Count Inversions

Given an array of integers. Find the Inversion Count in the array.

***Inversion Count**:* For an array, inversion count indicates how far (or close) the array is from being sorted. If array is already sorted then the inversion count is 0. If an array is sorted in the reverse order then the inversion count is the maximum.
Formally, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

Example 1:

```
Input: N = 5, arr[] = {2, 4, 1, 3, 5}
Output: 3
Explanation: The sequence 2, 4, 1, 3, 5
has three inversions (2, 1), (4, 1), (4, 3).
```

METHOD 1 (Simple)

- **Approach:** Traverse through the array, and for every index, find the number of smaller elements on its right side of the array. This can be done using a nested loop. Sum up the counts for all index in the array and print the sum.
- **Algorithm:**

    0. Traverse through the array from start to end
    1. For every element, find the count of elements smaller than the current number up to that index using another loop.
    2. Sum up the count of inversion for every index.
    3. Print the count of inversions.
- **Implementation:**

```
static int getInvCount(int[] arr, int n)

{

    int inv_count = 0;

    for (int i = 0; i < n - 1; i++)

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

            if (arr[i] > arr[j])

                inv_count++;

    return inv_count;

}
```

- **Complexity Analysis:**
    - **Time Complexity:** O(n^2), Two nested loops are needed to traverse the array from start to end, so the Time complexity is O(n^2)
    - **Space** Complexity**:**O(1), No extra space is required.

METHOD 2(Enhance Merge Sort)

Approach:
Suppose the number of inversions in the left half and right half of the array (let be inv1 and inv2); what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is - the inversions that need to be counted during the merge step. Therefore, to get the total number of inversions that needs to be added are the number of inversions in the left subarray, right subarray, and merge().

- **Algorithm:**
    0. The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.
    1. Create a function merge that counts the number of inversions when two halves of the array are merged, create two indices i and j, i is the index for the first half, and j is an index of the second half. if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].
    2. Create a recursive function to divide the array into halves and find the answer by summing the number of inversions is the first half, the number of inversion in the second half and the number of inversions by merging the two.
    3. The base case of recursion is when there is only one element in the given half.
    4. Print the answer

```
public class GFG {

  private static int mergeAndCount(int[] arr, int l,  int m, int r)

  {

    // Left subarray

    int[] left = Arrays.copyOfRange(arr, l, m + 1);

    // Right subarray

    int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);

    int i = 0, j = 0, k = l, swaps = 0;


    while (i < left.length && j < right.length) {

      if (left[i] <= right[j])

        arr[k++] = left[i++];

      else {

        arr[k++] = right[j++];

        swaps += (m + 1) - (l + i);
```

```
            }
        }
        while (i < left.length)
            arr[k++] = left[i++];
        while (j < right.length)
            arr[k++] = right[j++];
        return swaps;
    }
    // Merge sort function
    private static int mergeSortAndCount(int[] arr, int l,  int r)
    {
        // Keeps track of the inversion count at a
        // particular node of the recursion tree
        int count = 0;
        if (l < r) {
            int m = (l + r) / 2;
            // Total inversion count = left subarray count
            // + right subarray count + merge count

            // Left subarray count
            count += mergeSortAndCount(arr, l, m);
            // Right subarray count
            count += mergeSortAndCount(arr, m + 1, r);
            // Merge count
            count += mergeAndCount(arr, l, m, r);
        }
        return count;
    }
```

Complexity Analysis:

- **Time Complexity:** O(n log n), The algorithm used is divide and conquer, So in each level, one full array traversal is needed, and there are log n levels, so the time complexity is O(n log n).
- **Space** Complexity**:** O(n), Temporary array.

Find Missing And Repeating

Given an unsorted array of size n. Array elements are in the range of 1 to n. One number from set {1, 2, ...n} is missing and one number occurs twice in the array. Find these two numbers.

Examples:

Input: arr[] = {3, 1, 3}
Output: Missing = 2, Repeating = 3
Explanation: In the array, 2 is missing and 3 occurs twice

Input: arr[] = {4, 3, 6, 2, 1, 1}
Output: Missing = 5, Repeating = 1

Method 1 (Use Sorting)
Approach:

- Sort the input array.
- Traverse the array and check for missing and repeating.

Time Complexity: O(nLogn)

Method 2 (Use count array)
Approach:

- Create a temp array temp[] of size n with all initial values as 0.
- Traverse the input array arr[], and do following for each arr[i]
    - if(temp[arr[i]] == 0) temp[arr[i]] = 1;
    - if(temp[arr[i]] == 1) output "arr[i]" //repeating
- Traverse temp[] and output the array element having value as 0 (This is the missing element)

Time Complexity: O(n)
Auxiliary Space: O(n)

Method 3 (Use elements as Index and mark the visited places)
Approach:
Traverse the array. While traversing, use the absolute value of every element as an index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.

