Word break 1:

Question:

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Approach:

eg: leetcode and worddict = [leet, code] we used recursive solution to find first leet and then we made a call to find the rest of the word using recursion from main function we made a call to check function starting with the index 0 and check function uses a for loop for moving the ending index to get the leet and we check the condition if leet exist in the word dict then we make a call for recursive function to check for next words

Code:

```
class Solution {
    private Boolean dp[];
    public boolean check(String s, int idx,List<String> wordDict, int n, Boolean
dp[]){
        if(idx >= n){
            return true;
        }
        if(dp[idx] != null){
            return dp[idx];
        }
        for(int end = idx+1; end <= n; end++){
            String trim = s.substring(idx, end);
            if(wordDict.contains(trim) && check(s, end, wordDict, n, dp)){
                return dp[idx] = true;
            }
        }
        return dp[idx] = false;
    }
    public boolean wordBreak(String s, List<String> wordDict) {
        dp = new Boolean[s.length()+1];
        return check(s, 0, wordDict, s.length(), dp);
    }
}
```

Time complexity:

Good example, but you might be overcomplicating things. Given an array of length n, there are n+1 ways/intervals to partition it into two parts. Each interval has two choices - split or not. In the worse case, we will have to check all possibilities, which becomes O(2^(n+1)) -> O(2^n). This analysis is similar to palindrome partitioning

Space complexity :

O(N)

Question2:

Word break 2:

Given a string s and a dictionary of strings wordDict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in **any order**.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Approach :

This question is also similar like above one we have to check if trimmed word at from ith to jth index exists in word dict or not if yes the call for another part recursion and add space if there is length more than 1 of a string if reach to the last index it means that we have success fully reach to that index without failure so add the string to the list and return from there

code:

```
class Solution {
    List<String> al = new ArrayList<>();
    public void findWords(String s, List<String> wordDict, int i, int n, String
currSen){
        if(i == n){
            al.add(currSen);
            return;
        }
        for(int end = i+1; end <= n; end++){
            String trim = s.substring(i, end);
            if(wordDict.contains(trim)){
                String ogsentence = currSen;
                if(currSen.length() > 0){
                    currSen += " ";
                }
                currSen += trim;
                findWords(s, wordDict, end, n, currSen);
                currSen = ogsentence;
            }
        }
```

```
        return;
    }
    public List<String> wordBreak(String s, List<String> wordDict) {
        // StringBuilder sb = new StringBuilder();
        findWords(s, wordDict, 0, s.length(), "");
        return al;
    }
}
```

Complexities

O(2^n) time

O(n) space

Q3) sum of two numbers In the particular subarray in the array find the sum which is maximum

Comment :

Q4) permutation of an number

Approaches: there are two approaches

1st) backtracking with one for loop

In this approach we were adding and removing the number in a list and we are running a for loop to add all the numbers if all the elements already exist in the list then we just ignore and continue to iteration

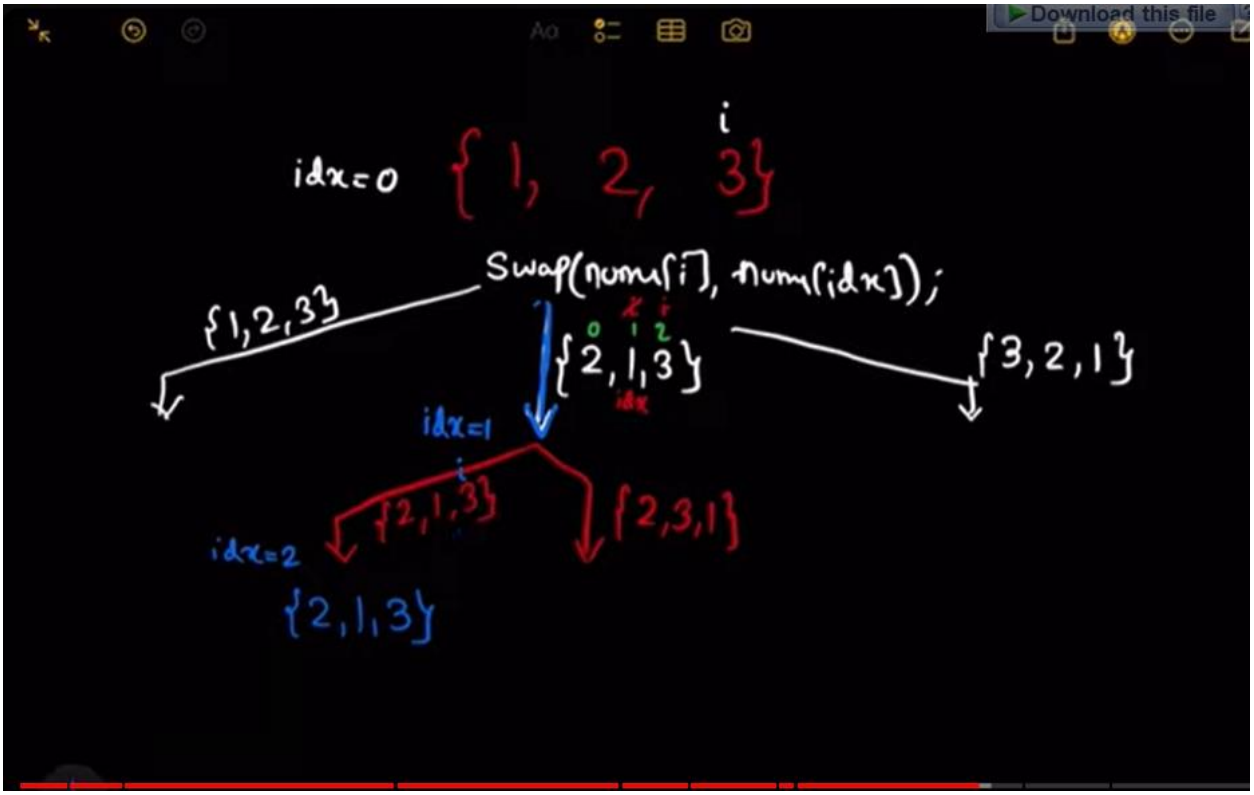
		1 2 3	
	1	2	3
12	13		
123(return)	132(return)	similar approach for the others	

2nd) swaping approach one for loop

In this approach we are just swaping the element we are carrying an index variable

And we are using a for loop which is not going to iterate from start to end because we will get same numbers multiple times for this approach so we are running for loop for index to n

And swaping elements at I,idx calling recursive function and again putting back the change



Q5) Minimum Cost To Make Two Strings Identical(Note this approach is same as lcs approach)

Code:

Approach:

