A2Z DSA Course

Arrays

Remove Duplicates from sorted array

- ➔ Use two pointers i,j. Point I at 0 index and j at 1 if a[i] != a[j], i += 1 and a[i] = a[j].
- ➔ Here we are changing the next ith element.
- ➔ T.C – O(N) S.C – O(1)

Rotate an array by k elements

- ➔ **Left rotate –** first reverse the array elements from 0 to k-1 and reverse array from k to n-k-1 and finally reverse the whole array.
- ➔ **Right rotate -** first reverse the array elements from 0 to n-k-1 and reverse array from n-k to n-1 and finally reverse the whole array.
- ➔ T.C – O(N) S.C – O(1)

Move zeroes to end

- ➔ Similar to removing duplicates, but j should be placed at first 0 occurance index, iterate the array from j+1 to n, if a[i] != 0 swap a[i] and a[j] and j += 1
- ➔ T.C – O(N) S.C – O(1)

Find missing number in an array

- ➔ **Using math –** Find sum of n natural numbers and subract the sum of array from it
- ➔ **Using XOR –** XOR the numbers from 1-n and XOR the whole array. Finally XOR the two to get missing element.
- ➔ T.C – O(N) S.C – O(1)

Longest subarray with sum K

- ➔ **Positive Elements**
  - o **Hashing –** Take a presume hashmap, where sum of elements w.r.t their indices are stored. Traverse the array and add the element to a variable sum. Check if k-sum is present in hashmap, if present check for the max-length.
  - o T.C – O(N) or O(NlogN) depending upon D.S S.C – O(N)
  - o **Two Pointer –** Take two pointers left and right and place them at 0 index. Run a loop for right pointer till n, and a left nested loop where the left pointer element will be removed from the sum if sum > k. Outside nested loop check if sum == k and chose the maxlength.
  - o T.C – O(2*N) S.C – O(1)
- ➔ **Positive and Negative Elements –** We can only using hashing approach for positive and negative elements

Sorting 0s, 1s and 2s array

- ➔ Keeping count of values and overwrite the array using these values T.C – O(N) + O(N) S.C – O(1)
- ➔ **Using 3 pointer –** set mid = 0, low = 0 and high = n-1. If a[mid] == 0 swap low & mid values and mid++, low++ if a[mid] == 1 mid++ and a[mid] == 2 swap mid & high and high--.
  T.C – O(N) S.C – O(1)

Majority Element >N/2 times

- ➔ **Hashing –** Keep track of each element and return element with >n/2 frequency T.C – O(NlogN) + O(N) S.C – O(N)
- ➔ **Moore's Voting Algorithm –** Initialize c=0, element = None. Traverse loop and check if c==0 element = i and c++, if i == element c++ else c-=1 return the element. T.C – O(N) S.C – O(1)

Kadane's Algorithm: Maximum Subarray sum in an array

- ➔ Add the array element to sum variable compare with max_sum and if sum <0 sum = 0. We're not considering negative sum as answer. T.C – O(N) S.C – O(1)

Stock Buy and Sell

- ➔ Keep track of minPrice and maxProfit while traversing array. minPrice = min(minPrice,a[i]) maxProfit = max(maxProfit, a[i] – minPrice) return maxPrice. T.C – O(N) S.C – O(1)

Rearraing the array in alternate positive and negative elements

- ➔ **Brute Force –** Use two array pos and neg to store respective elements and rearrange the array by iterating pos and neg arrays. T.C – O(N + N/2 + N/2) S.C – O(N/2 + N/2). This is the only solution if pos and neg elements are not same number.
- ➔ Take a res array and two indices posI and negI travers ethe array if element is pos add to res[posI] and posI += 2 else add element to res[negI] and negI += 2. T.C – O(N) S.C – O(N)

Next Permutation

- ➔ **Longest prefix match –** Find the break point in the array. A break point is an element less than its next element. First break point should be from back of the array. Next swap the element from back less than break point element. Next reverse the array from break point to end. T.C – O(N + N + N) S.C – O(1)

Leaders in an Array

- ➔ Traverse from back and keep track of max element if current element is max element insert current element to res array and change max to current. T.C – O(N) S.C – O(N)

