<https://leetcode.com/problems/single-number/description/>

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.

**Example 1:**

Input: nums = [2,2,1]

Output: 1

**Example 2:**

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

Output: 4

**Example 3:**

Input: nums = [1]

Output: 1

**Constraints:**

1 <= nums.length <= 3 \* 104

-3 \* 104 <= nums[i] <= 3 \* 104

Each element in the array appears twice except for one element which appears only once.

**Attempt 1: 2023-09-16**

**Solution 1: Hash Table (10 min)**

class Solution {

public int singleNumber(int[] nums) {

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

for(int num : nums) {

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

}

for(Map.Entry<Integer, Integer> entry : freq.entrySet()) {

if(entry.getValue() == 1) {

return entry.getKey();

}

}

return -1;

}

}

Time complexity : O(N)

Space complexity : O(N)

**Refer to**

<https://leetcode.com/problems/single-number/solutions/1771720/c-easy-solutions-sorting-xor-maps-or-frequency-array/>

##### **METHOD 1 : USING MAPS (NOT USING CONSTANT SPACE )**

The question states that we have to find an element in the array with frequency=1. So, the first idea that pops in the mind is to store the frequency of each element in a map (or a frequency array) and then traverse that map/array and return the element with frequency=1.

Map the given array's elements to their frequency. ( KEY : ELEMENT , VALUE : FREQUENCY )

Traverse that map and return the key whose value =1.

##### **CODE :**

class Solution {

public:

int singleNumber(vector<int>& nums) {

unordered\_map<int,int> a;

for(auto x: nums)

a[x]++;

for(auto z:a)

if(z.second==1)

return z.first;

return -1;

}

};

**TC: O(N)**

**SC: O(N)**

**Solution 2: Hash Set (10 min)**

class Solution {

public int singleNumber(int[] nums) {

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

for(int num : nums) {

// Returns true if this set contains the specified element

if(!set.contains(num)) {

set.add(num);

} else {

set.remove(num);

}

}

return set.iterator().next();

}

}

Time complexity : O(N)

Space complexity : O(N)

**Refer to**

<https://leetcode.wang/leetcode-136-Single-Number.html>

# **解法一**

题目要求线性复杂度内实现，并且要求没有额外空间。首先我们考虑假如没有空间复杂度的限制。

这其实就只需要统计每个数字出现的次数，很容易想到去用 HashMap 。

遍历一次数组，第一次遇到就将对应的 key 置为 1。第二次遇到就拿到 key 对应的 value 然后进行加 1 再存入。最后只需要寻找 value 是 1 的 key 就可以了。

利用 HashMap 统计字符个数已经用过很多次了，比如 [30 题](https://leetcode.wang/leetCode-30-Substring-with-Concatenation-of-All-Words.html)、[49 题](https://leetcode.wang/leetCode-49-Group-Anagrams.html) 等等，最重要的好处就是可以在 O(1) 下取得之前的元素，从而使得题目的时间复杂度达到 O(n)。

当然，注意到这个题目每个数字出现的次数要么是 1 次，要么是 2 次，所以我们也可以用一个 HashSet ，在第一次遇到就加到 Set 中，第二次遇到就把当前元素从 Set 中移除。这样遍历一遍后，Set 中剩下的元素就是我们要找的那个落单的元素了。

public int singleNumber(int[] nums) {

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

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

if (!set.contains(nums[i])) {

set.add(nums[i]);

} else {

set.remove(nums[i]);

}

}

return set.iterator().next();

}

当然，上边的解法空间复杂度是 O(n)，怎么用 O(1) 的空间复杂度解决上边的问题呢？

想了很久，双指针，利用已确定元素的空间，等等的思想都考虑了，始终想不到解法，然后看了官方的 [Solution](https://leetcode.com/problems/single-number/solution/) ，下边分享一下。

**Solution 3: Sorting (10 min)**

class Solution {

public int singleNumber(int[] nums) {

Arrays.sort(nums);

int n = nums.length;

// Because we check every thing as a pair, the only remain

// single one will always be at a even index after sorting,

// when we check pair by pair [i - 1, i] for i between [1, n),

// if we find a mismatch of a pair, the remain single one

// will be always at i - 1

for(int i = 1; i < n; i += 2) {

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

return nums[i - 1];

}

}

// Handle corner case as last element is single one

// e.g nums = [1,1,2]

return nums[n - 1];

}

}

Time complexity : O(N\*logN)

Space complexity : O(1)

**Refer to**

<https://leetcode.com/problems/single-number/solutions/1771720/c-easy-solutions-sorting-xor-maps-or-frequency-array/>

##### **METHOD 2 : USING SORTING (USING CONSTANT SPACE )**

As explained above , we do the following :

Sort the array.

