We can simplify the solution by using a cleaner approach to count frequencies and sort the array. Here's a more straightforward implementation:

**Simplified Java Code for Sorting Array Elements by Frequency:**

java

Copy code

import java.util.\*;

public class Main {

public static void main(String[] args) {

int[] arr = {4, 5, 6, 5, 4, 3}; // Example array

System.out.println("Original array: " + Arrays.toString(arr));

System.out.println("Sorted by frequency:");

sortByFrequency(arr);

}

public static void sortByFrequency(int[] arr) {

// Step 1: Count the frequency of each element using a HashMap

Map<Integer, Integer> freqMap = new HashMap<>();

for (int num : arr) {

freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);

}

// Step 2: Convert array to list for easy sorting

List<Integer> list = new ArrayList<>();

for (int num : arr) {

list.add(num);

}

// Step 3: Sort the list based on frequency and value

list.sort((a, b) -> {

int freq1 = freqMap.get(a);

int freq2 = freqMap.get(b);

// If frequencies are different, sort by frequency (higher comes first)

if (freq1 != freq2) {

return freq2 - freq1; // Higher frequency first

}

// If frequencies are the same, sort by value (ascending order)

return a - b;

});

// Step 4: Print the sorted array

for (int num : list) {

System.out.print(num + " ");

}

}

}

**Explanation:**

1. **Step 1**: Use a HashMap to count the frequency of each element in the array.
2. **Step 2**: Convert the array to a list so that it can be sorted easily.
3. **Step 3**: Sort the list using List.sort(). We define a custom comparator that:
   * First sorts by frequency in descending order.
   * If frequencies are the same, it sorts by the element value in ascending order.
4. **Step 4**: Print the sorted list.

**Example Input and Output:**

For input:

java

Copy code

int[] arr = {4, 5, 6, 5, 4, 3};

Output:

text

Copy code

Original array: [4, 5, 6, 5, 4, 3]

Sorted by frequency:

4 4 5 5 3 6

This approach is simple, easy to understand, and avoids unnecessary complexity. It uses Java's built-in sorting mechanisms and a straightforward HashMap to track frequencies.

Ques 2-

import java.util.\*;

public class Main {

public static void main(String[] args) {

System.out.println("Rank of each element:");

int[] arr = {20, 15, 26, 2, 98, 6, 15}; // Added duplicate value (15)

findRank(arr);

}

public static void findRank(int[] arr) {

int n = arr.length;

int[] dummy = new int[n];

// Copy original array to a dummy array

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

dummy[i] = arr[i];

}

// Sort the dummy array

Arrays.sort(dummy);

// HashMap to store ranks, but handling duplicates

HashMap<Integer, Integer> map = new HashMap<>();

int rank = 1; // Starting rank

// Assign ranks to elements in the sorted array, handling duplicates

map.put(dummy[0], rank); // First element gets rank 1

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

// If current element is same as the previous, assign same rank

if (dummy[i] != dummy[i - 1]) {

rank++;

}

map.put(dummy[i], rank);

}

// Print ranks of the original array elements

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

System.out.print(map.get(arr[i]) + " ");

}

}

}

Ques 3-

To solve the problem of finding symmetric pairs in an array of pairs, we can follow this approach:

**Steps:**

1. We need to identify pairs (a, b) and (b, a) in the array, which we call symmetric pairs.
2. A **HashMap** can help to check if we already encountered the reversed pair.
3. For each pair (a, b), we:
   * Check if the reverse pair (b, a) exists in the map.
   * If it exists, it means we have a symmetric pair.
   * If it doesn't exist, add the current pair (a, b) to the map.

**Example:**

For input: [(1,2), (2,1), (3,4), (4,5), (5,4)], the symmetric pairs are (2,1) and (5,4).