Longest consecutive sequence in an array

- ➔ **Using Sorting –** Sort the array and keep track of small element check if a[i] – 1 is small element and increase count and set small Element = a[i] elif a[i] !+ small element set small element to a[i] and set count to 1. T.C – O(NLogN) + O(N) S.C – O(1)
- ➔ **Using Set DS –** We use set to store all the elements and next iterate the set to check if x-1 is present in the set where x -> element of set. If present x would not be a starting element so continue. If x-1 not exists run a while loop until x-1 is in set, increment c+1 and set x = x-1. Set the max_length to max of count and max_length. T.C – O(N) + O(2*N) S.C – O(N)

Zero Matrix

- ➔ First we declare two arrays row and col. Traverse the matrix if cell(I,j) is 0 mark row[i] = 0 and col[j] = 0. Next traverse the magtrix again and check if the corresponding row or col of the cell has 0 or not if 0 mark cell(I,j) as 0. T.C – O(2*(N*M)) S.C – O(N)+O(M)
- ➔ Optimal approach is to reduce the space complexity. Instead of using extra row and col arrays use the first row as col array and col as row array. But cell(0,0) would be common to

both so just take a variable as Col0 and assign 0^{th} column value to it. And follow the same steps as previous approach. T.C – O(2*(N+M)) S.C – O(1)

Rotate matrix by 90degree

➔ Just transpose the matrix and reverse each row. T.C – O(N*N) + O(N*N) S.C – O(1)

Spiral Matrix

➔ We will be using 4 loops to move from left to right, top to bottom, right to left and bottom to top. Take 4 variables top=0, right = m-1, bottom = n-1, left =0 where n = len(mat) and m =len(mat[0]). Iterate a while loop till left <= right and top <= bottom, inside while loop we need to construct 4 loops, first loop from left to right append a[top[i] to res array later top++ loop 2 top to bottom append a[i][top later top—3^{rd} loop from right to left append a[bottom][i] and later bottom—4^{th} loop from bottom to top append a[i][left] later left++. T.C – O(M*N) S.C – O(M+N)

Count all subarrays with sum k

➔ Use pre sum map and keep track of pre sum and its occurance, initially add a 0 presum with cnt 1 to the map. Travers ethe array add elements to sum variable. Check if sum-target is present in map, if present add the occurance of it to the total count. T.C – O(N) + O(NLogN) S.C – O(N)

Pascal's Triangle

➔ **Variation 1 –** given row and col print element at (row,col)
- Naive Approach would be to find each element in pascal triangle and print the element at (r,c). T.C – O(N) + O® + O(N-R). Optimal Approach is combination of N and r nCr. To calculate nCr optimally we can loop till r and keep a res variable to 1. In the loop multiple res with n-i and divide by i+1. Example -> 8C4 = (8*7*6*5)/(4*3*2*1) T.C – O(C) C->Column S.C – O(1)

➔ **Variation 2 –** Given N print Nth row of pascals triangle.
- Naïve approach would be to calculate the whole nth row elements using variation 1. T.C – O(N*C). Optimal approach is to find a series we can see every row starts with 1. Example N=6 o/p -> 1 5 10 10 5 1. Here 1^{st} element is 1 next element is 5 can be written as 5/1 same with 10 5*4/1*2 next would be 5*4*3/1*2*3 so on. We can code this by traversing loop from 1 to n and keep track of ans variable multiple with n-i and divide by i. T.C – O(N) S.C – O(1)

➔ **Variation 3 –** Print the pascal triangle.
- Naïve approach is using variation 2 naïve approach to print triangle. T.C – O(N*R*C) S.C – O(1). Optimal is to use variation 2 optimal approach. T.C – O(N*N) S.C – O(1).

