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

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.

**Example 1:**

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

Output: 3

**Example 2:**

Input: nums = [0,1,0,1,0,1,99]

Output: 99

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -231 <= nums[i] <= 231 - 1
* Each element in nums appears exactly **three times** except for one element which appears **once**.

**Attempt 1: 2023-11-10**

**Solution 1: Bitwise (120 min)**

class Solution {

public int singleNumber(int[] nums) {

int result = 0;

// Default as 32 bit counter to guarantee all numbers between

// [1, 3 \* 10^4] under control since these numbers won't exceed

// 32 bitwise representation

for(int i = 0; i < 32; i++) {

int count = 0;

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

// >>> is logical shift right, it simply moves everything

// to the right and fills in from the left with 0s

if((nums[j] >>> i & 1) == 1) {

count++;

}

}

// 如果所有数字都出现了 3 次，那么每一列的 1 的个数就一定是 3 的倍数。

// 之所以有的列1的个数不是 3 的倍数，就是因为只出现了 1 次的数贡献出了 1。

// 所以所有不是 3 的倍数的列写 1，其他列写 0 ，就找到了这个出现 1 次的数。

if(count % 3 != 0) {

// The 1 << i means shift 1 to ith digit index, use '|' or bit

// operation to set that column to 1 (所以所有不是 3 的倍数的列写 1)

result = result | 1 << i;

}

}

return result;

}

}

Time Complexity: O(N)

Space Complexity: O(1)

**Solution 2: General way (600 min)**

class Solution {

public int singleNumber(int[] nums) {

int x1 = 0, x2 = 0, mask = 0;

for(int i : nums) {

x2 ^= x1 & i;

x1 ^= i;

mask = ~(x1 & x2);

x2 &= mask;

x1 &= mask;

}

// Since p = 1, in binary form p = '01', then p1 = 1, so we should return x1.

// If p = 2, in binary form p = '10', then p2 = 1, and we should return x2.

// Or alternatively we can simply return (x1 | x2).

return x1;

}

}

Time Complexity: O(N)

Space Complexity: O(1)

**Detailed explanation and generalization of the bitwise operation method for single numbers**

**Refer to**

<https://leetcode.com/problems/single-number-ii/solutions/43295/detailed-explanation-and-generalization-of-the-bitwise-operation-method-for-single-numbers/>

I -- Statement of our problem

"Given an array of integers, every element appears k (k > 1) times except for one, which appears p times (p >= 1, p % k != 0). Find that single one."

II -- Special case with 1-bit numbers

As others pointed out, in order to apply the bitwise operations, we should rethink how integers are represented in computers -- by bits. To start, let's consider only one bit for now. Suppose we have an array of **1-bit** numbers (which can only be 0 or 1), we'd like to count the number of 1's in the array such that whenever the counted number of 1 reaches a certain value, say k, the count returns to zero and starts over (in case you are curious, this k will be the same as the one in the problem statement above). To keep track of how many 1's we have encountered so far, we need a counter. Suppose the counter has m bits in binary form: xm, ..., x1 (from most significant bit to least significant bit). We can conclude at least the following four properties of the counter:

1. There is an initial state of the counter, which for simplicity is zero;
2. For each input from the array, if we hit a 0, the counter should remain unchanged;
3. For each input from the array, if we hit a 1, the counter should increase by one;
4. In order to cover k counts, we require 2^m >= k, which implies m >= logk.

Here is the key part: how each bit in the counter (x1 to xm) changes as we are scanning the array. Note we are prompted to use bitwise operations. In order to satisfy the second property, recall what bitwise operations will not change the operand if the other operand is 0? Yes, you got it: x = x | 0 and x = x ^ 0.

Okay, we have an expression now: x = x | i or x = x ^ i, where i is the scanned element from the array. Which one is better? We don't know yet. So, let's just do the actual counting.