**Java Code:**

java

Copy code

import java.util.HashMap;

public class SymmetricPairs {

public static void findSymmetricPairs(int[][] pairs) {

// HashMap to store each pair (a,b)

HashMap<Integer, Integer> map = new HashMap<>();

// Traverse through the array of pairs

for (int i = 0; i < pairs.length; i++) {

int first = pairs[i][0]; // a

int second = pairs[i][1]; // b

// Check if (b,a) exists in the map

if (map.containsKey(second) && map.get(second) == first) {

System.out.println("(" + first + "," + second + ")");

} else {

// Add the pair (a,b) to the map

map.put(first, second);

}

}

}

public static void main(String[] args) {

// Example 1

int[][] pairs1 = { {1, 2}, {2, 1}, {3, 4}, {4, 5}, {5, 4} };

System.out.println("Symmetric Pairs in Example 1:");

findSymmetricPairs(pairs1);

// Example 2

int[][] pairs2 = { {1, 5}, {2, 3}, {4, 2}, {5, 1}, {2, 4} };

System.out.println("Symmetric Pairs in Example 2:");

findSymmetricPairs(pairs2);

}

}

**Explanation:**

1. **HashMap** is used to store pairs as keys (where the first element is the key, and the second is the value).
2. As we iterate through each pair (a, b), we check if the reverse pair (b, a) already exists in the HashMap.
   * If it exists, the pair is symmetric.
   * If not, we add the current pair (a, b) to the map.
3. The pairs are printed only when they are found to be symmetric.

**Example Output:**

Symmetric Pairs in Example 1:

(2,1)

(5,4)

Symmetric Pairs in Example 2:

(5,1)

(2,4)

**Time Complexity:**

* The solution runs in **O(n)** time, where n is the number of pairs, since each operation (insert and lookup) in the HashMap takes constant time on average.

Ques 5- **Rotate array by K elements**

import java.util.\*;

public class Main {

*// Function to Reverse the array*

public static void Reverse(int[] arr, int start, int end) {

while (start <= end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

*// Function to Rotate k elements to left*

public static void Rotateeletoleft(int[] arr, int n, int k) {

*// Reverse first k elements*

Reverse(arr, 0, k - 1);

*// Reverse last n-k elements*

Reverse(arr, k , n - 1);

*// Reverse whole array*

Reverse(arr, 0, n - 1);

}

public static void main(String args[]) {

int[] arr = {1,2,3,4,5,6,7};

int n = 7;

int k = 2;

Rotateeletoleft(arr, n, k);

System.out.print("After Rotating the k elements to left ");

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

System.out.print(arr[i] + " ");

System.out.println();

}

}

**Output:**

After Rotating the k elements to left 3 4 5 6 7 1 2

* **Time Complexity -**O(N) where N is the n**For Rotating Elements to**right

**Step 1:** Reverse the last k elements of the array

**Step 2:** Reverse the first n-k elements of the array.

**Step 3:** Reverse the whole array.

For Eg , arr[]={1,2,3,4,5,6,7} , k=2

umber of elements in an array

**Space Complexity -**O(1) since no extra space is required

public static void Rotateeletoright(int[] arr, int n, int k) {

*// Reverse first n-k elements*

Reverse(arr, 0, n - k - 1);

*// Reverse last k elements*

Reverse(arr, n - k, n - 1);

*// Reverse whole array*

Reverse(arr, 0, n - 1);

}

**Finding Equilibrium index in an array**

**Problem Statement:**Finding Equilibrium index in an array

Given a 0-indexed integer array nums, find the leftmost equilibrium Index.

An equilibrium Index is an index at which sum of elements on its left is equal to the sum of element on its right. That is, nums[0] + nums[1] + ... + nums[equilibriumIndex-1] == nums[equilibriumIndex+1] + nums[equilibriumIndex+2] + ... + nums[nums.length-1].

If equilibriumIndex == 0, the left side sum is considered to be 0. Similarly, if equilibriumIndex == nums.length - 1, the right side sum is considered to be 0.

Return the leftmost equilibrium Index that satisfies the condition, or -1 if there is no such index.

**Example 1:**

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

**Output:** 3

**Explanation:** The sum of the numbers before index 3 is: 2 + 3 + -1 = 4

The sum of the numbers after index 3 is: 4 = 4

**Example 2:**

**Input:** nums = [1,-1,4]

**Output:** 2

**Explanation:** The sum of the numbers before index 2 is: 1 + -1 = 0

The sum of the numbers after index 2 is: 0

**Solution 2: Using Total Sum**

Calculate the sum = total sum of all the integers in the array.

Keep leftSum = 0, rightSum = sum.

leftSum = sum of all the integers to its left

 rightSum = sum of all the integers to it’s right.