Majority Elements >N/3 Times

➔ **Extended Boyer Moore's Voting Algorithm –** Use similar approach of >n/2 problem, just keep track of 2 elements and 2 counts instead of 1. T.C – O(N) + O(N) S.C – O(1)

3 Sum Problem

- ➔ **Hashing –** Traverse 2 loops and check if target – (a[i] + a[j]) is in hashmap, if in hashmap sort it and add to set. If not add a[j] to hashmap. T.C – O(N^2 + Log(no of unique triplets)) S.C – O(2*no of unique triplets) + O(N)
- ➔ **3 Pointer –** Use 3 pointers i,j,k where I is fixed and j and k would be moving. Initially I is at 0 and j=i+1 and k=n-1. Check if summation of these 3 gives k. if sum==0 add to res array elif sum<k j++ else k--. Also skip duplicates by using while loop. T.C – O(NLogN) + O(N*N) S.C – O(no of triplets)

4 Sum Problem

- ➔ Same as 3 sum problem just add extra loop to every approach.

Length of longest subarray with sum 0

- ➔ If a subarray(i,j) has sum s and a subarray(i,x) also has sum s, we can say thay subarray(x+1,j) sum is 0. Keep track of presum in a hashmap, check if current presume is present in the hashmap if true comp[are the length with max_length. T.C – O(N) S.C – O(N)

Count no of subarrays with xor k

- ➔ Same approach as count all subarrays with sum k.

Merge Overlapping Sub-intervals

- ➔ Sort the array. Traverse elements add first element to res array, check if next elements overlapping by comparing with interval in res array. If true adjust the interval in res else, else add the element to res array. T.C – O(NLogN) + O(N) S.C – O(N)

Merge two sorted arrays without external array

- ➔ Use two pointers left and right. Set left to len(arr1) -1 and right to 0. Compare arr1[left] to arr2[right] if left > right swap and left—right++. Sort the two arrays. T.C – O(min(N,M)) + O(NLogN) + O(MLogM) S.C – O(1)
- ➔ **Using Gap Method –** gap = ceil(len(arr1)+len(arr2))/2. Loop till gap>0 true, set left=0 right=gap loop till right <len(arr1) + len(arr2) and swap according to left and right pointers. T.C – O((N+M)*(Log(N+M))) S.C – O(1)

Find the repeating and missing elements

- ➔ **Using math –** Use summation of array and 1-n and also sum of squartes of array and 1-n to find X and Y. S – Sn = X – Y X^2 – Y^2 = S2 – S2n X+Y = S2-S2n / S – Sn. T.C – O(N) S.C – O(1)
- ➔ **Using XOR –** Find X^Y by xoring all elements in array and 1-n. Find the different bit from repeating and missing element, i.e first set bit from right of X^Y. Based on the position of the different bit group array elements and 1-n into two groups 1 bit and 0 bit. Now Xor the two groups to get two numbers. Check thye array to find which is missing and which is repeating. T.C – O(N) S.C – O(1)

Count Inversions in an array

- ➔ **Using Merge Sort Technique –** Compare elements from 2 sorted arrays by using 2 pointers, i=0 of arr1 and j=0 of arr2 if a[i] <= a[j] i++ else we can say that remaining elements in arr1 are also > a[j]. Increase count acc to that. Modify merge function. T.C – O(NLogN) S.C – O(N)

Count Reverse Pairs

- ➔ Similar approach to count inversions with a slight change, compare I and j pointers by a[i] > 2* a[j] i++ j++ set the count value. If a[i] > 2*a[j] elements before a[i] are also > 2*a[j] set the count acc to it. Modift merge sort function. T.C – O(2NLogN) extra N for countPairs function S.C – O(N)

Maximum Product Sub-array

- ➔ Need to consider 4 ways, if elements all are +ve whole array product is answer, if even -ve elements whole array product is answer, if odd -ves either suffix or prefix is answer, if 0 element start the product at next element as 1. First run a loop and find prefix product and also suffix product. If prefix and suffix = 0 set them to 1. T.C – O(N) S.C – O(1)

Binary Search

On 1D Arrays

Floor and Ceil in a sorted Array of given K

- ➔ Use Lower bound and upper bound. T.C – O(logN) S.C – O(1)

First and Last occurrences of an element in a sorted Array

- ➔ Use Lower bound and upper bound. T.C – O(logN) S.C – O(1)
- ➔ Use Binary Search from scratch. Write two BS functions to find last and first just add a last = mid or first = mid acc to fun when a[mid] == k. T.C – O(2*logN) S.C – O(1)
- ➔ Use the same approaches to find count of occurrences of a number.