At the beginning, all bits of the counter is initialized to zero, i.e., xm = 0, ..., x1 = 0. Since we are gonna choose bitwise operations that guarantee all bits of the counter remain unchanged if we hit 0's, the counter will be 0 until we hit the first 1 in the array. After we hit the first 1, we got: xm = 0, ...,x2 = 0, x1 = 1. Let's continue until we hit the second 1, after which we have: xm = 0, ..., x2 = 1, x1 = 0. Note that x1 changed from 1 to 0. For x1 = x1 | i, after the second count, x1 will still be 1. So it's clear we should use x1 = x1 ^ i. What about x2, ..., xm? The idea is to find the condition under which x2, ..., xm will change their values. Take x2 as an example. If we hit a 1 and need to change the value of x2, what must be the value of x1 right before we do the change? The answer is: x1 must be 1 otherwise we shouldn't change x2 because changing x1 from 0 to 1 will do the job. So x2 will change value only if x1 and i are both 1, or mathematically, x2 = x2 ^ (x1 & i). Similarly xm will change value only when xm-1, ..., x1 and i are all 1: xm = xm ^ (xm-1 & ... & x1 & i). Bingo, we've found the bitwise operations!

However, you may notice that the bitwise operations found above will count from 0 until 2^m - 1, instead of k. If k < 2^m - 1, we need some "cutting" mechanism to reinitialize the counter to 0 when the count reaches k. To this end, we apply bitwise **AND** to xm,..., x1 with some variable called mask, i.e., xm = xm & mask, ..., x1 = x1 & mask. If we can make sure that mask will be 0 only when the count reaches k and be 1 for all other count cases, then we are done. How do we achieve that? Try to think what distinguishes the case with k count from all other count cases. Yes, it's the count of 1's! For each count, we have unique values for each bit of the counter, which can be regarded as its state. If we write k in its binary form: km,..., k1, we can construct mask as follows:

mask = ~(y1 & y2 & ... & ym), where yj = xj if kj = 1, and yj = ~xj if kj = 0 (j = 1 to m).

Let's do some examples:

k = 3: k1 = 1, k2 = 1, mask = ~(x1 & x2);

k = 5: k1 = 1, k2 = 0, k3 = 1, mask = ~(x1 & ~x2 & x3);

In summary, our algorithm will go like this (nums is the input array):

for (int i : nums) {

xm ^= (xm-1 & ... & x1 & i);

xm-1 ^= (xm-2 & ... & x1 & i);

.....

x1 ^= i;

mask = ~(y1 & y2 & ... & ym) where yj = xj if kj = 1, and yj = ~xj if kj = 0 (j = 1 to m).

xm &= mask;

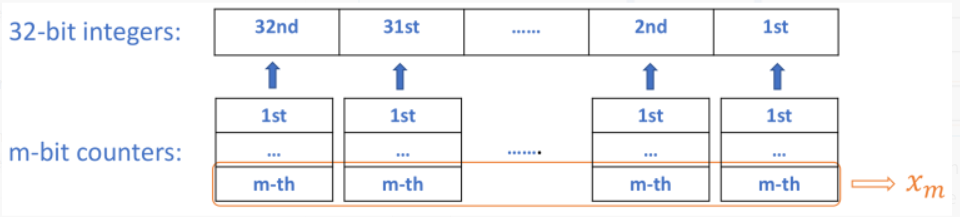
......

x1 &= mask;

}

III -- General case with 32-bit numbers

Now it's time to generalize our results from 1-bit number case to 32-bit integers. One straightforward way would be creating 32 counters for each bit in the integer. You've probably already seen this in other posted [solutions](https://discuss.leetcode.com/topic/455/constant-space-solution/4). However, if we take advantage of bitwise operations, we may be able to manage all the 32 counters "collectively". By saying "collectively", we mean using m **32-bit** integers instead of 32 **m-bit** counters, where m is the minimum integer that satisfies m >= logk. The reason is that bitwise operations apply only to each bit so operations on different bits are independent of each other (kind obvious, right?). This allows us to group the corresponding bits of the 32 counters into one 32-bit integer. Here is a schematic diagram showing how this is done.