At every index i rightSum would be rightSum excluding the current index value.now we will is Check if(leftSum == rightSum)

If yes then return that index else keep moving forward.while moving forward it can be seen that we are considering that current index value to be on left so update the leftSum value.

leftSum = leftSum + nums[i]. If no such index is found return -1.

import java.util.\*;

class Main {

public static int findEquilibriumIdx(int nums[], int n) {

int totalSum = 0;

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

totalSum += nums[i];

}

int leftSum = 0, rightSum = totalSum;

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

rightSum -= nums[i];

if (leftSum == rightSum) {

return i;

}

leftSum += nums[i];

}

return -1;

}

**Check if array is subset of another array**

**Check if array is subset of another array**.

Write a program to find whether an array is a subset of another array or not.

Given arr1[] and arr2[], we need to find whether arr1[] is a subset of arr2[]. An array is called a subset of another if all of its elements are present in the other array.

**Note:**Array elements are assumed to be unique.

**Examples:**

**Example 1:**

**Input:** arr1[]= [1,3,4,5,2]

arr2[]= [2,4,3,1,7,5,15]

**Output:** arr1[] is a subset of arr2[]

**Example 2:**

**Input:** arr1[]= [1,3,4,5,2]

arr2[]= [4,5,2]

**Output:** arr1[] is not a subset of arr2[]

**Example 3:**

**Input:** arr1[]= [1,3,4,5,2]

arr2[]= [11,12,13,15,16]

**Output:** arr1[] is not a subset of arr2[]

**Approach:**

Convert arr2[] into a HashSet so that we can perform quick lookups to check if each element of arr1[] exists in arr2[].

Iterate through each element of arr1[] and check if it's present in the HashSet created from arr2[].

If all elements of arr1[] are found in the HashSet, then arr1[] is a subset of arr2[]. Otherwise, it's not.

import java.util.\*;

public class Main {

public static void main(String[] args) {

// Example 1

int[] arr1 = {1, 3, 4, 5, 2};

int[] arr2 = {2, 4, 3, 1, 7, 5, 15};

checkSubset(arr1, arr2); // Output: arr1[] is a subset of arr2[]

// Example 2

int[] arr3 = {1, 3, 4, 5, 2};

int[] arr4 = {4, 5, 2};

checkSubset(arr3, arr4); // Output: arr1[] is not a subset of arr2[]

// Example 3

int[] arr5 = {1, 3, 4, 5, 2};

int[] arr6 = {11, 12, 13, 15, 16};

checkSubset(arr5, arr6); // Output: arr1[] is not a subset of arr2[]

}

public static void checkSubset(int[] arr1, int[] arr2) {

// Step 1: Convert arr2 into a HashSet for fast lookups

Set<Integer> set = new HashSet<>();

for (int num : arr2) {

set.add(num);

}

// Step 2: Check if all elements of arr1 are present in arr2 (HashSet)

boolean isSubset = true;

for (int num : arr1) {

if (!set.contains(num)) {

isSubset = false;

break;

}

}

// Step 3: Print the result

if (isSubset) {

System.out.println("arr1[] is a subset of arr2[]");

} else {

System.out.println("arr1[] is not a subset of arr2[]");

}

}

}  
  
Ques-- **Q2) Given an array of integers and a sum, the task is to count all subsets of given array with sum equal to given sum.**

**Input :**

The first line of input contains an integer T denoting the number of test cases. Then T test cases follow. Each test case contains an integer n denoting the size of the array. The next line contains n space separated integers forming the array. The last line contains the sum.

**Output :**

Count all the subsets of given array with sum equal to given sum.

NOTE: Since result can be very large, print the value modulo 109+7.