Search Element in a rotated sorted array I

- ➔ If we find the mid element we can say which half is sorted. If a[low] <= mid and if target <= a[mid] and a[low] <= target we can eliminate right half else left half. Similar conditions to check for high, if a[high] >= a[mid] right half is sorted and check if target is in between mid and high. T.C – O(logN) S.C – O(1)

Search Element in a rotated sorted array II

- ➔ Consequently, the previous approach will not work when **arr[low] = arr[mid] = arr[high].** If duplicate are present in the array, the array may get sorted [3, 3, 3, 3, 3]. In such cases above approach will fail. Need to add extra condition if a[mid] == a[low] == a[high] just increase low++ and high--.

Minimum in rotated sorted array

- ➔ Similar to above problem, find mid and check which half is sorted and find the min. T.C – O(LogN) S.C – O(1)

Find Number of times an array was sorted

- ➔ Use the same approach of above problem, just return the index of min element.

Search Single Element in a sorted array

➔ Before single element the two elements are placed at even and odd indices after single element it becomes vice-a-versa. Check if mid is in left or right half and eliminate it. Also check for few edge cases if array size is 1 and for low =0 and high = n-1 cases. T.C – O(logN) S.C – O(1)

Peak Element In an Array

➔ A peak element is > its left and right elements. We can eliminate left and right halves by observing whether mid lies in left or right half of a peak. If a[mid-1] < a[mid] left side of the mid lies in left half as it is in increasing order and if a[mid+1] > a[mid] right side lies right of mid as it is in decreasing order. Also check for edge cases for 0 and n-1 indices. T.C – O(logN) S.C – O(1)

BS on Answers

Sqrt of a number

➔ Place low = 1 and high = n find mid and check if mid*mid <= n set low = mid + 1 else high = mid - 1. Return high, at one point high will be less than low that will be the answer. T.C – O(logN) S.C – O(1)

Koko Eating Bananas

➔ We can say that min bananas money can eat is 1 and max is max(a). BS on 1,max(a) and loop through the array and find total hours by getting ceil of a[i]/mid check if total > h then low = mid + 1 else high = mid – 1. T.C = O(N*Log(max(a))) S.C – O(1)

Minimum Days to make m bouquets

➔ Impossible case would be if arr length < m*k. The search limit would be min(a) and max(a). Do a BS on the limit and check for every element in the arr if it is <= mid. And set the count acc to k. T.C – O((log(max(a)-min(a)+1)*N) S.C – O(1)

Find the Smallest Divisor Given a Threshold

➔ The search limit would be 1 and max(a). Check the count of divisors when elements divided by mid. T.C – O(log(max(a))*N) S.C – O(1)

Capacity to Ship Packages within D Days

➔ The max limit would be sum(a) and the min limit would be max(a) – as every element should be able to get completed in the given limit. Do a BS on [max(a), sum(a)] and check if mid would be a suitable answer. Keep a count of days and a load variable to add each element and increase count if load > mid. T.C – O(N * log(sum(a) – max(a) + 1)) S.C – O(1)

Kth Missing Positive Number

➔ Calculate the missing no of elements at each index by a[i] – (i+1). Now do a BS and check for the no of missing elements at mid index. If the missing numbers are < k eliminate left half else right half. Return k + high + 1 as res. T.C – O(logN) S.C – O(1)

Aggressive Cows

- ➔ Sort the array. Min possible distance to place cows is 1 and max distance is max(a)-min(a). Now check if cows can be placed from min dist. T.C – O(NlogN) + O(N*log(max(a)-min(a))) S.C – O(1)

Allocate Minimum Number of Pages

- ➔ Min number of books assigned to a student is max(a) and max is sum(a). Do a binary search on limit [max(a), sum(a)]. Keep a track of sum of elements if sum > mid increase books count. Eliminate left half if count > m else eliminate right half. T.C – O(log(max(a), sum(a)+1) * N) S.C – O(1).

Split Array Largest Sum & Painter's Partition

- ➔ Same as allocate min number of pages problem.