Time Complexity: O(n)

```
static void printTwoElements(int arr[], int size)
  {
     int i;
     System.out.print("The repeating element is ");

     for (i = 0; i < size; i++) {
```

```
        int abs_val = Math.abs(arr[i]);
        if (arr[abs_val - 1] > 0)
            arr[abs_val - 1] = -arr[abs_val - 1];
        else
            System.out.println(abs_val);
    }

    System.out.print("and the missing element is ");
    for (i = 0; i < size; i++) {
        if (arr[i] > 0)
            System.out.println(i + 1);
    }
}
```

Method 4 (Make two equations using sum and sum of squares)
Approach:

- Let x be the missing and y be the repeating element.
- Let N is the size of array.
- Get the sum of all numbers using formula **S = N(N+1)/2**
- Get the sum of square of all numbers using formula **Sum_Sq = N(N+1)(2N+1)/6**
- Iterate through a loop from i=1....N
- **S -= A[i]**
- **Sum_Sq -= (A[i]*A[i])**
- It will give two equations
  x-y = S - (1)
  x^2 - y^2 = Sum_sq
  x+ y = (Sum_sq/S) - (2)

Time Complexity: O(n)

```
static Vector<Integer> repeatedNumber(int[] a)
  {

    BigInteger n=BigInteger.valueOf(a.length);

    BigInteger s=BigInteger.valueOf(0);
    BigInteger ss=BigInteger.valueOf(0);

    for(int x : a)
```

```
        {
            s=  s.add(BigInteger.valueOf(x));
            ss= ss.add(BigInteger.valueOf(x).multiply(BigInteger.valueOf(x)));
        }


        BigInteger as= n.multiply(n.add(BigInteger.valueOf(1))).divide(BigInteger.valueOf(2));
        BigInteger ass=
as.multiply(BigInteger.valueOf(2).multiply(n).add(BigInteger.valueOf(1))).divide(BigInteger.valu
eOf(3));


        BigInteger sub=as.subtract(s);
        BigInteger add=(ass.subtract(ss)).divide(sub);
        //(ass-ss)/sub;


        int b = sub.add(add).divide(BigInteger.valueOf(2)).intValue();
        //(sub+add)/2;
        int A = BigInteger.valueOf(b).subtract(sub).intValue();
        Vector<Integer> ans = new Vector<>();
        ans.add(A);
        ans.add(b);
        return ans;
    }
```

Method 5 (Use XOR)

Approach:

- Let x and y be the desired output elements.
- Calculate XOR of all the array elements.

xor1 = arr[0]^arr[1]^arr[2].....arr[n-1]

- XOR the result with all numbers from 1 to n

xor1 = xor1^1^2^.....^n

In the result *xor1*, all elements would nullify each other except x and y. All the bits that are set in *xor1* will be set in either x or y. So if we take any set bit (We have chosen the rightmost set bit in code) of *xor1* and divide the elements of the array in two sets – one set of elements with the same bit set and other set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in first set, we will get x, and by doing same in other set we will get y.

Time Complexity: O(n)

```
int[] findTwoElement(int arr[], int n) {
      int x=0;
      for(int i=0;i<n;i++)
      {
         x=x^arr[i];
         x=x^(i+1);
      }
      int pos=x&(-x);
      int repeat=0,missing=0;

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

         if((arr[i]&pos)!=0)
         {
            missing= missing^arr[i];
         }

         if(((i+1)&pos)!=0)
         {
             missing= missing^(i+1);
         }
      }
     repeat=missing^x;
     boolean flag=true;
```

```
  for(int i=0;i<n;i++)
  {
      if(repeat==arr[i])
      {
          flag=false;
          break;
      }
  }
  if(flag)
  {
      int t=repeat;
      repeat=missing;
      missing=t;
  }
   int[] res=new int[2];
   res[0]=repeat;
   res[1]=missing;
   return res;
}
```

Find the Duplicate Number

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

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

Example 1:

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

Output: 2
```

Example 2:

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

Output: 3
```

Approach 1: Sort

Time Complexity: $O(n\log n)$

Space Complexity: $O(n)$

Intuition

In an unsorted array, duplicate elements may be scattered across the array. However, in a sorted array, duplicate numbers will be next to each other.

```
class Solution {
    public int findDuplicate(int[] nums) {
        Arrays.sort(nums);
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] == nums[i-1])
                return nums[i];
        }

        return -1;
    }
```

```
}
```

Approach 2: Set

Intuition

As we traverse the array, we need a way to "remember" values that we've seen. If we come across a number that we've seen before, we've found the duplicate. An efficient way to record the seen values is by adding each number to a set as we iterate over the nums array.

