**LC#136 :SINGLE NUMBER**

**Given a non-empty array of integers nums, every element appears *twice* except for one. Find that single one.You must implement a solution with a linear runtime complexity and use only constant extra space.**

**Input: nums = [2,2,1]**

**Output: 1**

**APPROACH 1:BRUTE (O(N^2))**

**->Check every element count and return the element where its count is 1**

**CODE:**

class Solution {

public int singleNumber(int[] nums) {

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

int count = 0;

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

if (nums[i] == nums[j]) count++;

}

if (count == 1) return nums[i];

}

return -1; // should never happen

}

}

**APPROACH2:O(NLOGN)**

**->sort the array**

**-> check if the previous element and the current element are matching , if not return that element**

import java.util.Arrays;

class Solution {

public int singleNumber(int[] nums) {

Arrays.sort(nums);

for (int i = 0; i < nums.length - 1; i += 2) {

if (nums[i] != nums[i + 1]) {

return nums[i];

}

}

return nums[nums.length - 1];

}

}

**APPROACH 4:HASH MAP (SIMILAR TO COUNTING WITH REDUCED TC => TC/SC:O(N)**

import java.util.HashMap;

class Solution {

public int singleNumber(int[] nums) {

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

for (int num : nums) {

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

}

for (int key : map.keySet()) {

if (map.get(key) == 1) return key;

}

return -1;}}

**APPROACH 5:OPTIMAL AND THEY HAVE ASKED THIS IN THE QUESTION**

**->loop through the elements and perform xor operation return the resultant as there is guaranteed single element**

class Solution {

public int singleNumber(int[] nums) {

int result = 0;

for (int num : nums) {

result ^= num;

}

return result;

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**LC#137:Single number II**

**Given an integer array nums where every element appears three times except for one, which appears exactly once. *Find the single element and return it*.**

**You must implement a solution with a linear runtime complexity and use only constant extra space.**

**Input: nums = [2,2,3,2] Output: 3**

**APPROACH1:HASH MAP-> return where count==1**

class Solution {

    public int singleNumber(int[] nums) {

        Map<Integer , Integer> hm = new HashMap()

        for(int a : nums)

            hm.put(a , hm.getOrDefault(a , 0)+1);

        for(int i : hm.keySet())

            if(hm.get(i)==1)

                return i;

        return -1;}}**TC:O(N log M)+O(M) SC :O(N) ❌ M-SIZE OF MAP**

**APPROACH 2: USING RIGHT SHIFT(>>)**

**->loop the elements and take the binary value of the current num and add It sum it to a result value**

**Process:[2,2,3,2]->**

**2 → 0010**

**2 → 0010**

**3 → 0011**

**2 → 0010**

**T--->0041 =>(sum of 1) &1**

**After that check that value %3 ->** if not zero **, right shift ,perform or (|) to the resultant**

**CODE:**

class Solution {

    public int singleNumber(int[] nums) {

        int res = 0 ;

        for(int i=0;i<32;i++)//32bit rep

        {   int temp = 0;

            for(int j:nums)

            {

                int sum =(1&(j>>i));//extract last digit

                temp = temp+sum;

            }

            if(temp%3!=0)

            {

                res = res |(1<<i);// set i-th bit in result

            }

        }

        return res;

    }}

**APPROACH 3: SORTING**

* **Sort the array**
* **start from 0 compare the current and the next (skip for the first only)**
* **not same return nums[i]**

**CODE:**

class Solution {

public int singleNumber(int[] nums) {

Arrays.sort(nums);

for (int i = 0; i < nums.length**; i += 3**) {

if (i + 1 >= nums.length || nums[i] != nums[i + 1]) {

return nums[i];

}

}

return -1;

}

}

**APPROACH 4:BIT INVERSION**

* <https://www.youtube.com/watch?v=5Bb2nqA40JY>
* **Concept: two variables ones and twos**

**->store ones with xor of current val and & of inv(twos)**

**->store twos with xor of current val and & of inv(ones)**

**-> in this way it stores and deletes automatically whenever it detects such that ones will contain only the element which is occurring once in the array**

class Solution {

    public int singleNumber(int[] nums) {

        int ones = 0 , twos = 0 ;

        for(int n :nums)

        {

            ones = (ones^n) &~twos;

            twos = (twos^n)& ~ones;