The top row is the 32-bit integer, where for each bit, we have a corresponding m-bit counter (shown by the column below the upward arrow). Since bitwise operations on each of the 32 bits are independent of each other, we can group, say the m-th bit of all counters, into one 32-bit number (shown by the orange box). All bits in this 32-bit number (denoted as xm) will follow the same bitwise operations. Since each counter has m bits, we end up with m 32-bit numbers, which correspond to x1, ..., xm defined in part II, but now they are 32-bit integers instead of 1-bit numbers. Therefore, in the algorithm developed above, we just need to regard x1 to xm as 32-bit integers instead of 1-bit numbers. Everything else will be the same and we are done. Easy, hum?

IV -- What to return

The last thing is what value we should return, or equivalently which one of x1 to xm will equal the single element. To get the correct answer, we need to understand what the m 32-bit integers x1 to xm represent. Take x1 as an example. x1 has 32 bits and let's label them as r (r = 1 to 32). After we are done scanning the input array, the value for the r-th bit of x1 will be determined by the r-th bit of all the elements in the array (more specifically, suppose the total count of 1 for the r-th bit of all the elements in the array is q, q' = q % k and in its binary form: q'm,...,q'1, then by definition the r-th bit of x1 will be equal to q'1). Now you can ask yourself this question: what does it imply if the r-th bit of x1 is 1?

The answer is to find what can contribute to this 1. Will an element that appears k times contribute? No. Why? Because for an element to contribute, it has to satisfy at least two conditions at the same time: the r-th bit of this element is 1 and the number of appearance of this 1 is not an integer multiple of k. The first condition is trivial. The second comes from the fact that whenever the number of 1 hit is k, the counter will go back to zero, which means the corresponding bit in x1 will be reset to 0. For an element that appears k times, it's impossible to meet these two conditions simultaneously so it won't contribute. At last, only the single element which appears p (p % k != 0) times will contribute. If p > k, then the first k \* [p/k] ([p/k]denotes the integer part of p/k) single elements won't contribute either. So we can always set p' = p % k and say the single element appears effectively p' times.

Let's write p' in its binary form: p'm, ..., p'1 (note that p' < k, so it will fit into m bits). Here I **claim the condition** for xj to equal the single element is p'j = 1 (j = 1 to m), with a quick proof given below.

If the r-th bit of xj is 1, we can safely say the r-th bit of the single element is also 1 (otherwise nothing can make the r-th bit of xj to be 1). We are left to prove that if the r-th bit of xj is 0, then the r-th bit of the single element can only be 0. Just suppose in this case the r-th bit of the single element is 1, let's see what will happen. At the end of the scan, this 1 will be counted p' times. By definition the r-th bit of xj will be equal to p'j, which is 1. This contradicts with the presumption that the r-th bit of xj is 0. Therefore we conclude the r-th bit of xj will always be the same as the r-th bit of the single number as long as p'j = 1. Since this is true for all bits in xj (i.e., true for r = 1 to 32), we conclude xj will equal the single element as long as p'j = 1.

So now it's clear what we should return. Just express p' = p % k in its binary form and return any of the corresponding xj as long as p'j = 1. In total, the algorithm will run in O(n \* logk) time and O(logk) space.

**Side note**: There is a general formula relating each bit of xj to p'j and each bit of the single number s, which is given by (xj)\_r = s\_r & p'j, with (xj)\_r and s\_r denoting respectively the r-th bit of xj and the single number s. From this formula, it's easy to see that (xj)\_r = s\_r if p'j = 1, that is, xj = s as long as p'j = 1, as shown above. Furthermore, we have (xj)\_r = 0 if p'j = 0, regardless of the value of the single number, that is, xj = 0 as long as p'j = 0. So in summary we obtain: xj = s if p'j = 1, and xj = 0 if p'j = 0. This implies the expression (x1 | x2 | ... | xm) will also be evaluated to the single number s, since the expression will essentially take the OR operations of the single number with itself and some 0s, which boils down to the single number eventually.