Here we are playing with the index only starting with top down approach we are checking if both are equal the call recursively for decreasing bith the index if not same then we are exploring the paths first we are reducing the index of first string and adding the cost of that string and similar for the second one we are taking minimum of both

```
class Solution {

   public int cost(String x, String y,int costX, int costY, int i, int j, int dp[][]){

      if(i < 0){

         return costY * (j+1);

      }

      if(j < 0){

         return costX * (i+1);

      }
```

```
        if(dp[i][j] != -1){
            return dp[i][j];
        }
        if(x.charAt(i) == y.charAt(j)){
            return dp[i][j] = cost(x, y, costX, costY, i-1, j-1, dp);
        }
// here if not dp[i][j] then we have to use else
        return dp[i][j] = Math.min(costX+cost(x, y, costX, costY, i-1, j, dp), costY+cost(x, y, costX, costY, i, j-1,
dp));
    }
    public int findMinCost(String x, String y, int costX, int costY) {
        int dp[][] = new int[x.length()+1][y.length()];
        for(int[] temp : dp){
            Arrays.fill(temp, -1);
        }
        return cost(x, y, costX, costY, x.length()-1, y.length()-1, dp);
    }
}
```

Q) LCS

Code:-

```
class Solution {
    public static int lcs(String s1, String s2, int i, int j){
        if(i < 0 || j < 0){
            return 0;
        }
        if(s1.charAt(i) == s2.charAt(j)){
            return 1+lcs(s1, s2, i-1, j-1);
        }
```

```
        else{
            int first = lcs(s1, s2, i-1, j);
            int second = lcs(s1, s2, i, j-1);
            return Math.max(first, second);
        }
    }
    public int longestCommonSubsequence(String text1, String text2) {
        return lcs(text1, text2, text1.length()-1, text2.length()-1);
    }
}
```

Q6) **552. Student Attendance Record II**

Int this approach we have explored each option as string were not given so were dealing with the numbers only first we have solved took a count variable then we make count for if present is their for that absence count remains as it is but consecutive late will get o for the absence the absence count get increased and late count 0 and for the late count we have same absence count but late count is 1 in this question we are checking whether the condition is breaking of more than 1 absence and more than equal to 3 consecutive late count

Code:

```
class Solution {
    private final int mod = 1000000007;
    public int findRecord(int n, int acount, int lcount, int dp[][][]){
        if(acount > 1 || lcount >= 3){
            return 0;
        }
        if(n == 0){
            return 1;
        }
        if(dp[n][acount][lcount] != -1){
            return dp[n][acount][lcount] % mod;
        }
        int count = 0;
        count = (count + findRecord(n-1, acount, 0, dp)) % mod;
        count = (count + findRecord(n-1, acount+1, 0, dp)) % mod;
        count = (count +  findRecord(n-1, acount, lcount+1, dp)) % mod;
        return dp[n][acount][lcount] = count;
    }
```

```
    public int checkRecord(int n) {
        int dp[][][] = new int[n+1][2][3];
        for(int[][] temp : dp){
            for(int[] curr : temp){
                Arrays.fill(curr, -1);
            }
        }
        return findRecord(n, 0, 0, dp);
    }
}
```

Q7) Longest subsequence-1

Aproach1:

a) First we have tried with backtracking using recursive calls in which first call was for acceptance condition where condition were match so we can say it take it condition and added into the list the removed it and not take it condition and call recursive for exploration
But in this approach we can optimize it by using only one variable and we don't need of any extra list instead of that we can use two variables one for size and one for previous number in adding condition we increase the size and update the previous number with current number and in not take it condition we have not updated the previous with current and size was also kept same
Code:
For approach 1:

```
class Solution {
  public static void findMaxLength(int n, int[] a, int idx ,int max[], List<Integer> al){
    max[0] = Math.max(max[0], al.size());
    if(idx == n){
      return;
    }
    if(al.size() == 0 || Math.abs(al.get(al.size()-1) - a[idx]) == 1){
      al.add(a[idx]);
    }
    findMaxLength(n, a, idx+1, max, new ArrayList<>(al));
    al.remove(al.size()-1);
    findMaxLength(n, a, idx+1, max, new ArrayList<>(al));

  }
  public static int longestSubseq(int n, int[] a) {
    // code here
    int max[] = new int[1];
    List<Integer> al = new ArrayList<>();
    findMaxLength(n, a, 0, max, al);
```

```
        return max[0];
    }
}
```

Code for approach 2:

```
class Solution {
    public static int findLength(int n, int[] a, int prev, int idx, int size){
        if(idx == n){
            return 0;
        }
        int count = 0;
        if(prev == -1 || Math.abs(prev - a[idx]) == 1){
            count += 1 + findLength(n, a, a[idx], idx+1, size+1);
        }
        int count2 = findLength(n, a, prev, idx+1, size);
        return Math.max(count, count2);

    }
    public static int longestSubseq(int n, int[] a) {
        // code here
        return findLength(n, a, -1, 0, 0);
    }
}
```

Q7) 128. Longest Consecutive Sequence

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

You must write an algorithm that runs in O(n) time.

Example 1:

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:

1st) this is brute force with O(n^2)

2nd) this method uses sorting of array and in this method we are increasing the count if the number is increasing else we set again count to 1 and continuously we are updating the length

3rd) in this method we are storing whole array in hashset and while iterating in the hashset we are looking for the starting element of any sequence means if 100 102 4 3 2 1

Here 100 is starting element as it don't have any element before this means 99 so after that we are increasing count if any element exists in the hashset more than 1 same for 1

Code:-

```
class Solution {
    public int longestConsecutive(int[] nums) {
        if(nums.length == 0){
            return 0;
        }
        Arrays.sort(nums);
        int length = 1;
        int count = 1;
        for(int i = 1; i < nums.length; i++){
            if(Math.abs(nums[i]-nums[i-1]) == 1){
                count++;
            }
            else if(nums[i] -nums[i-1] != 0){
                count = 1;
            }
            length = Math.max(count, length);
        }
        return length;
    }
}
```

Q8)

1404. Number of Steps to Reduce a Number in Binary Representation to One

Approach :

1st) int this method we have seen an observation that if the number is even then the lsb would be 0 else 1

(This conditions were given in the question itself)So if the number is even we simply removed it from the string but if the number is odd then we added one to it means if number is odd the lsb would be 1 so we need to carry some carry after addition after doing some observations on some testcases we have to just find first occurred 0 then change it to 1 and after that 1 all the number should changed to 0;

10001 (odd)-> count = 0(here count is number of operation)

10010(even) count = 1

1001(odd) count = 2

1010(even) count = 3

101(odd) count = 4

110(even) count = 5

11(odd) count = 6

100(even) count = 7

10(even) count = 8

1(ans) count = 8

On the basis of above observation the code were going in to time complexity of o(n^2) but we have reduced it to big o(n)

Code:

```
class Solution {
    public int numSteps(String s) {
        int count = 0;
        int carry = 0;
        for(int i = s.length()-1; i > 0; i--){
            int curr = (s.charAt(i)-'0')+carry;
            if(curr % 2 == 1){
                carry = 1;
                count += 2;
            }
            else{
                count += 1;
            }
        }
        return count + carry;
    }
}
```