Time Complexity: $O(n)$

Space Complexity: $O(n)$

```
class Solution {
   public int findDuplicate(int[] nums) {
      Set<Integer> seen = new HashSet<Integer>();
      for (int num : nums) {
         if (seen.contains(num))
            return num;
         seen.add(num);
      }
      return -1;
   }
}
```

Approach 3: Negative Marking

Time Complexity: $O(n)$

Space Complexity: $O(1)$

Intuition

There are $n + 1$ positive numbers in the array (nums) (all in the range $[1, n]$). Since the array only contains positive integers, we can track each number (num) that has been seen before by flipping the sign of the number located at index $|\text{num}|$, where $||$ denotes absolute value.

For example, if the input array is $[1, 3, 3, 2]$, then for 1, flip the number at index 1, making the array $[1,-3,3,2]$. Next, for -3 flip the number at index 3, making the array $[1,-3,3,-2]$. Finally, when

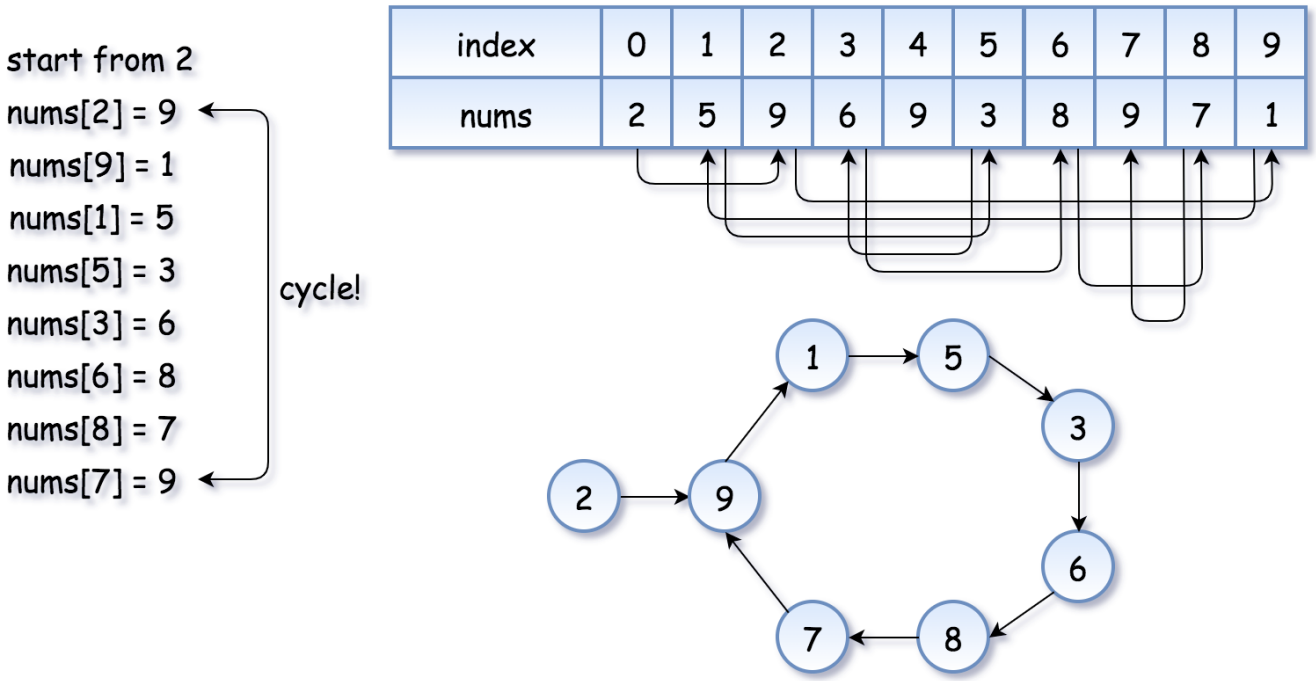
we reach the second 33, we'll notice that $\text{nums}[3]$ is already negative, indicating that 33 has been seen before and hence is the duplicate number.

```
class Solution {

    public int findDuplicate(int[] nums) {

        int duplicate = -1;

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

            int cur = Math.abs(nums[i]);

            if (nums[cur] < 0) {

                duplicate = cur;

                break;

            }

            nums[cur] *= -1;

        }



        // Restore numbers

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

            nums[i] = Math.abs(nums[i]);



        return duplicate;

    }

}
```