V -- Quick examples

Here is a list of few quick examples to show how the algorithm works (you can easily come up with other examples):

* k = 2, p = 1
* k is 2, then m = 1, we need only one 32-bit integer (x1) as the counter. And 2^m = k so we do not even need a mask! A complete java program will look like:

public int singleNumber(int[] nums) {

int x1 = 0;

for (int i : nums) {

x1 ^= i;

}

return x1;

}

* k = 3, p = 1
* k is 3, then m = 2, we need two 32-bit integers(x2, x1) as the counter. And 2^m > k so we do need a mask. Write k in its binary form: k = '11', then k1 = 1, k2 = 1, so we have mask = ~(x1 & x2). A complete java program will look like:

public int singleNumber(int[] nums) {

int x1 = 0, x2 = 0, mask = 0;

for (int i : nums) {

x2 ^= x1 & i;

x1 ^= i;

mask = ~(x1 & x2);

x2 &= mask;

x1 &= mask;

}

return x1; // Since p = 1, in binary form p = '01', then p1 = 1, so we should return x1.

// If p = 2, in binary form p = '10', then p2 = 1, and we should return x2.

// Or alternatively we can simply return (x1 | x2).

}

* k = 5, p = 3
* k is 5, then m = 3, we need three 32-bit integers(x3, x2, x1) as the counter. And 2^m > k so we need a mask. Write k in its binary form: k = '101', then k1 = 1, k2 = 0, k3 = 1, so we have mask = ~(x1 & ~x2 & x3). A complete java program will look like:

public int singleNumber(int[] nums) {

int x1 = 0, x2 = 0, x3 = 0, mask = 0;

for (int i : nums) {

x3 ^= x2 & x1 & i;

x2 ^= x1 & i;

x1 ^= i;

mask = ~(x1 & ~x2 & x3);

x3 &= mask;

x2 &= mask;

x1 &= mask;

}

return x1; // Since p = 3, in binary form p = '011', then p1 = p2 = 1, so we can return either x1 or x2.

// If p = 4, in binary form p = '100', only p3 = 1, which implies we can only return x3.

// Or alternatively we can simply return (x1 | x2 | x3).

}

**Refer to**

<https://leetcode.wang/leetcode-137-Single-NumberII.html>

# **解法三 位操作**

136 题通过异或解决了问题，这道题明显不能用异或了，参考 [这里](https://leetcode.com/problems/single-number-ii/discuss/43297/Java-O(n)-easy-to-understand-solution-easily-extended-to-any-times-of-occurance) 的一个解法。

我们把数字放眼到二进制形式

假如例子是 1 2 6 1 1 2 2 3 3 3, 3 个 1, 3 个 2, 3 个 3,1 个 6

1 0 0 1

2 0 1 0

6 1 1 0

1 0 0 1

1 0 0 1

2 0 1 0

2 0 1 0

3 0 1 1

3 0 1 1

3 0 1 1

看最右边的一列 1001100111 有 6 个 1

再往前看一列 0110011111 有 7 个 1

再往前看一列 0010000 有 1 个 1

我们只需要把是 3 的倍数的对应列写 0，不是 3 的倍数的对应列写 1

也就是 1 1 0,也就是 6。

原因的话，其实很容易想明白。如果所有数字都出现了 3 次，那么每一列的 1 的个数就一定是 3 的倍数。之所以有的列不是 3 的倍数，就是因为只出现了 1 次的数贡献出了 1。所以所有不是 3 的倍数的列写 1，其他列写 0 ，就找到了这个出现 1 次的数。

public int singleNumber(int[] nums) {

int ans = 0;

//考虑每一位

for (int i = 0; i < 32; i++) {

int count = 0;

//考虑每一个数

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

//当前位是否是 1

if ((nums[j] >>> i & 1) == 1) {

count++;

}

}

//1 的个数是否是 3 的倍数

if (count % 3 != 0) {

ans = ans | 1 << i;

}

}

return ans;

}

时间复杂度：O（n）

空间复杂度：O（1）

# **解法四 通用方法**

参考 [这里](https://leetcode.com/problems/single-number-ii/discuss/43295/Detailed-explanation-and-generalization-of-the-bitwise-operation-method-for-single-numbers)。