Q9) optimal startegy for a game questions

Q10) String Subsequence

Approach :

1) **We have tried for take it not take it approach and we added in the base case if at any moment if we get the equal string then we return 1;**
   Else we return 0 if we get index out of bound
2) **In this method we deal with the indexes rather then the actual string at one place we checking if the current charcter at both indexes are same then we reduce the both string this is the condition were both the character at ith index are equal in this condition we are doing two thing one is considering both the indexs on s1 and s2 and reducing both the indexes another one is we stoped one index at s2 and moving index which is at s1**
   Code:

```
class Solution {
    static int mod = (int)1e9+7;
    public static int count(String s1, String s2, int m, int n, int[][] memo){
        if(n < 0){
            return 1; // Reached end of s2, found a match
        }
        if(m < 0){
            return 0; // Reached end of s1, but not end of s2, no match found
        }
        if(memo[m][n] != -1) {
            return memo[m][n]; // If result already calculated, return from memo
        }
        int counting = 0;
        if(s1.charAt(m) == s2.charAt(n)){
            counting = ((count(s1, s2, m-1, n-1, memo) % mod )+ (count(s1, s2, m-1, n, memo) %
mod))%mod;
        } else {
            counting = count(s1, s2, m-1, n, memo) % mod;
        }
        memo[m][n] = counting; // Store result in memoization table
        return counting;
    }

    public static int countWays(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();
        int[][] memo = new int[m][n];
        for (int[] row : memo) {
```

```
            Arrays.fill(row, -1); // Initialize memoization table with -1
        }
        return count(s1, s2, m-1, n-1, memo);
    }
}
```

Date: 31/05/2024
Concept bit manipulation
Question:

260. Single Number III

Description: Given an integer array nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once. You can return the answer in **any order**.

You must write an algorithm that runs in linear runtime complexity and uses only constant extra space.

Key learnings: in this question we have learned about the xor concepts and how to find the bit mask using 2s complement (here bit mask was to find the first set bit)

Optimal code with tc(O(n)):

```
class Solution {
    public int[] singleNumber(int[] nums) {
        int ans[] = new int[2];
        int num = nums[0];
        for(int i = 1; i < nums.length; i++){
            num ^= nums[i];
        }
        int a = num & (-num); // this line is going to give us bit 1 only where the
first bit is set
….. by using this for loop we are dividing element which have 2nd binary number is
one from the lsb
        for(int i = 0; i < nums.length; i++){
            if((nums[i] & a) == 0){
                ans[0] ^= nums[i];
            }
            else{
                ans[1] ^= nums[i];
            }
        }
        return ans;
    }
```

```
}
```

Special Algorithm (moores voting algorithm):- used to find the number of element greater than n/k times

229. Majority Element II

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

In this algorithm we are making an assumption that let our first element is our element which occurs more than n/3 times and with count 1 if any other element occurs rather than that element we will reduce the count of that element at any time if the count gets zero than it Is not possible for that element to be our majority element so change curr element to nums[i]

Code:-

```
class Solution {
    public List<Integer> majorityElement(int[] nums) {
        int count1 = 0;
        int count2 = 0;
        int major1 = 1000000007;
        int major2 = 1000000007;
        List<Integer> al = new ArrayList<>();
        for(int i = 0; i < nums.length; i++){
            if(nums[i] == major1){
                count1++;
            }
            else if(nums[i] == major2){
                count2++;
            }
            else if(count1 == 0){
                count1 = 1;
                major1 = nums[i];
            }
            else if(count2 == 0){
                count2 = 1;
                major2 = nums[i];
            }
            else{
                count1--;
                count2--;
            }
        }
        count1 = 0;
```

```
        count2 = 0;
        for(int i = 0; i < nums.length; i++){
            if(nums[i] == major1){
                count1++;
            }
            else if(nums[i] == major2){
                count2++;
            }
        }
        if(count1 > nums.length / 3){
            al.add(major1);
        }
        if(count2 > nums.length / 3){
            al.add(major2);
        }
        return al;
    }
}
```

From this code wirte the approach in point wise to unserstand the logic

Q) 3-Sum problem

15. 3Sum

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

Example 1:

Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not matter.

Approach:

There are 3 approaches we have seen first and second one was better and optimal second was using extra hashmap

Approach2:-

i) We have fixed the two pointers I and j and then we were looking for the number \
Nums[i]+nums[j]+nums[k] = 0
Nums[i]+nums[j] = -nums[k]
So we check for -nums[k] in hashmap if we found then added all num[i], nums[j], -nums[k]

Approach3:-

i)firs sorted the array to satisfy the case were no repeated element we should we take

ii)we have fixed I and we were checking at each step for nums[i]+nums[j]+nums[i] = 0

using a while loop

code:

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        int previ = Integer.MIN_VALUE;
        List<List<Integer>> al = new ArrayList<>();
        for(int i = 0; i < nums.length; i++){
            if(nums[i] == previ){
                continue;
            }
            int j = i+1;
            int k = nums.length-1;
            int prevj = Integer.MIN_VALUE;
            int prevk = Integer.MIN_VALUE;
            while(j < k){
                if(prevj == nums[j] && prevk == nums[k]){
                    j++;
                    k--;
                    continue;
                }
                else if(prevj == nums[j] ){
                    j++;
                    continue;
                }
                else if(prevk == nums[k] ){
                    k--;
                    continue;
                }
                int sum = nums[i] + nums[j] + nums[k];
                if(sum == 0){
                    prevj = nums[j];
                    prevk = nums[k];
                    List<Integer> curr = new ArrayList<>();
                    curr.add(nums[i]);
                    curr.add(nums[j]);
```

```
                    curr.add(nums[k]);
                    al.add(curr);
                    j++;
                    k--;
                }
                else if(sum < 0){
                    j++;
                }
                else if(sum > 0){
                    k--;
                }

            }
            previ = nums[i];
        }
        return al;
    }
}
```

Q) largest subarray with 0 sum

Description: Given an array having both positive and negative integers. The task is to compute the length of the largest subarray with sum 0.

Example 1:

Input:
N = 8
A[] = {15,-2,2,-8,1,7,10,23}
Output: 5
Explanation: The largest subarray with
sum 0 will be -2 2 -8 1 7.

We have used prefix sum concetpt here

Code:

```
class GfG
{
   int maxLen(int arr[], int n)
   {
```

```
        // Your code here
        int sum = 0;
        int max_length = 0;
        HashMap<Integer, Integer> hm = new HashMap<>();
        hm.put(0, -1);
        for(int i = 0; i < n; i++){
          sum += arr[i];
          if(hm.containsKey(sum)){
            max_length = Math.max(i-hm.get(sum), max_length);
          }
          else{
            hm.put(sum, i);
          }
        }
        return max_length;
    }
}
```

Que

Q) 4-sum

All sum 2, 3, 4 having three approaches 1st brute force using nested loops 2nd approach using set data stucuture with list containing to store all the unique list

3rd approach using two pointers in 3-sum we have fixed one I pointer and we were moving j and k j = j+1 and k = nums.length-1 in 4 sum similar but we required 4 pointers

Code:

```java
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        Arrays.sort(nums);
        int prev = Integer.MIN_VALUE;
        int n = nums.length;
        List<List<Integer>> al = new ArrayList<>();
        for(int i = 0; i < n; i++){
```

```
            if(prev == nums[i]){
                continue;
            }
            int prev_j = Integer.MIN_VALUE;
            for(int j = i+1; j < n; j++){
                if(prev_j == nums[j]){
                    continue;
                }
                int k = j+1;
                int l = n-1;
                int prev_k = Integer.MIN_VALUE;
                int prev_l = Integer.MIN_VALUE;
                while(k < l){
                    if(prev_k == nums[k] && prev_l == nums[l]){
                        k++;
                        l--;
                        continue;
                    }
                    else if(prev_k == nums[k]){
                        k++;
                        continue;
                    }
                    else if(prev_l == nums[l]){
                        l--;
                        continue;
                    }
                    long sum = (long)nums[i] + (long)nums[j] + (long)nums[k] +
(long)nums[l];
                    if(sum == target){
                        List<Integer> temp = new ArrayList<>();
                        temp.add(nums[i]);
                        temp.add(nums[j]);
                        temp.add(nums[k]);
                        temp.add(nums[l]);
                        prev_k = nums[k];
                        prev_l = nums[l];
                        al.add(temp);
                        k++;
                        l--;
                    }
                    if(sum > target){
                        l--;
                    }
                    else if(sum < target){
                        k++;
                    }
                }
                prev_j = nums[j];
```

```
            }
            prev = nums[i];
        }
        return al;
    }
}
```

Greedy and PriorityQueue new Type pattern

846. Hand of Straights

Alice has some number of cards and she wants to rearrange the cards into groups so that each group is of size groupSize, and consists of groupSize consecutive cards.

Given an integer array hand where hand[i] is the value written on the i^{th} card and an integer groupSize, return true if she can rearrange the cards, or false otherwise.

Key To Solve(solved after watching video):

In this question we were stuck on how to handle the repeating numbers

In this question we are using a priority queue to keep the smallest element at the top as in all group first element will be the smallest. In this pq we have inserted card value with its frequency for calculating frequency we have used hashmap and then we have inserted in priority queue in pairs

Also we have used a queue for storing the element which having the frequency greater than 1

Code:

```
class Solution {

    public class Pair{
        int card;
        int count;
        public Pair(int card, int count){
            this.card = card;
            this.count = count;
        }
    }
    public boolean isNStraightHand(int[] hand, int groupSize) {
        int n = hand.length;
        if(n % groupSize != 0){
            return false;
        }
        HashMap<Integer, Integer> hm = new HashMap<>();
```

```
        PriorityQueue<Pair> pq = new PriorityQueue<>(new Comparator<>(){
            public int compare(Pair o1, Pair o2){
                return Integer.compare(o1.card, o2.card);
            }
        }
        );
        for(int i = 0; i < hand.length; i++){
            hm.put(hand[i], hm.getOrDefault(hand[i], 0)+1);
        }
        for(int num : hm.keySet()){
            pq.add(new Pair(num, hm.get(num)));
        }
        while(!pq.isEmpty()){
            Queue<Pair> q = new LinkedList<>();
            Pair element = pq.remove();
            element.count -= 1;
            if(element.count > 0){
                q.add(element);
            }
            for(int i = 1; i < groupSize; i++){
                if(!pq.isEmpty() && element.card+i == pq.peek().card){
                    Pair currel = pq.remove();
                    currel.count -= 1;
                    if(currel.count > 0){
                        q.add(currel);
                    }
                }
                else{
                    return false;
                }
            }
            while(!q.isEmpty()){
                pq.add(q.poll());
            }
        }
        return true;
    }
}
```

This is also same question:

1296. Divide Array in Sets of K Consecutive Numbers

Given an array of integers nums and a positive integer k, check whether it is possible to divide this array into sets of k consecutive numbers.

Return true *if it is possible*. Otherwise, return false.

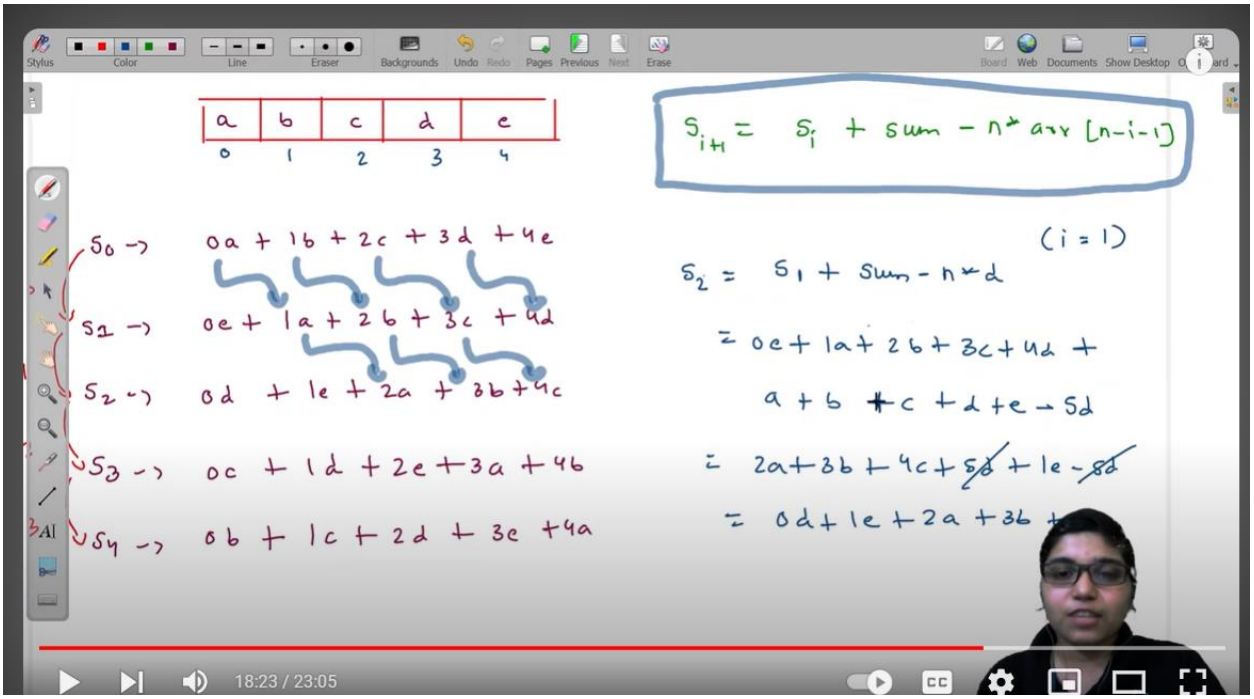
Q) Max sum in the configuration

Given an integer array(0-based indexing) **a** of size **n.** Your task is to **return** the **maximum** value of the sum of **i*a[i]** for all **0<= i <=n-1**, where a**[i]** is the element at index **i** in the array. The only operation allowed is to rotate(clockwise or counterclockwise) the array any number of times.

Example 1:

Input: n = 4, a[] = {8, 3, 1, 2}
Output: 29
Explanation: All the configurations possible by rotating the elements are:
3 1 2 8 here sum is 3*0+1*1+2*2+8*3 = 29
1 2 8 3 here sum is 1*0+2*1+8*2+3*3 = 27
2 8 3 1 here sum is 2*0+8*1+3*2+1*3 = 17
8 3 1 2 here sum is 8*0+3*1+1*2+2*3 = 11, so the maximum sum will be 29.

Idea behind the approach:



Code:

class Solution {