Minimize Maximum Distance between Gas Stations

- ➔ **Brute Force:** Take an additional array to stores no of gas stations to be placed in the gaps. Take the highest gap and find the sectionLength = diff/(howMany[i]+1). Continue the process until k gas stations are placed. T.C – O(K*N) S.C – O(N-1)
- ➔ **Better Approach:** To optimize T.C. Use priority queue add gaps of the array with index to PQ. Run a k loop, pop PQ and check for gas stations to be placed. And change the PQ acc to gas stations placed. Return the top element of PQ. Additional array to store gas stations would be same. T.C – O(NlogN + KlogN) S.C – O(N-1) + O(N-1)
- ➔ **Optimal Approach:** Using BS, cant use regular BS, use low = 0 and high = max gap. Condition would be low – high > 10^-6. Check if mid gap can be used to place k gas stations. Keep a cnt of gas stations placed using mid as gap. There would be an absolute case while placing gas stations with mid as gap i.e if mid = 0.5, when 0.5/2 gives 1 but gas stations should be 2. In an absolute case add 1 to cnt. If cnt > k then low = mid else high = mid. Return high. T.C – O(N*log(Length of answer space)) + O(N). S.C – O(1)

Median of Two Sorted Arrays of different sizes

- ➔ **Better Approach:** Use merge fun in merge sort to merge arrays. We can reduce the space complexity by removing merge array and keeping a count of elements merged. T.C – O(n1+n2) S.C – O(1)
- ➔ **Optimal Approach:** A median divides array into two equal halves. Try to select elements from arr1 and arr2 and take it as 2 halves selected elements from arr1 and arr2 would be of one half and the remaining would be second half. Check if the divided halves are valid or not. By assigning variables l1 and r1 to last elements of arr1 and arr2 of left half. And l2 and r2 for first elements of arr1 and arr2 of right half. If l1 <= r2 and r1 <= l2 the halves are valid and return (max(l1,r1)+min(l2,r2))/2. T.C – O(log(min(n1,n2)) S.C – O(1)

Kth Element of two sorted arrays

- ➔ Similar to above problem with few modifications.

BS on 2D Arrays

Find the row with maximum number of 1's

➔ For every row find the first occurrence of 1 using lower bound BS approach. T.C – O(NlogM) S.C – O(1)

Search in a sorted 2D matrix

➔ Complete matrix is sorted, if we place all the elements as a 1d array it will be a sorted array and we can apply BS to find k. To get the element, we will convert index 'mid' to the corresponding cell using the above formula. Here no. of columns of the matrix = M.
row = mid / M, col = mid % M.

Search in a row and column-wise sorted matrix

➔ We can start traversal at 4 corners, but choose two corners as both have decreasing and increasing sequences – (0,m-1) and (n-1,0) m->col size and n->row size. If a[row][col] == k, then return True. If a[row][col] > k eliminate col else eliminate row. T.C – O(N+M) S.C – O(1)

Find Peak Element (2D Matrix)

➔ Use similar approach as Peak Element in 1D Array. Traverse row or col wise. Ex – col wise low = 0 high = m-1, find mid and get max element from mid col. Check if max element is peak by comparing to its adjacent elements. If peak return the max element.
If previous element > max then eliminate right half else eliminate left half. T.C – O(LogM * N) S.C – O(1)

Median of row wise sorted matrix

➔ Do BS on limit 1, 10^9. Find mid and check how many elements in the array are <= mid. Use upper-bound approach to check this. If the cnt <= (m*n)//2 eliminate left half else right half. Finally return low. T.C – O(log2(2^32) * N * logM) -> O(32 * N * logM) S.C – O(1)

Strings

Remove Outermost Parentheses

➔ Take stack to store parentheses. For "(" If stack is not empty add parentheses to result, and add to stack. For ")" pop the stack and if stack is not empty add to res. T.C – O(N) S.C – O(N)
➔ Instead of stack keep a count of parentheses, if "(" increase the count and if c != 1 add to res. If ")" c--, and if c >= 1 add to res. T.C – O(N) S.C – O(1)

Longest Common Prefix

➔ Sort the array. Take first and last elements of the array. Print the common characters in both strings. T.C – O(NlogN) + O(min(len(first), len(last)) S.C – O(1)

Count with K distinct characters

➔ The idea is to count all the subarrays with at most K distinct characters and then subtract all the subarrays with atmost K – 1 characters. That leaves us with count of subarrays with exactly K distinct characters. T.C – O(N) S.C – O(1)