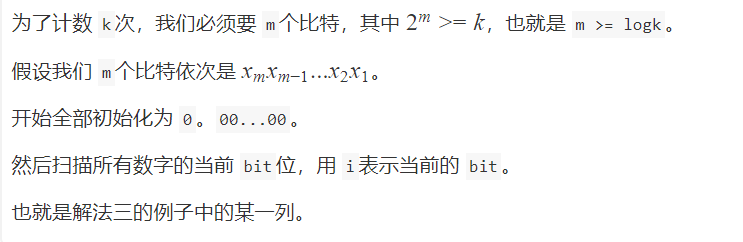
解法三中，我们将数字转为二进制，统计了每一位的 1 的个数。我们使用了一个 32位 的 int 来统计。事实上，我们只需要看它是不是 3 的倍数，所以我们只需要两个 bit 位就够了。初始化为 00，遇到第一个 1 变为 01，遇到第二个 1 变为 10，遇到第三个 1 变回 00 。接下来就需要考虑怎么做到。

本来想按自己理解的思路写一遍，但 [这里](https://leetcode.com/problems/single-number-ii/discuss/43295/Detailed-explanation-and-generalization-of-the-bitwise-operation-method-for-single-numbers) 写的很好了，主要还是翻译下吧。

## **1. 将问题一般化**

给一个数组，每个元素都出现 k ( k > 1) 次，除了一个数字只出现 p 次(p >= 1, p % k !=0)，找到出现 p 次的那个数。

## **2. 考虑其中的一个 bit**



假如例子是 1 2 6 1 1 2 2 3 3 3, 3 个 1, 3 个 2, 3 个 3,1 个 6

1 0 0 1

2 0 1 0

6 1 1 0

1 0 0 1

1 0 0 1

2 0 1 0

2 0 1 0

3 0 1 1

3 0 1 1

3 0 1 1

初始 状态 00...00。

第一次遇到 1 , m 个比特依次是 00...01。

第二次遇到 1 , m 个比特依次是 00...10。

第三次遇到 1 , m 个比特依次是 00...11。

第四次遇到 1 , m 个比特依次是 00..100。

x1 的变化规律就是遇到 1 变成 1 ，再遇到 1 变回 0。遇到 0 的话就不变。

所以 x1 = x1 ^ i，可以用异或来求出 x1 。

那么 x2...xm 怎么办呢？

x2 的话，当遇到 1 的时候，如果之前 x1 是 0，x2 就不变。如果之前 x1 是 1，对应于上边的第二次遇到 1 和第四次遇到 1。 x2 从 0 变成 1 和 从 1 变成 0。

所以 x2 的变化规律就是遇到 1 同时 x1 是 1 就变成 1，再遇到 1 同时 x1 是 1 就变回 0。遇到 0 的话就不变。和 x1 的变化规律很像，所以同样可以使用异或。

x2 = x2 ^ (i & x1)，多判断了 x1 是不是 1。

x3，x4 ... xm 就是同理了，xm = xm ^ (xm-1 & ... & x1 & i) 。

再说直接点，上边其实就是模拟了每次加 1 的时候，各个比特位的变化。所以高位 xm 只有当低位全部为 1 的时候才会得到进位 1 。

00 -> 01 -> 10 -> 11 -> 00

上边有个问题，假设我们的 k = 3，那么我们应该在 10 之后就变成 00，而不是到 11。

所以我们需要一个 mask ，当没有到达 k 的时候和 mask进行与操作是它本身，当到达 k 的时候和 mask 相与就回到 00...000。

根据上边的要求构造 mask，假设 k 写成二进制以后是 km...k2k1。

mask = ~(y1 & y2 & ... & ym),

如果kj = 1，那么yj = xj

如果 kj = 0，yj = ~xj 。

举两个例子。

k = 3: 写成二进制，k1 = 1, k2 = 1, mask = ~(x1 & x2);

k = 5: 写成二进制，k1 = 1, k2 = 0, k3 = 1, mask = ~(x1 & ~x2 & x3);