```
    // static int mod = 1000000007;
    long max_sum(int a[], int n) {
        // Your code here
        long sum = 0;
        long si = 0;
        long val;
        for(int i = 0; i < n; i++){
            val = i;
            sum += a[i];
            si += a[i]*val;
        }
        long max = si;
        for(int i = 0; i < n-1; i++){
            long curr = (si+sum)-(a[n-i-1]*(long)n);
            if(curr > max){
                max = curr;
            }
            si = curr;
        }
        return max;
    }
}
```

This concept was used in a daily question

To find the common characters in whole array we can use hashmap for storing the element of first index then at each index create an hashmap and compare the frequency and put the minimum of curr and total frequency if any element do not exists then in the curr but in the main it does then remove it from the hashmap (Note: do consider for the case of concurrent modification exception)

152. Maximum Product Subarray

Given an integer array nums, find a

subarray
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.

Approach:

1)can be solve using brute foce using three loops

2)this question can be solve by using normal two for loops also

3)using prefix and suffix multiplication and also consider the case for if prefix is 0 or suffix is 0

We are iterating from right to left and from left to right because of three cases

- **i) 1, -2, -3, 4, 5 in this case we simply multiply whole array and we will getting the maximum value**
- **ii) 1, -2, 3, 4, 5 in this case we can divide the array in two half before the negative value and after the negative maximum value for this reason we are iterating from left to right and right to left**
- **iii) Also third case is for zero 1,2,3,0,5,6,7,0,8,9,3**
  In this numbers because of zeros we can divide the array in three half so whenever we encounter with zero lets assume that it is a new array

Code:

```
class Solution {
    public int maxProduct(int[] nums) {
        int temp[] = {0,10,10,10,10,10,10,10,10,10,-10,10,10,10,10,10,10,10,10,10,0};
        if(Arrays.equals(nums,temp)){
            return 1000000000;
        }
        int n = nums.length;
        int prefix = 1;
        int suffix = 1;
        int max = Integer.MIN_VALUE;
        for(int i = 0; i < nums.length; i++){
            if(prefix == 0){
                prefix = 1;
            }
            if(suffix == 0){
                suffix = 1;
            }
            prefix *= nums[i];
```

```
            suffix *= nums[n-i-1];
            max = Math.max(max, prefix);
            max = Math.max(suffix, max);
        }
    return max;
    }
}
```

56. Merge Intervals

Solved

Medium

Topics

Companies

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*.

Approach: Solved greedily

Extra: "Assume an array of video watch times, where each segment represents the times a user started and stopped a video, calculate the total number of unique minutes watched" question could be reframe like this

Code:

```
class Solution {
    public int[][] merge(int[][] intervals) {
        List<List<Integer>> al = new ArrayList<>();
        Arrays.sort(intervals, Comparator.comparing(o->o[0]));
        int prevStart = intervals[0][0];
        int prevEnd = intervals[0][1];
        for(int i = 1; i < intervals.length; i++){
            int currStart = intervals[i][0];
            int currEnd = intervals[i][1];
            if(currStart <= prevEnd && currEnd > prevEnd){
                prevEnd = currEnd;
            }
            else if(currStart > prevEnd){
                List<Integer> temp = new ArrayList<>();
                temp.add(prevStart);
                temp.add(prevEnd);
                al.add(temp);
                prevStart = currStart;
```

```
                prevEnd = currEnd;
            }
        }
        List<Integer> temp = new ArrayList<>();
                temp.add(prevStart);
                temp.add(prevEnd);
                al.add(temp);
                System.out.println(al);
        int ans[][] = new int[al.size()][2];
        for(int i = 0; i < ans.length; i++){
            ans[i][0] = al.get(i).get(0);
            ans[i][1] = al.get(i).get(1);
        }
                return ans;
    }
}
```

88. Merge Sorted Array

Solved without using any extra space:

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**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

Example 1:

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.

Code:

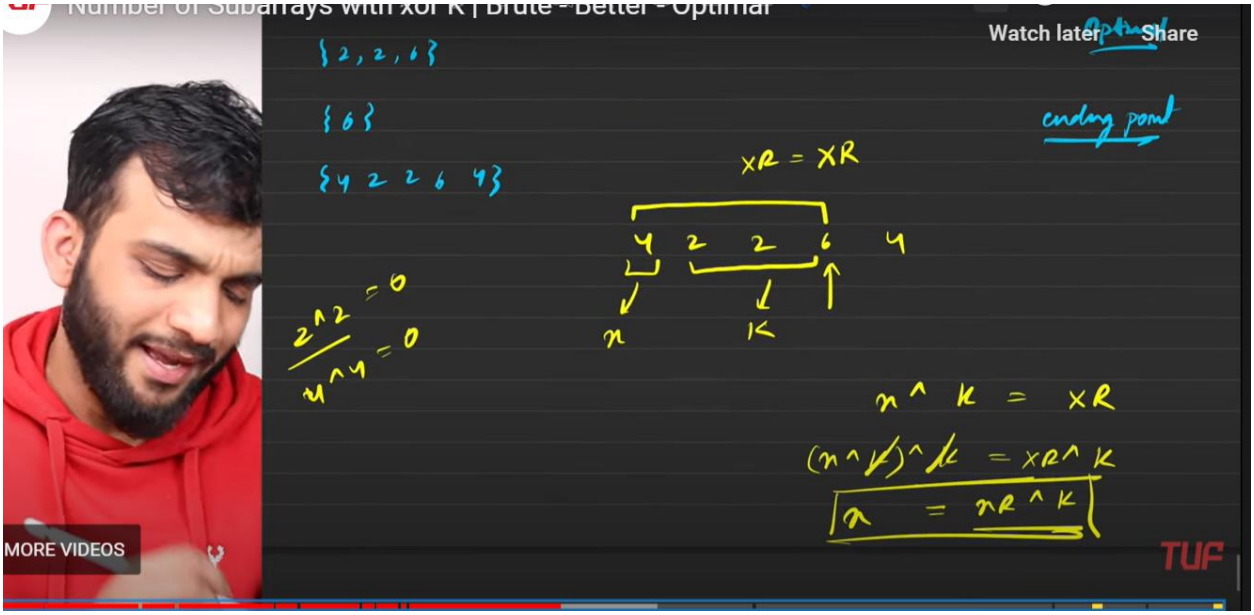
```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int idx = nums1.length-1;
```