Traverse the array and check if one of the adjacent elements is equal to the current element or not.

If yes , move ahead. Else return the current element.

##### **CODE :**

class Solution {

public:

int singleNumber(vector<int>& nums) {

sort(nums.begin(),nums.end());

for(int i=1;i<nums.size();i+=2)

{

if(nums[i]!=nums[i-1])

return nums[i-1];

}

return nums[nums.size()-1];

}

};

**TC: O(NlogN)**

**SC: O(1)**

**Solution 4: XOR (10 min)**

class Solution {

public int singleNumber(int[] nums) {

int result = 0;

for(int num : nums) {

result ^= num;

}

return result;

}

}

Time complexity : O(N)

Space complexity : O(1)

**Refer to**

<https://leetcode.com/problems/single-number/solutions/1771720/c-easy-solutions-sorting-xor-maps-or-frequency-array/>

##### **METHOD 3 : USING BITWISE XOR OPERATOR (USING CONSTANT SPACE )**

To use this approach you first need to understand about Bitwise XOR operator. Most of us who have a background in physics ( highschool level ) , are aware of the LOGIC GATES. One of such gates is the XOR Gate :According to this gate , the output is true , only if both the inputs are of opposite kind .That is ,**A B Y0 0 00 1 11 0 11 1 0**

We apply the extended version of this gate in our bitwise XOR operator. If we do "a^b" , it means that we are applying the XOR gate on the 2 numbers in a bitwise fashion ( on each of the corresponding bits of the numbers).Similarly , if we observe ,

***A^A=0***

***A^B^A=B***

***(A^A^B) = (B^A^A) = (A^B^A) = B*** This shows that position doesn't matter.

Similarly , if we see , ***a^a^a......... (even times)=0 and a^a^a........(odd times)=a***

Google It for more details.

We apply the above observations :

Traverse the array and take the Bitwise XOR of each element.

Return the value.

**Why does this work ??**

Because , the elements with frequency=2 will result in 0. And then the only element with frequency=1 will generate the answer.

class Solution {

public:

int singleNumber(vector<int>& nums) {

int ans=0;

for(auto x:nums)

ans^=x;

return ans;

}

};

**TC: O(N)**

**SC: O(1)**

**Refer to**

<https://leetcode.wang/leetcode-136-Single-Number.html>

# **解法三 异或**

还记得位操作中的异或吗？计算规则如下。

0 ⊕ 0 = 0

1 ⊕ 1 = 0

0 ⊕ 1 = 1

1 ⊕ 0 = 1

总结起来就是相同为零，不同为一。

根据上边的规则，可以推导出一些性质

0 ⊕ a = a

a ⊕ a = 0

此外异或满足交换律以及结合律。

所以对于之前的例子 a b a b c c d ，如果我们把给定的数字相互异或会发生什么呢？

a ⊕ b ⊕ a ⊕ b ⊕ c ⊕ c ⊕ d

= ( a ⊕ a ) ⊕ ( b ⊕ b ) ⊕ ( c ⊕ c ) ⊕ d

= 0 ⊕ 0 ⊕ 0 ⊕ d

= d

是的，答案就这样出来了，我妈妈问我为什么要跪着。。。

java 里的异或是 ^ 操作符，初始值可以给一个 0。

public int singleNumber(int[] nums) {

int ans = 0;

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

ans ^= nums[i];

}

return ans;

}

**Solution 5: Math (10 min)**

class Solution {

public int singleNumber(int[] nums) {

int sum = 0;

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

for(int num : nums) {

sum += num;

set.add(num);

}

int sum1 = 0;

for(int i : set) {

sum1 += i;

}

return sum1 \* 2 - sum;

}

}

Time complexity : O(N)

Space complexity : O(1)

**Refer to**

<https://leetcode.wang/leetcode-136-Single-Number.html>

# **解法二 数学推导**

假设我们的数字是 a b a b c c d

怎么求出 d 呢？

只需要把出现过的数字加起来乘以 2 ，然后减去之前的数字和就可以了。

什么意思呢？

上边的例子出现过的数字就是 a b c d ，加起来乘以二就是 2 \* ( a + b + c + d)，之前的数字和就是 a + b + a + b + c + c + d 。

2 \* ( a + b + c + d) - (a + b + a + b + c + c + d)，然后结果是不是就是 d 了。。。。。。

看完这个解法我只能说 tql。。。

找出现过什么数字，我们只需要一个 Set 去重就可以了。

public int singleNumber(int[] nums) {

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

int sum = 0;//之前的数字和

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

set.add(nums[i]);

sum += nums[i];

}

int sumMul = 0;//出现过的数字和

for (int n : set) {

sumMul += n;

}

sumMul = sumMul \* 2;

return sumMul - sum;

}