很容易想明白，当 x1x2...xm 达到 k1k2...km 的时候因为我们要把 x1x2...xm 归零。我们只需要用 0 和每一位进行与操作就回到了 0。

所以我们只需要把等于 0 的比特位取反，然后再和其他所有位相与就得到 1 ，然后再取反就是 0 了。

如果 x1x2...xm 没有达到 k1k2...km ，那么求出来的结果一定是 1，这样和原来的 bit 位进行与操作的话就保持了原来的数。

总之，最后我们的代码就是下边的框架。

for (int i : nums) {

xm ^= (xm-1 & ... & x1 & i);

xm-1 ^= (xm-2 & ... & x1 & i);

.....

x1 ^= i;

mask = ~(y1 & y2 & ... & ym) where yj = xj if kj = 1, and yj = ~xj if kj = 0 (j = 1 to m).

xm &= mask;

......

x1 &= mask;

}

# **3. 考虑全部 bit**

假如例子是 1 2 6 1 1 2 2 3 3 3, 3 个 1, 3 个 2, 3 个 3,1 个 6

1 0 0 1

2 0 1 0

6 1 1 0

1 0 0 1

1 0 0 1

2 0 1 0

2 0 1 0

3 0 1 1

3 0 1 1

3 0 1 1

之前是完成了一个 bit 位，也就是每一列的操作。因为我们给的数是 int 类型，所以有 32 位。所以我们需要对每一位都进行计数。有了上边的分析，我们不需要再向解法三那样依次考虑每一位，我们可以同时对 32 位进行计数。

对于 k 等于 3 ，也就是这道题。我们可以用两个 int，x1 和 x2。x1 表示对于 32 位每一位计数的低位，x2 表示对于 32 位每一位计数的高位。通过之前的公式，我们利用位操作就可以同时完成计数了。

int x1 = 0, x2 = 0, mask = 0;

for (int i : nums) {

x2 ^= x1 & i;

x1 ^= i;

mask = ~(x1 & x2);

x2 &= mask;

x1 &= mask;

}

## **4.返回什么**

最后一个问题，我们需要返回什么？

解法三中，我们看 1 出现的个数是不是 3 的倍数，不是 3 的倍数就将对应位置 1。

这里的话一样的道理，因为所有的数字都出现了 k 次，只有一个数字出现了 p 次。

因为 xm...x2x1 组合起来就是对于每一列 1 的计数。举个例子

假如例子是 1 2 6 1 1 2 2 3 3 3, 3 个 1, 3 个 2, 3 个 3,1 个 6

1 0 0 1

2 0 1 0

6 1 1 0

1 0 0 1

1 0 0 1

2 0 1 0

2 0 1 0

3 0 1 1

3 0 1 1

3 0 1 1

看最右边的一列 1001100111 有 6 个 1, 也就是 110

再往前看一列 0110011111 有 7 个 1, 也就是 111

再往前看一列 0010000 有 1 个 1, 也就是 001

再对应到 x1, x2, x3 就是

x1 1 1 0

x2 0 1 1

x3 0 1 1

如果 p = 1，那么如果出现一次的数字的某一位是 1 ，一定会使得 x1 ，也就是计数的最低位置的对应位为 1，所以我们把 x1 返回即可。对于上边的例子，就是 110 ，所以返回 6。

如果 p = 2，二进制就是 10，那么如果出现 2次的数字的某一位是 1 ，一定会使得 x2 的对应位变为 1，所以我们把 x2 返回即可。

如果 p = 3，二进制就是 11，那么如果出现 3次的数字的某一位是 1 ，一定会使得 x1 和x2的对应位都变为1，所以我们把 x1 或者 x2 返回即可。

所以这道题的代码就出来了

public int singleNumber(int[] nums) {

int x1 = 0, x2 = 0, mask = 0;

for (int i : nums) {

x2 ^= x1 & i;

x1 ^= i;

mask = ~(x1 & x2);

x2 &= mask;

x1 &= mask;

}

return x1;