```
        int i  = m-1;
        int j = n-1;
        if(j < 0){
            return;
        }
        if(i < 0){
            for(int k = 0; k < nums1.length; k++){
                nums1[k] = nums2[k];
            }
        }
        while(i >= 0 && j >= 0){
            if(nums2[j] >= nums1[i]){
                nums1[idx] = nums2[j];
                j--;
                idx--;
            }
            else if(nums2[j] < nums1[i]){
                nums1[idx] = nums1[i];
                nums1[i] = 0;
                i--;
                idx--;
            }
            // this case for the element if any in the nums2 array is less than the
whole array and we gone out of bound then simply we have to put the contents of nums2
in nums1 from jth index to 0 index i have considered here j for the sake of
simplicity
            if(i < 0 && j >= 0){
                int index = j;
                for(int l = index; l >= 0; l--){
                    nums1[l] = nums2[j];
                    j--;
                }
            }
        }
    }
}
```

Q) Subarray with given XOR

Keep in mind this approach we can apply it to optimize o(n2) subarrays problem

```
A = [4, 2, 2, 6, 4]
 B = 6
```



Lower bound means the largest element which is smaller than given element

34. Find First and Last Position of Element in Sorted Array

Note: By using same approach we can solve for number of occurences of any of the element

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

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

Example 1:

Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]

Solved using upper bound and lower bound for first element used lower bound and for the last element used upper bound and lower bound

Code:

```
class Solution {
    public int findBound(int[] arr, int n, int x, int b){
        Arrays.sort(arr);
        int start = 0;
        int end = n-1;
        int value1 = Integer.MAX_VALUE;
        int value2 = Integer.MIN_VALUE;
        while(start <= end){
            int mid = start + (end-start)/2;
            if(arr[mid] < x){
                start = mid+1;
            }
            else if(arr[mid] > x){
                end = mid-1;
            }
            else{
                if(b==0){
                    end = mid-1;
                    value1 = Math.min(value1, mid);
                }
                else{
                    start = mid+1;
                     value2 = Math.max(value2, mid);
                }
            }
        }
        return b == 0 ? value1 == Integer.MAX_VALUE ? -1 : value1 : value2 ==
Integer.M_VALUE ? -1 : value2;
    }
    public int[] searchRange(int[] nums, int target) {
        int ans[] = new int[2];
        ans[0] = findBound(nums, nums.length, target, 0);
        ans[1] = findBound(nums, nums.length, target, 1);
        return ans;
    }
}
```

But more efficiently just run a for loop and look up for target value and also take a pointer and run till we get same value as target increase it if target is in the given array then update the ans with lower index and upper index else update it with -1

Maximum occured integer

Given **n** integer ranges, the task is to return the **maximum occurring integer** in the given ranges. If more than one such integer exists, return the **smallest** one.
The ranges are in two arrays **l**[] and **r[]**. **l[i]** consists of the starting point of the range and **r[i]** consists of the corresponding endpoint of the range & a maxx which is the **maximum** value of r[].

For example, consider the following ranges.
l[] = {2, 1, 3}, r[] = {5, 3, 9)
Ranges represented by the above arrays are.
[2, 5] = {2, **3**, 4, 5}
[1, 3] = {1, 2, **3**}
[3, 9] = {**3**, 4, 5, 6, 7, 8, 9}
The maximum occurred integer in these ranges is 3.

Examples :

Input: n = 4, l[] = {1,4,3,1}, r[] = {15,8,5,4}, maxx = 15
Output: 4
Explanation: The given ranges are [1,15] [4, 8] [3, 5] [1, 4]. The smallest number that is most common or appears most times in the ranges is 4.

Approaches :

1) **Brute force we can use by iterating on the given range in the array**
2) **Optimize with using array and prefix sum in this approach we have used a array with size maxx+1 and update the lower range as a index in arr with +1 and upper range with -1**

Code:

```
class Solution {

  // Function to find the maximum occurred integer in all ranges.

  // Function to find the maximum occurred integer in all ranges.

  public static int maxOccured(int n, int l[], int r[], int maxx) {

    int arr[] = new int[maxx+2];

    for(int i = 0; i < n; i++){
```

```
            arr[l[i]]++;
            arr[r[i]+1]--;
        }
        int sum = 0;
        for(int i = 0; i < arr.length; i++){
            arr[i] += sum;
            sum = arr[i];
        }
        int index = 0;
        for(int i = 0; i < arr.length; i++){
            if(arr[i] > arr[index]){
                index = i;
            }
        }
        return index;
    }
}
```

Note: sometimes instead of hashmap we can use hasharray

Q)Find Missing And Repeating

Given an unsorted array **Arr** of size **N** of positive integers. **One number 'A'** from set {1, 2,....,N} is missing and **one number 'B'** occurs twice in array. Find these two numbers.

Example 1:

Input:
N = 2
Arr[] = {2, 2}

Output: 2 1

Explanation: Repeating number is 2 and smallest positive missing number is 1.

Approach:

1) **To find the missing and repeating number we can first use brute force with O(n2) approach**
2) **Second method is we can use hashing here er are not going to use hashmap instead we are going to use hashed array but it will be using extra space for optimization**
3) **The third method Is using maths**

i)find the sum of all number lets say it as x now find the sum of n natural number lets,s say it y then take difference of these both we will getting one equation

ii) find the sum of square of all number lets say it as x^2 now find the sum of square of n natural number let's say it y^2 then take difference of these both we will getting one equation

x+y = -4

x^2-y^2 = -24

(x+y)(x-y) = -24

x-y = 6;

solve it we will get our ans for repeating number and another one as not present number

Code:

```
class Solve {
  int[] findTwoElement(int arr[], int n) {
    // code here
    long nSum = (n*(n+1L))/2;
    long nsqrSum = (n*(n+1L)*(2*n+1L))/6;
    long sum = 0;
    long sqrSum = 0;
    for(int i = 0; i < n; i++){
      sum += (long)arr[i];
      sqrSum += (long)arr[i]*(long)arr[i];
```