Approach 4: Floyd's Tortoise and Hare (Cycle Detection)

Time Complexity: $\mathbf{O(n)}$

Space Complexity: $\mathbf{O(1)}$




```
class Solution {
    public int findDuplicate(int[] nums) {

        // Find the intersection point of the two runners.
        int tortoise = nums[0];
        int hare = nums[0];

        do {
            tortoise = nums[tortoise];
            hare = nums[nums[hare]];
        } while (tortoise != hare);
```

```
        // Find the "entrance" to the cycle.
        tortoise = nums[0];

        while (tortoise != hare) {
            tortoise = nums[tortoise];
            hare = nums[hare];
        }

        return hare;
    }
}
```

Merge Intervals

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

Example 1:

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].
```

A **simple approach** is to start from the first interval and compare it with all other intervals for overlapping, if it overlaps with any other interval, then remove the other interval from the list and merge the other into the first interval. Repeat the same steps for remaining intervals after first. This approach cannot be implemented in better than O(n^2) time.

Approach 1

```
class Solution {
  public int[][] merge(int[][] intervals) {
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
    LinkedList<int[]> merged = new LinkedList<>();
    for (int[] interval : intervals) {
      // if the list of merged intervals is empty or if the current
      // interval does not overlap with the previous, simply append it.
      if (merged.isEmpty() || merged.getLast()[1] < interval[0]) {
        merged.add(interval);
      }
      // otherwise, there is overlap, so we merge the current and previous intervals.
      else {
        merged.getLast()[1] = Math.max(merged.getLast()[1], interval[1]);
      }
    }
    return merged.toArray(new int[merged.size()][]);
  }                          }
```

Approach 2 without Collections

```
class Solution {
    public int[][] merge(int[][] intervals) {
       Arrays.sort(intervals,(a,b)->Integer.compare(a[0],b[0]));
      int n=intervals.length;
        int j=0;
        int l=intervals[0][0];
        int r=intervals[0][1];
        for(int i=1;i<n;i++)
        {
            if(r>=intervals[i][0]&&r<=intervals[i][1])  {
                r=intervals[i][1];
            }
            else if(r<intervals[i][0])  {
                intervals[j][0]=l;
                intervals[j][1]=r;
                l=intervals[i][0];
                r=intervals[i][1];
                j++;
            }
        }
         intervals[j][0]=l;
        intervals[j][1]=r;
        int[][] res=new int[j+1][2];
        for(int i=0;i<j+1;i++) {
            res[i][0]=intervals[i][0];
            res[i][1]=intervals[i][1];
        }
        return res;
    }
}
```

Merge Sorted Array

Given two sorted arrays **arr1[]** and **arr2[]** of sizes **n** and **m** in non-decreasing order. Merge them in sorted order without using any extra space. Modify arr1 so that it contains the first N elements and modify arr2 so that it contains the last M elements.

Input:

n = 4, arr1[] = [1 3 5 7]

m = 5, arr2[] = [0 2 6 8 9]

Output:

arr1[] = [0 1 2 3]

arr2[] = [5 6 7 8 9]

Explanation:

After merging the two

non-decreasing arrays, we get,

0 1 2 3 5 6 7 8 9.

Naïve Approach: time = O(m+n) ; space=O(m+n);

```
class Solution {
   public void merge(int[] nums1, int m, int[] nums2, int n) {
int a[]=new int[m+n];
      int i=0,j=0,l=0;
      while(i<m&&j<n)
      {
         if(nums1[i]<=nums2[j])
         {
            a[l++]=nums1[i++];


         }
         else
         {
```

```
            a[l++]=nums2[j++];
          }
        }
        while(i<m)
        {
          a[l++]=nums1[i++];
        }
        while(j<n)
        {
          a[l++]=nums2[j++];
        }
        for(int k=0;k<a.length;k++)
        {
          nums1[k]=a[k];
        }
}
}
```

Better Approach: time = (m*n) space=O(1)

It uses insertion sort technique

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
      int l=0;
      int r=0;
        while(l<m&&r<n)
        {

          if(nums1[l]>nums2[r])
          {
```

```
            int t=nums1[l];
            nums1[l]=nums2[r];
            nums2[r]=t;
            insertElement(nums2);

        }
        l++;
    }
    while(r<n)
    {
        nums1[l++]=nums2[r++];
    }

}
public void insertElement(int[] arr)
{
    int e=arr[0];
    for(int i=1;i<arr.length;i++)
    {
        if(e>arr[i])
        {
            arr[i-1]=arr[i];
        }
        else
        {
            arr[i-1]=e;
            return;
        }
```

```
        }
        arr[arr.length-1]=e;
    }
}
```