}

至于为什么先对 x2 异或再对 x1 异或，就是因为 x2 的变化依赖于 x1 之前的状态。颠倒过来明显就不对了。

再扩展一下题目，对于 k = 5, p = 3 怎么做，也就是每个数字出现了5 次，只有一个数字出现了 3 次。

首先根据 k = 5，所以我们至少需要 3 个比特位。因为 2 个比特位最多计数四次。

然后根据 k 的二进制形式是 101，所以 mask = ~(x1 & ~x2 & x3)。

根据 p 的二进制是 011，所以我们最后可以把 x1 返回。

public int singleNumber(int[] nums) {

int x1 = 0, x2 = 0, x3 = 0, mask = 0;

for (int i : nums) {

x3 ^= x2 & x1 & i;

x2 ^= x1 & i;

x1 ^= i;

mask = ~(x1 & ~x2 & x3);

x3 &= mask;

x2 &= mask;

x1 &= mask;

}

return x1;

}

而 [136 题](https://leetcode.wang/leetcode-136-Single-Number.html) 中，k = 2, p = 1 ，其实也是这个类型。只不过因为 k = 2，而我们用一个比特位计数的时候，等于 2 的时候就自动归零了，所以不需要 mask，相对来说就更简单了。

public int singleNumber(int[] nums) {

int x1 = 0;

for (int i : nums) {

x1 ^= i;

}

return x1;

}

这个解法真是太强了，完全回到二进制的操作，五体投地了，推荐再看一下英文的 [原文](https://leetcode.com/problems/single-number-ii/discuss/43295/Detailed-explanation-and-generalization-of-the-bitwise-operation-method-for-single-numbers) 分析，太强了。

**Refer to**

<https://stackoverflow.com/questions/2811319/difference-between-and>

**>> is arithmetic shift right, >>> is logical shift right.**

In an arithmetic shift, the sign bit is extended to preserve the signedness of the number.

For example: -2 represented in 8 bits would be 11111110 (because the most significant bit has negative weight). Shifting it right one bit using arithmetic shift would give you 11111111, or -1. Logical right shift, however, does not care that the value could possibly represent a signed number; it simply moves everything to the right and fills in from the left with 0s. Shifting our -2 right one bit using logical shift would give 01111111.

**Refer to**

<https://www.educative.io/blog/bit-manipulation-in-java>

### NOT Operator

NOT (~), or sometimes called the bitwise complement operator, is a unary operation that takes a single input and **swaps each bit** in its binary representation to the opposite value.

All instances of 0 become 1, and all instances of 1 become 0. In other words, NOT inverts each input bit. This inverted sequence is called the **one’s complement** of a bit series.

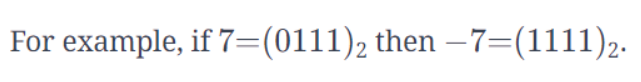
**Refer to**

<https://www.educative.io/answers/how-to-represent-negative-binary-numbers>

Since computers only understand binary numbers but need to perform operations involving negative numbers, there is a way to represent negative numbers (like positive numbers) in a binary format. A negative binary number can be made from its positive version in the following two ways:

## 1. Sign and magnitude

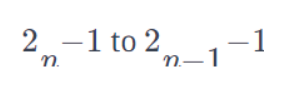
This is a simple approach that adds an extra bit (i.e., *sign-bit*) to detect the sign of a number. 1 indicates a *-ve* number, and 0 indicates a *+ve* number or vice versa (depending on the architecture of the computer).



A drawback is that the adders, in the underlying hardware of the computer, need to determine the *sign-bit* of an operation’s result. Due to this, and other disadvantages, the sign-bit representation of negative numbers is now obsolete.

## 2. Two’s complement

In this representation, the left-most bit is considered to be the *sign-bit*(without adding an extra bit), where 1 is a -ve and 0 is a +ve. This reduces the range of positive numbers that can be represented (using *n* bits) from



Despite this drawback, it is now the standard way of representing negative binary numbers.

To convert a positive number into a negative number, using the two’s complement representation, invert all of the bits of the number and add 11.

### Example