```
        }
        long xydiff = sum-nSum; // x-y
        long xysqrdiff = sqrSum-nsqrSum; // x^2-y^2
        xysqrdiff = xysqrdiff/xydiff;
        long x = (xysqrdiff+xydiff)/2;
        long y = x-xydiff;
        int ans[] = new int[2];
        ans[0] =(int) x;
        ans[1] = (int)y;
        return ans;
      }
    }
```

Q) find number of inversions means we have to find the number of swaps we need to make to make array sorted

Little modification in merge sort

For counting the numbers of inversion we used one algorithm

Idea behind the algorithm for counting the pairs for which arr1 element is greater than arr2 element

- ⇨ **As array is sorted in ms so if 3 is greater than 2 then it is compulsory that rest of the elements after 3 will be greater than 3 so instead of increasing count by 1 we are taking count += n-l;**
- ⇨ **Same approach used down wards to calculate the number of inversions**

Code:

```
class Solution
{
  static void merge(long arr[], int l, int m, int r, long ans[])
  {
    // Find sizes of two subarrays to be merged
    System.out.println(l+" "+m+" "+r);
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temp arrays
    long L[] = new long[(int)n1];
    long R[] = new long[(int)n2];

    // Copy data to temp arrays
    for (int i = 0; i < n1; ++i)
      L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
      R[j] = arr[m + 1 + j];

    // Merge the temp arrays

    // Initial indices of first and second subarrays
    int i = 0, j = 0;
```

```
    // Initial index of merged subarray array
    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            ans[0] += (long)n1 - (long)i;
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[] if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[] if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
    // Main function that sorts arr[l..r] using
    // merge()
    static void sort(long arr[], int l, int r, long ans[])
    {
        if (l < r) {

            // Find the middle point
            int m = l + (r - l) / 2;

            // Sort first and second halves
            sort(arr, l, m, ans);
            sort(arr, m + 1, r, ans);

            // Merge the sorted halves
            merge(arr, l, m, r, ans);
        }
    }
    static long inversionCount(long arr[], long n)
    {
        // Your Code Here
        long ans[] = new long[1];
        ans[0] = 0L;
        sort(arr, 0, (int)n-1, ans);
        return ans[0];
    }
}
```

523. Continuous Subarray Sum

Note: here we are using a important concept of modulo that if we add any number who is divisible by k to any number then the remainder will be same for both number before adding k and after adding k

Given an integer array nums and an integer k, return true *if* nums *has a* ***good subarray*** *or* false *otherwise*.

A **good subarray** is a subarray where:

- its length is **at least two**, and
- the sum of the elements of the subarray is a multiple of k.

Note that:

- A **subarray** is a contiguous part of the array.
- An integer x is a multiple of k if there exists an integer n such that x = n * k. 0 is **always** a multiple of k.

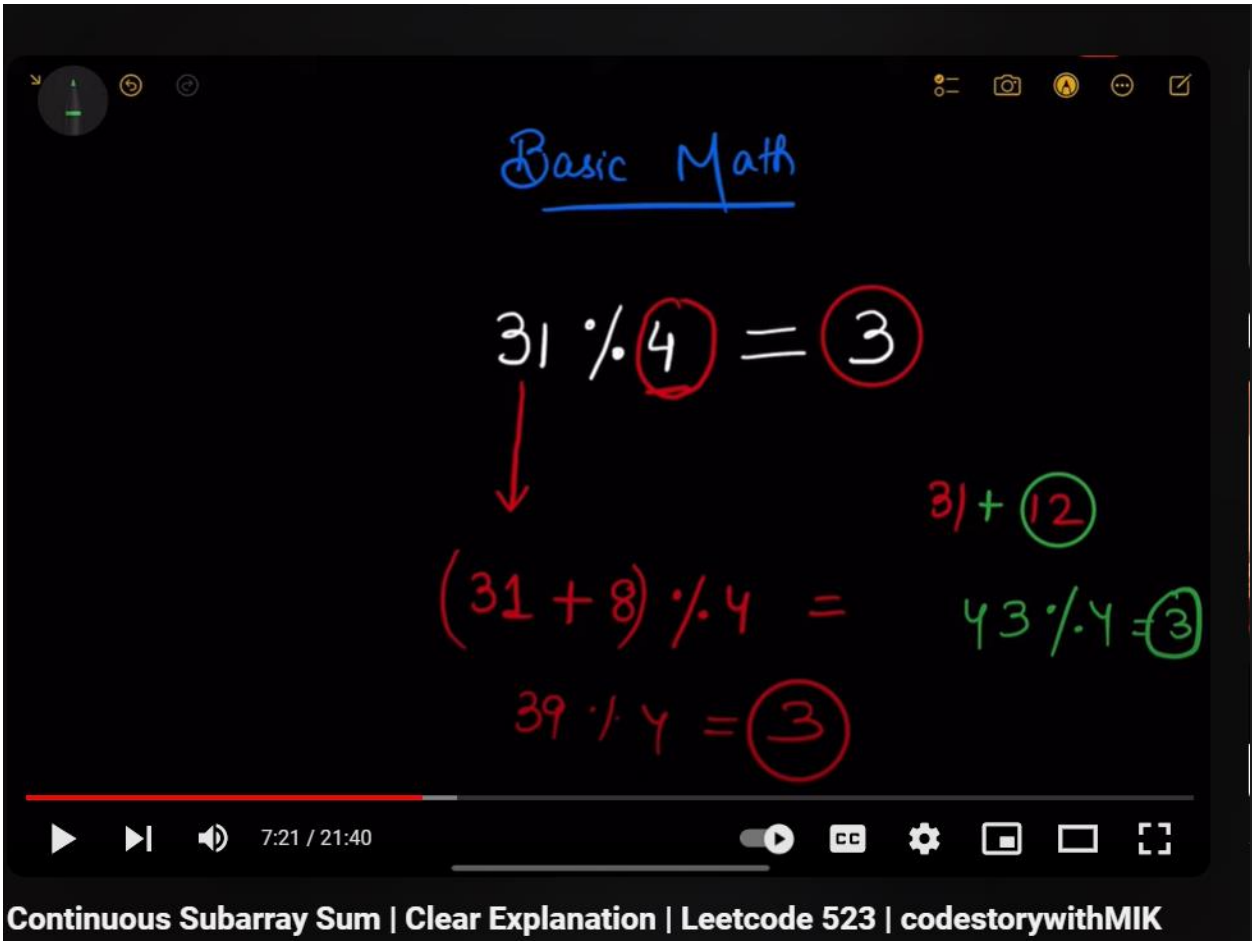
Example 1:

Input: nums = [23,2,4,6,7], k = 6
Output: true
Explanation: [2, 4] is a continuous subarray of size 2 whose elements sum up to 6.

Code:

```
class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        HashMap<Integer, Integer> hm = new HashMap<>();
        hm.put(0, -1);
        int sum = 0;
        for(int i = 0; i < nums.length; i++){
                sum += nums[i];
                int mod = sum % k;
                if(hm.containsKey(mod)){
                    if((i-hm.get(mod)) >= 2)
                    return true;
                }
                else{
                    hm.put(mod, i);
                }
        }
```