**Constraints :**  
  
1<=T<=100   
  
1<=n<=103

1<=a[i]<=103

1<=sum<=103

**Example :**

**Input :**

2

6

2 3 5 6 8 10

10

5

1 2 3 4 5

10

**Output :**

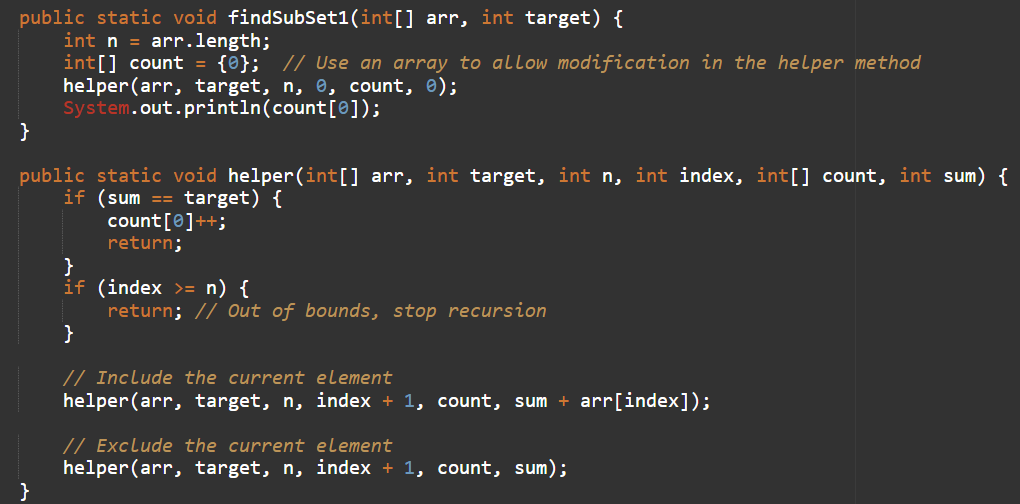
3

3

**Explanation :**

Testcase 1: possible subsets : (2,3,5) , (2,8) and (10)

Testcase 2: possible subsets : (1,2,3,4) , (2,3,5) and (1,4,5)



Q1) Given an array of integers where every element appears even number of times except one element which appears odd number of times, write a program to find that odd occurring element in O(log n) time.

The equal elements must appear in pairs in the array but there cannot be more than two consecutive occurrences of an element.

For example :

3 2 3 2

It doesn't have equal elements appear in pairs

7 1 1 2 2 2 3 3

It contains three consecutive instances of an element.

5 2 2 3 1 1

It is valid and the odd occurring element present in it is 3.

Enter only valid inputs.

Sample Input :

5 2 2 3 1 1

Sample Output : 3

**public static int findOddOccurringElement(int[] arr) {**

**int low = 0;**

**int high = arr.length - 1;**

**// Binary search logic**

**while (low < high) {**

**int mid = low + (high - low) / 2;**

**// Ensure that mid is even to compare pairs**

**if (mid % 2 != 0) {**

**mid--;**

**}**

**// If the element at mid and the next element form a pair, move right**

**if (arr[mid] == arr[mid + 1]) {**

**low = mid + 2;**

**} else {**

**high = mid;**

**}**

**}**

**// When low == high, we have found the odd occurring element**

**return arr[low];**

**}  
Another Approach-**

**// Function to find the odd occurring element in O(n) time using XOR**

**public static int findOddOccurringElement(int[] arr) {**

**int result = 0;**

**// XOR all elements in the array**

**for (int num : arr) {**

**result ^= num;**

**}**

**return result;**

**}**

**Ques-** Q3) Before the outbreak of corona virus to the world, a meeting happened in a room in Wuhan. A person who attended that meeting had COVID-19 and no one in the room knew about it! So everyone started shaking hands with everyone else in the room as a gesture of respect and after meeting unfortunately every one got infected! Given the fact that any two persons shake hand exactly once, Can you tell the total count of handshakes happened in that meeting?

Input Format :

The first line contains the number of test cases T, T lines follow.

Each line then contains an integer N, the total number of people attended that meeting.

Output Format :

Print the number of handshakes for each test-case in a new line.

Constraints :

1 <= T <= 1000

0 < N < 106

Sample Input :

2

1

2

Output :

0

1

import java.util.Scanner;

public class Handshakes {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Read the number of test cases

int T = sc.nextInt();

// Loop over each test case

while (T-- > 0) {

// Read the number of people

int N = sc.nextInt();

// Calculate the number of handshakes using the formula N \* (N - 1) / 2

int handshakes = (N \* (N - 1)) / 2;

// Output the result for this test case

System.out.println(handshakes);

}

sc.close();

}

}

Ques- Q4) For enhancing the book reading, school distributed story books to students as part of the Children’s day celebrations. To increase the reading habit, the class teacher decided to exchange the books every weeks so that everyone will have a different book to read. She wants to know how many possible exchanges are possible.

Find the number of possible exchanges, if the books are exchanged so that every student will receive a different book.

Constraints 1<= N <= 1000000

Input Format

Input contains one line with N, indicates the number of books and number of students.