        }

        return ones;   }}

**LC#389 :FIND DIFFERENCE**

**Input: s = "abcd", t = "abcde"**

**Output: "e" [ 'e' is the letter that was added.]**

**APPROACH 1 : sort and compare SC:O(n log n) TC:o(1)**

class Solution {

    public char findTheDifference(String s, String t) {

        char s\_arr[]= s.toCharArray();

        char t\_arr[] = t.toCharArray();

        Arrays.sort(s\_arr);

        Arrays.sort(t\_arr);

        for(int i = 0 ; i<s\_arr.length;i++)

        {

            if(s\_arr[i] != t\_arr[i])

                return t\_arr[i];

        }

        return t\_arr[t\_arr.length - 1];

    }

}

**APPROACH 2: FREQUENCY COUNTING USES ONLY 26 SPACES AS IT IS LOWERCASE ONLY**

**->calculate the count of elements in s .**

**-> decrement the same count when in t , the solution is where the count of the current char <0 => return it .**

class Solution {

public char findTheDifference(String s, String t) {

int[] freq = new int[26];

for (char c : s.toCharArray()) {

freq[c - 'a']++;

}

for (char c : t.toCharArray()) {

freq[c - 'a']--;

if (freq[c - 'a'] < 0) {

return c;

}

}

return ' '; // never happens

}

}

**APPROACH 3: Time: O(n) ,Space: O(1)**

class Solution {

public char findTheDifference(String s, String t) {

int sum = 0;

for (char c : t.toCharArray()) sum += c;

for (char c : s.toCharArray()) sum -= c;

return (char) sum;

}

}

**APPROACH 4:PREFERRED (XOR)**

**Time: O(n) ,Space: O(1)**

class Solution {

public char findTheDifference(String s, String t) {

char res = 0;

for (char c : s.toCharArray()) res ^= c;

for (char c : t.toCharArray()) res ^= c;

return res;

}

}

**APPROACH 4:HASH MAP WORKS FOR ANY CHARACTER TYPE**

**SPACE & TIME o(N)**

class Solution {

    public char findTheDifference(String s, String t) {

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

        for(char sc : s.toCharArray())

            hm.put(sc,hm.getOrDefault(sc,0)+1);

        for(char tc : t.toCharArray())

        {

            hm.put(tc, Character hm.getOrDefault(tc , 0)-1);

            if(hm.get(tc)<0)

                return tc;

        }

        return ' ';

    }

}

**LC#268:MISSING NUMBER:**

**1)Brute Force (Check Each Number)**

* **Loop through numbers from 0 to n.**
* **For each number, check if it exists in the array.**
* **The first one not found is the missing number.**

class Solution {

public int missingNumber(int[] nums) {

int n = nums.length;

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

boolean found = false;

for (int num : nums) {

if (num == i) {

found = true;

break;

}

}

if (!found) return i;

}

return -1;

}

}

✅ TC: **O(n²)** (nested loops)  
✅ SC: **O(1)**

**APPOACH 2: SORT THE ARRAY AND COMPUTE N2->N LOG N**

**✅ TC: O(n log n) (sorting)  
✅ SC: O(1) or O(log n) (depending on sort implementation)**

class Solution {

public int missingNumber(int[] nums) {

Arrays.sort(nums);

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

if (nums[i] != i) return i;

}

return nums.length;

}

}

**APPROACH 3:MATH FORMULA**

class Solution {

public int missingNumber(int[] nums) {

int n = nums.length;

int expectedSum = n \* (n + 1) / 2;

int actualSum = 0;

for (int num : nums)

actualSum += num;

return expectedSum - actualSum;

}

}

**APPROACH 4:XOR**

class Solution {

    public int missingNumber(int[] nums) {

        int res = 0, i = 0;

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

            res = res ^ i ^ nums[i];//i=0 to n remain will be there

        return res ^ nums.length;//edge case last number n is not in loop(AOE)

    }

}

**APPROACH 5:HASH SET**

class Solution {

public int missingNumber(int[] nums) {

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

for (int num : nums) {

set.add(num); // Add all numbers into the set

}

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

if (!set.contains(i)) return i; // Check from 0..n }

return -1; // should never happen

}

}

**APPROACH 6 : HASH MAP APPROACH 7:HASHING INTO BOOL ARRAY AND FINDING**

class Solution {

public int missingNumber(int[] nums) {

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

// Store all numbers as keys

for (int num : nums) {

map.put(num, true);

}

// Check from 0..n

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

if (!map.containsKey(i)) return i;

}

return -1; // should never happen

}

}