3,3}    n3 = 3; C = {3, 3,3} o/p = {3}
```

Approach: Take 3 pointers i=0,j=0,k=0

2) while(i<n1 && j<n2 && k<n3)

3) if all values are same and the value != last elemnt in our ans vector then push that val in ans i++,j++,k++

3) else values not same find the lowest value and increment that pointer respectively

4) return ans

TC = O(len of the arr with max number as the min in other 2 arrays) = O(n) Sc = O(1)

560. Subarray Sum Equals K

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
```

Approach: Steps 1) use a map<int,int>mp, sum = 0, cnt = 0

2) iterate over nums : sum+= nums[i]

If sum == k: ans++

If (sum-k) value is in map ans+= mp[sum-k]

mp[sum]++

TC = O(n) Sc = O(n)

152. Maximum Product Subarray

Given an integer array `nums`, find a contiguous non-empty subarray within the array that has the largest product, and return *the product*.The test cases are generated so that the answer will fit in a **32-bit** integer.

```
Input: nums = [2,3,-2,4]        Output: 6            Explanation: [2,3] has the
largest product 6.
```

Approach: **Steps:** **1)** curr = 1, ans = INT_MIN,

2) iterate over arr and curr *=arr[i] , ans = max(ans,curr) ; if curr == 0: curr=1

3) iterate over arr from last and curr *=arr[i] , ans = max(ans,curr) ; if curr == 0: curr=1

Return ans

TC = O(n) Sc = O(1)

Factorials of large numbers

Given an integer N, find its factorial. And return the ans in the form of a vector

```
Input: N = 10  Output: 3628800Explanation :  10! = 1*2*3*4*5*6*7*8*9*10 =
3628800
```

Approach: Steps 1) Create a vector ans = {1}
2) run a loop from 2 to N
Carry = 0
Iterate over ans from last:
temp = i*ans[j]+carry; carry = temp/10; ans[j] = temp%10
iterate till(carry>0):
insert val = carry%10 at the beginning of ans; carry/=10
TC = O(n^2) Sc = O(1)

128. Longest Consecutive Sequence

Given an unsorted array of integers `nums`, return *the length of the longest consecutive elements sequence.*

```
Input: nums = [100,4,200,1,3,2]  Output: 4  Explanation: The
longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length
is 4.
```

Approach: Steps: 1) insert all the values of nums in a set
2) Now iterate over the set:
If(set.count(num-1)): **continue;** //means if a no. 1 less than the curr no. is present in set skip it coz already iterated.
J = 1;
Iterate over set till num+j value is present : j++
Update ans = max(ans,j)
3) return ans
TC = O(n) Sc = O(n)

Given an array of size n and a number k, find all elements that appear more than " n/k " times.

Approach: Steps: 1) insert the frequencies of all values of nums in a map
2) Now iterate over the map:
And print those vals that are present more than n/k times
TC = O(n) Sc = O(n)

Array Subset of another array

Given two arrays: **a1[0..n-1]** of size **n** and **a2[0..m-1]** of size **m**. Task is to check whether a2[] is a subset of a1[] or not. Both the arrays can be sorted or unsorted.
Approach: Steps 1) insert the values of 1st arr in a map
2) iterate over the second arr2 : if arr2[i] is not in map than print **"NO"**
TC = O(n) Sc = O(n)

Triplet Sum in Array

Given an array arr of size n and an integer X. Find if there's a triplet in the array which sums up to the given integer X.
Approach: Steps1) sort the arr
2) iterate the arr from i=0
3) take j=i+1,k=n-1
4) while(j<k): if (a[i]+a[j]+a[k] == x) return true; if (a[i]+a[j]+a[k] > x) k—; else j++;

TC = O(n^2) Sc = O(1)

42. Trapping Rain Water

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]  Output: 6
```

```
Explanation: The above elevation map (black section) is represented by array
[0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section)
are being trapped.
```

Approach: Steps 1) take 4 variables left =0 , right = n-1 , left_max = 0, right_max = 0, ans = 0

2) while(left<=right)

If h[left] <= h[right] //means there exists a value in the right that is greater than left

If h[left] >= left_max: //means we don't have a greater value in the left

Left max = h[left]

else: **//**means there is greater value in the left

ans+= (left_max – h[left])

left++

else: **//** means there exists a value in the left that is greater than right

If h[right] >= right_max: //means we don't have a greater value in the right

right_max = h[right]

else: **//**means there is greater value in the left

ans+= (right_max – h[right])

right—

3) return ans

TC = O(n) Sc = O(1)

Chocolate Distribution Problem

Given an array **A[ ]** of positive integers of size **N**, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are **M** students, the task is to distribute chocolate packets among **M** students such that :

1. Each student gets **exactly** one packet.
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