Output Format

Output the answer modulo 100000007.

Sample Input : 4

Sample Output : 9

The number of such derangements of N items can be calculated using the following recursive formula:

D(n)=(n−1)×(D(n−1)+D(n−2))D(n) = (n - 1) \times (D(n - 1) + D(n - 2))D(n)=(n−1)×(D(n−1)+D(n−2))

Where:

* D(n)D(n)D(n) is the number of derangements of nnn items.
* The base cases are:
  + D(1)=0D(1) = 0D(1)=0 (since 1 item cannot be rearranged to satisfy the condition)
  + D(2)=1D(2) = 1D(2)=1 (since 2 items can be swapped)

Since NNN can be as large as 1,000,0001,000,0001,000,000, we need an efficient approach to compute the number of derangements modulo 100000007100000007100000007.

**import java.util.Scanner;**

**public class Derangements {**

**private static final int MOD = 100000007;**

**// Function to calculate the number of derangements modulo MOD**

**public static long derangement(int N) {**

**// Base cases**

**if (N == 1) return 0; // D(1) = 0**

**if (N == 2) return 1; // D(2) = 1**

**// Initialize variables to store previous two results**

**long prev2 = 0; // D(1)**

**long prev1 = 1; // D(2)**

**long current = 0;**

**// Iterate to calculate D(n) from D(3) to D(N)**

**for (int i = 3; i <= N; i++) {**

**current = (i - 1) \* (prev1 + prev2) % MOD;**

**prev2 = prev1;**

**prev1 = current;**

**}**

**return prev1;**

**}**

**}**

**Ques-** You are given a string A and you have to find the number of different sub-strings of the string A which are fake palindromes.

Note: 1. Palindrome: A string is called a palindrome if you reverse the string yet the order of letters remains the same.

For example, MADAM. 2. Fake Palindrome: A string is called as a fake palindrome if any of its permutations is a palindrome. For example, AAC is fake palindrome, but ACD is not.

3. Sub-string: A sub-string is a contiguous sequence (non-empty) of characters within a string.

4. Two sub-strings are considered same if their starting indices and ending indices are equal.

Input Format: First line contains a string S

Output Format: Print a single integer (number of fake palindrome sub-strings).

Constraints: 1 <= |S| <= 2 \* 105

The string will contain only Upper case 'A' to 'Z'

Sample Input 1: ABAB

Sample Output 1: 7

Explanation: The fake palindrome for the string ABAB are A, B, A, B, ABA, BAB, ABAB.

Sample Input 2: AAA

Sample output 2: 6

Explanation: The fake palindrome for the string AAA are A, A, A, AA, AA, AAA

### Solution Strategy:

1. Use a **prefix bitmask** where each bit represents whether the count of a character is odd or even up to that point in the string.
2. Traverse through the string and update the bitmask.
3. Use a hashmap (or array) to store how many times each bitmask has occurred so far.
4. For each prefix, check if the current bitmask is already in the map (this means we found a substring with the same character count pattern). Also, check for each possible single-bit difference to handle cases where one character can appear an odd number of times.

import java.util.HashMap;

import java.util.Map;

public class FakePalindromeSubstrings {

public static void main(String[] args) {

String S = "ABAB"; // Replace with input string

System.out.println(countFakePalindromes(S));

}

public static int countFakePalindromes(String S) {

int n = S.length();

int mask = 0;

int fakePalindromeCount = 0;

// Map to store the number of times each mask appears

Map<Integer, Integer> maskCount = new HashMap<>();

maskCount.put(0, 1); // Initial condition: empty prefix mask

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

// Update mask for the current character

int charIndex = S.charAt(i) - 'A';

mask ^= (1 << charIndex); // Toggle the bit for the current character

// If the same mask has appeared before, it means the substring between these two positions is a fake palindrome

fakePalindromeCount += maskCount.getOrDefault(mask, 0);

// Check for single-bit differences (allowing one character to appear an odd number of times)

for (int j = 0; j < 26; j++) {

int alteredMask = mask ^ (1 << j);

fakePalindromeCount += maskCount.getOrDefault(alteredMask, 0);

}

// Increment the count for this mask in the map

maskCount.put(mask, maskCount.getOrDefault(mask, 0) + 1);

}

return fakePalindromeCount;

}

}