Efficient Approach: time =O(Nlog(N)) space: O(1)

It uses gap method

```
class Solution
{
    //Function to merge the arrays.
    public static void merge(long arr1[], long arr2[], int n, int m)
    {

        int gap=n+m;
        for( gap=nextGap(gap);gap>0;gap=nextGap(gap))
        {

            for(int i=0;i<(m+n);i++)
            {
                int j=i+gap;
                //j reach out of the m+n
                if(j>=m+n)
                break;
                //if i and j in the first array
                if(i<n&&j<n)
                {
                    if(arr1[i]>arr1[j])
                    {
                        swap(arr1,arr1,i,j);
```

```
            }
        }
        //if i in first array and j in second array
        else if(i<n&&j>=n)
        {
            if(arr1[i]>arr2[j-n])
            swap(arr1,arr2,i,j-n);
        }
        //if i and j both are in second array
        else if(i>=n&&j>=n)
        {
            if(arr2[i-n]>arr2[j-n])
            {
                swap(arr2,arr2,i-n,j-n);
            }
        }
      }
   }
}
public static void swap(long[] a,long[] b,int i,int j)
{
    long l=a[i];
    a[i]=b[j];
    b[j]=l;
}
public static int nextGap(int gap)
{
    if(gap==0||gap==1)
```

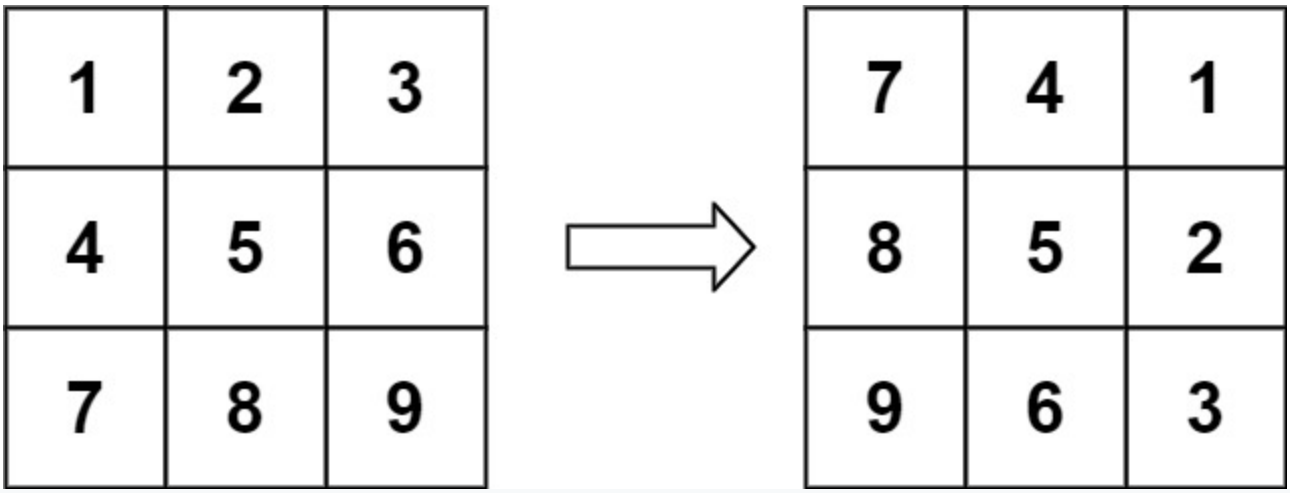
```
        {
            return 0;
        }
        //gap%2 is to get the celing value
        return gap/2+gap%2;
    }
}
```

Rotate Image

You are given an `n x n` 2D `matrix` representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

Example 1:




```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]

Output: [[7,4,1],[8,5,2],[9,6,3]]

Steps:
```

1. Transpose the given matrix
2. Reverse the elements in all the rows

```
class Solution {
    public void rotate(int[][] matrix) {
        int n=matrix.length;
        transport(matrix);

        for(int i=0;i<n;i++)
        {
            int l=0, r=n-1;
            while(l<r)
            {
```

```
            int t=matrix[i][l];
            matrix[i][l]=matrix[i][r];
            matrix[i][r]=t;
            l++;
            r--;
        }
    }
  }
  public void transport(int[][] matrix)
  {
    int n=matrix.length;
    for(int i=0;i<n;i++)
    {
      for(int j=i+1;j<n;j++)
      {
        int t=matrix[i][j];
    matrix[i][j]=matrix[j][i];
    matrix[j][i]=t;
      }
    }
  }

}
```