```
Input:  N = 8, M = 5   A = {3, 4, 1, 9, 56, 7, 9, 12} Output: 6

Explanation: The minimum difference between maximum chocolates and minimum
chocolates is 9 - 3 = 6    by choosing following M packets :{3, 4, 9, 7, 9}
```

Approach: 1) sort the array , ans = A[m-1] – A[0]

2) sliding window of width **m** : ans = min(ans, A[i+m-1] – A[i])

TC = O(n log n) Sc = O(1)

209. Minimum Size Subarray Sum

Given an array of positive integers `nums` and a positive integer `target`, return the minimal length of a **contiguous subarray** $[nums_l, nums_{l+1}, ..., nums_{r-1}, nums_r]$ of which the sum is greater than or equal to `target`. If there is no such subarray, return `0` instead.

```
Input: target = 7, nums = [2,3,1,2,4,3]    Output: 2              Explanation:
[4,3]
```

Approach: 1) st=0 en=0 , ans = INT_MAX, sum=0

2) run a loop till en<n

Keep on adding the values in sum

If sum becomes >= target , keep on subtracting nums[st]

Update ans = min(ans, en-st+1) and increase st

TC = O(n) Sc = O(1)

Three way partitioning

Given an array of size n and a range [**a**, **b**]. The task is to partition the array around the range such that array is divided into three parts.

1) All elements smaller than **a** come first.

2) All elements in range **a** to **b** come next.

3) All elements greater than **b** appear in the end.

The individual elements of three sets can appear in any order. You are required to return the modified array.

Approach: DNF sort (exactly same as sort 0 1 2) **TC = O(n) Sc = O(1)**

Minimum swaps and K together

Given an array **arr** of **n** positive integers and a number **k**. One can apply a swap operation on the array any number of times, i.e choose any two index i and j (i < j) and swap arr[i] , arr[j] . Find the **minimum** number of swaps required to bring all the numbers less than or equal to **k** together, i.e. make them a contiguous subarray.

```
arr[ ] = {2, 1, 5, 6, 3}  K = 3                Output : 1

Explanation:To bring elements 2, 1, 3 together,swap index 2 with 4 (0-based
indexing),i.e. element arr[2] = 5 with arr[4] = 3such that final array will
be- arr[] = {2, 1, 3, 6, 5}
```

Approach: 1) We will use sliding window here

2) count the numbers <=k in the arr

3) now our subarray will be of length cnt

4) iterate from i=0 to cnt: and count the nums>k (high_cnt)

5) ans = high_cnt, i=0

6) iterate from j=cnt till n: if arr[j]>k: high_cnt++ if arr[i]>k: high_cnt-- then update ans = min(ans,high_cnt)

TC = O(n) Sc = O(1)

4. Median of Two Sorted Arrays

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return **the median** of the two sorted arrays.

```
Input: nums1 = [1,3,4,5,6], nums2 = [2,7,9,10,11,14] Output: 6.00000
Explanation: merged array = [1,2,3,4,5,6,7,9,10,11,14] and median is 6.
```

Approach: 1) we will try to partition both the arrays in equal halves (for odd length, left half will have 1 element more)

2) While making the partition check if max val in left half of nums1 <= min val in right half of nums2 **AND** max val in left half of nums2 <= min val in right half of nums1 :

if yes

If(even length)

ans = (max(max val in left half of nums1 and nums2) + min(min values in right halves of nums1 and nums2))/2

if odd length: ans = max(max val in left half of nums1 and nums2)

3) we will only look on the small size array (say nums1) if nums2 is small call the same function func(nums2,nums1)

```
int st = 0,en = n
while(st<=en)
   int cut1 = (st+en)/2;
   int cut2 = (n+m+1)/2-cut1;
   int left1 = cut1==0 ? INT_MIN: nums1[cut1-1];
   int left2 = cut2==0 ? INT_MIN: nums2[cut2-1];
   int right1 = cut1==n ? INT_MAX: nums1[cut1];
   int right2 = cut2==m ? INT_MAX: nums2[cut2];
   if(left1 <= right2 && left2 <= right1)
      if((n+m)%2==0){
         return (std::max(left1,left2)+std::min(right1,right2))/2.0;
      else{
```

```
                return std::max(left1,left2);
            else if(left1>right2){
                en = cut1-1;
            else{
                st = cut1+1;
```

TC = O(log(min(n,m)) Sc = O(1)