Longest Palindromic Substring

➔ Take two pointers p1 and p2. For odd p1 = i-1 and p2 = i+1, for even p1 = I and p2 = i+1. Run a loop to check if s[p1] == s[p2] then p1—and p2++. This gives palindromic string, check if length is > max_length, then set this string to res. T.C – O(N*N) S.C – O(1)

Linked Lists

Middle Element

➔ Use slow and fast pointers. Slow moves at one step and fast at two steps, when fast is at end of linked list slow will be at middle. Return slow. T.C – O(N) S.C – O(1)

Detect and Remove loop

➔ Use the same slow fast pointer approach, if slow == fast loop exists. Set slow = head and check until slow->next == fast->next, set fast->next to None. T.C – O(N) S.C – O(1)

Reverse a Linked List

➔ **Iterative Approach –** Flip the node link by using temp and prev variables, assign prev to none and temp to head. Traverse LL, set front to temp->next set temp->next to prev, prev to temp and temp to front. T.C – O(N) S.C -O(1)

➔ **Recursive –** Decreasing the length of nodes by recursion, set base condition to return the head as last node. For every recursion set front as head->next and front-> next to head and head->next to None. T.C – O(N) S.C – O(1)

Is Palindrome

➔ Use slow and fast pointers to get middle node of LL. Reverse the LL from slow->next node. Set first to head and second to new-head. Check if the values are same for first and second till second is None. T.C – O(2*N) – N for traversing and another N for reversing. S.C – O(1)

Remove Nth node from End

➔ Use slow and fast pointer technique, set fast pointer at nth node from beginning. Slow will be at head, traverse till fast->next is None. Slow pointer would be placed at (L-N)th node L – len of the LL. Delete the slow->next node. T.C – O(N) S.C – O(1)

Delete Middle Node

➔ We know slow will be at mid if we traverse LL using slow and fast pointers. But we need node before mid to remove mid node. Just skip one slow traverse. To do this start fast at head->next->next and slow at head. T.C – O(N/2) S.C – O(1)

Sort a LL

➔ Apply merge sort on LL. Find the mid by using slow and fast pointer approach. Divide the LL based on mid, merge the two LLs using dummy node. T.C – O(logN + N + N/2) S.C – O(1)

Sort LL of 0s, 1s and 2s

➔ We can change the links of 0, 1 and 2 nodes using dummy nodes. T.C – O(N) S.C – O(1)

Intersection point of 2 LLs

- ➔ Find the difference between two LLs, start the larger LL at the difference node and traverse two LLs return when two LLs nodes are equal. T.C – O(2*M) S.C – O(1)
- ➔ Take two dummy nodes for two LLs, traverse them if anyone becomes null point them to head of opposite LL. Traverse till dummy1 != dummy2. T.C – O(2*M) S.C – O(1)

Add one to LL

- ➔ **Iterative –** Reverse the LL, add carry from head, reverse the LL after traversal. If carry == 1 after LL traversal, add newNode(1) to head. T.C – O(3N) S.C – O(1)
- ➔ **Recursive –** We can use recursion to add carry from reverse. Create a recursive function passing head, if head reaches null return 1 as carry. Add the carry to the next nodes and return remaining carry. If carry == 1 after recursion add a new node to head. T.C – O(N) S.C – O(N) -> recursive stack space

Reverse Nodes in k-Group

- ➔ Find Kth node for every group, reverse the k nodes, connect the nodes using prev and next dummy nodes. Traversal until kthnode is NULL. T.C – O(2N) S.C – O(1)

Rotate LL to right by k nodes

- ➔ Find the length of LL. Set k=k%n n -> length of LL. Make LL Circular LL. Cut the link at N-K node. T.C – O(N) + O(N – (K%N)) S.C – O(1)

Flattening a LL

- ➔ Use merge two LL technique, merge 2 LLs from end using recursion. T.C – O(N) S.C – O(1)

Clone Linked List

- ➔ Insert deep copy nodes next to original nodes. Traverse again and set random pointers to deep copied nodes. Use dummy node and traverse LL to set next pointers. T.C – O(N + N + N) S.C – O(1)