```
        return false;
    }
}
```



493. 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].

Example 1:

Input: nums = [1,3,2,3,1]
Output: 2
Explanation: The reverse pairs are:

```
(1, 4) --> nums[1] = 3, nums[4] = 1, 3 > 2 * 1
(3, 4) --> nums[3] = 3, nums[4] = 1, 3 > 2 * 1
```

Approach: in this question also we used the approach similar like count inversions but here we only need to count the reverse pair so before adding it we have just counted the number of reverse pair for sorted array int this question we are first generating whole tree in merge sort then we start counting after counting we merge it

And the algorithm we are using to count is if both half array is sorted and if any number in right half at jth index and it is smaller then the number at ith place in left half by 2*nums[j] then we are just increasing the j until the condition fails for ith index number and then we calculated count = count+j-(mid+1) here we added count if suppose any number in future we get in I if that also satisfies the condition and we know the array is sorted so all the number before j and including j will satisfies the condition that's why we are taking in account of all previous numbers

Code

```
class Solution {
    void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        // System.out.println(l+" "+m+" "+r);
        int n1 = m - l + 1;
        int n2 = r - m;

        // Create temp arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

        // Copy data to temp arrays
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];

        // Merge the temp arrays

        // Initial indices of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
```

```
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of L[] if any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy remaining elements of R[] if any
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
    void count(int l, int m, int r, int[] arr, int[] ans){
        int a = l;
        int b = m+1;
        while(a <= m){
            while(b <= r && arr[a] > (long)2*arr[b]){
                b++;
            }
            ans[0] += (b-(m+1));
            a++;
        }
    }
    // Main function that sorts arr[l..r] using
    // merge()
    void sort(int arr[], int l, int r, int ans[])
    {
        if (l < r) {

            // Find the middle point
            int m = l + (r - l) / 2;

            // Sort first and second halves
            sort(arr, l, m, ans);
            sort(arr, m + 1, r, ans);
            count(l, m, r, arr, ans);
            // Merge the sorted halves
            merge(arr, l, m, r);
        }
```

```
    }
    public int reversePairs(int[] nums) {
        int n = nums.length;
        int ans[] = new int[1];
        sort(nums, 0, n-1, ans);
        // System.out.println(Arrays.toString(nums));
        return ans[0];
    }
}
```

33. Search in Rotated Sorted Array

Example 1:

Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4

Example 2:

Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1

Example 3:

Input: nums = [1], target = 0
Output: -1

Approach : just look at each step which half is sorted and according to that we have to move either in the right half or left half it is also decided whether the element lies in that range or not according to that we moves either in the forward or backward direction

Code:

```
class Solution {
    public int search(int[] nums, int target) {
        int n = nums.length;
        int start = 0;
        int end = n-1;
        while(start <= end){
            System.out.println(start+" "+end);
            int mid = start + (end-start)/2;
            if(nums[mid] == target){
                return mid;
            }
            // checking which partv of the array is sorted either right half or left half if left half is sorted then the first value in left should be <= mid else right half will be sorted
```

```
            if(nums[start] <= nums[mid]){
                // if our target lies in between start and mid
                if(nums[start] <= target && nums[mid] >= target){
                    end = mid-1;
                }
                else{
                    start = mid+1;
                }
            }
            else{
                // if our target lies in between mid and length of array
                if(nums[mid] <= target && target <= nums[n-1]){
                    start = mid+1;
                }
                else{
                    end = mid-1;
                }
            }ng
        }
        return -1;
    }
}
```

(Follow up question if the array contains any repeating numbers then the time compl. Might go to O(n))

Assusme this test case in this case if right and left are same to mid or any of them are same then we will just move start++ and we also checking if the nums[start] is equals to target like linear search

Sorting Steps:

1) **Heap Sort**

   i)convert the array to the binary tree(for imagination not actual binary tree we are using indexes from mid to 0 calling for non leaf nodes);
   ii)for max heap swapping of max element to the root node
   iii)swap the max element with the min element in the tree and putting max element at the end of the array
   iv)again heapifying it

code:

```
class Solution {
    public void heapify(int arr[], int n, int i){
        int largest = i;
```

```
        int left = (2*i)+1;
        int right = (2*i)+2;
        if(left < n && arr[left] > arr[largest]){
            largest = left;
        }
        if(right < n && arr[right] > arr[largest]){
            largest = right;
        }
        if(largest != i){
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;
            heapify(arr, n, largest);
        }
    }
    public void sort(int arr[]){
        int n = arr.length;
        // we are runnig from n/2 to 0 wecause we have to call heapify function for
the non leaf nodes
        // and non leaf nodes are lying before 2*i+2
        for(int i = (n/2); i >= 0; i--){
            heapify(arr, n, i);
        }
        for(int i = n-1; i > 0; i--){
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;a
            heapify(arr, i, 0);
        }
    }
    public int heightChecker(int[] heights) {
        int temp[] = heights.clone();
        sort(heights);
        System.out.println(Arrays.toString(heights));
        int count = 0;
        for(int i = 0; i < heights.length; i++){
            if(heights[i] != temp[i]){
                count++;
            }
        }
        return count;
    }
}
```

Radix sort works on soring of any number from right to left means from lsb to msb of any number it follows stable sorting technique

540. Single Element in a Sorted Array

Solved

Medium

Topics

Companies

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return *the single element that appears only once*.

Your solution must run in O(log n) time and O(1) space.

Example 1:

Input: nums = [1,1,2,3,3,4,4,8,8]
Output: 2

Example 2:

Input: nums = [3,3,7,7,10,11,11]
Output: 10

Approach: in this question we focused on even and odd mid values and also we have to take in the consideration of the edge case of left most and right most position separately

Code:

```
class Solution {
    public int singleNonDuplicate(int[] nums) {
        if(nums.length == 1){
            return nums[0];
        }
        int start = 0;
        int end = nums.length-1;
        int ans = start+(end-start)/2;
        while(start <= end){
            int mid = start+(end-start)/2;
            if(mid+1 > nums.length-1 && nums[mid] != nums[mid-1]){
                return nums[mid];
```

```
            }
            if(mid-1 < 0 && nums[mid] != nums[mid+1]){
                return nums[mid];
            }
            if(mid-1 < 0 || mid+1 > nums.length-1){
                return ans;
            }
            if(mid%2 == 0){
                if(nums[mid] != nums[mid+1]){
                    ans = mid;
                    end = mid-1;
                }
                else{
                    start = mid+1;
                }
            }
            else{
                if(nums[mid] != nums[mid-1]){
                    ans = mid;
                    end = mid-1;
                }
                else{
                    start = mid+1;
                }
            }
        }
        return nums[ans];
    }
}
```