

Single Number II

Given an array of integers, every element appears *three* times except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Solution 1

```
public int singleNumber(int[] A) {  
    int ones = 0, twos = 0;  
    for(int i = 0; i < A.length; i++){  
        ones = (ones ^ A[i]) & ~twos;  
        twos = (twos ^ A[i]) & ~ones;  
    }  
    return ones;  
}
```

written by [againest1](#) original link [here](#)

Solution 2

Statement of our problem: "Given an array of integers, every element appears k ($k > 1$) times except for one, which appears p times ($p \geq 1$, $p \% k \neq 0$). Find that single one."

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: x_m, \dots, x_1 (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 $2^m \geq k$, which implies $m \geq \log k$.

Here is the key part: how each bit in the counter (x_1 to x_m) 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: $x = x \mid 0$ and $x = x \wedge 0$.

Okay, we have an expression now: $x = x \mid i$ or $x = x \wedge i$, 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., $x_m = 0, \dots, x_1 = 0$. Since we are gonna choose bitwise operations that guarantees 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: $x_m = 0, \dots, x_2 = 0, x_1 = 1$. Let's continue until we hit the second '1', after which we have: $x_m = 0, \dots, x_2 = 1, x_1 = 0$. Note that x_1 changes from 1 to 0. For $x_1 = x_1 \mid i$, after the second count, x_1 will still be 1. So it's clear we should use $x_1 = x_1 \wedge i$. What about x_2, \dots, x_m ? The idea is to find the condition under which x_2, \dots, x_m will change their values. Take x_2 as an example. If we hit a '1', and we need to change the value of x_2 , what must the value of x_1 be before we do the change? The answer is: x_1 must be 1 otherwise we shouldn't change x_2 because changing x_1 from 0 to 1 will do the job. So x_2 will change only if x_1 and i are both 1, or mathematically, $x_2 = x_2 \wedge (x_1 \& i)$. Similarly x_m will change only when x_{m-1}, \dots, x_1 and i are all 1: $x_m = x_m \wedge (x_{m-1} \& \dots \& x_1 \& i)$; 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 x_m, \dots, x_1 with some variable called mask, i.e., $x_m = x_m \& \text{mask}$, ..., $x_1 = x_1 \& \text{mask}$. If we can make sure the 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: k_m, \dots, k_1 . we can construct the mask as follows:

$\text{mask} = \sim(x_1' \& x_2' \& \dots \& x_m')$, where $x_j' = x_j$ if $k_j = 1$ and $x_j' = \sim x_j$ if $k_j = 0$ ($j = 1$ to m).

Let's do some examples:

$k = 3$: $k_1 = 1, k_2 = 1, \text{mask} = \sim(x_1 \& x_2)$;

$k = 5$: $k_1 = 1, k_2 = 0, k_3 = 1, \text{mask} = \sim(x_1 \& \sim x_2 \& x_3)$;

In summary, our algorithm will go like this:

```
for (int i : array) {
     $x_m \wedge = (x_{m-1} \& \dots \& x_1 \& i)$ ;
     $x_{m-1} \wedge = (x_{m-2} \& \dots \& x_1 \& i)$ ;
    .....
     $x_1 \wedge = i$ ;
     $\text{mask} = \sim(x_1' \& x_2' \& \dots \& x_m')$  where  $x_j' = x_j$  if  $k_j = 1$  and  $x_j' = \sim x_j$  if  $k_j = 0$  ( $j = 1$  to  $m$ ).
     $x_m \&= \text{mask}$ ;
    .....
     $x_1 \&= \text{mask}$ ;
}
```

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 codes. But 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 $m \geq \log k$. 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. Since each counter has m bits, we end up with m 32-bit integers. Therefore, in the algorithm developed above, we just need to regard x_1 to x_m as 32-bit integers instead of 1-bit numbers and we are done. Easy, hum?

The last thing is what value we should return, or equivalently which one of x_1 to x_m will equal the single element. To get the correct answer, we need to understand what the m 32-bit integers x_1 to x_m represent. Take x_1 as an example. x_1 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 x_1 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, \dots, q'_1 , then by definition the r -th bit of x_1 will equal q'_1). Now you can ask yourself this question: what does it imply if the r -th bit of x_1 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 x_1 will be set 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 ($p \% k \neq 0$) times will contribute. If $p > k$, then the first $\lfloor p/k \rfloor$ (denotes the integer part of p/k) single elements won't contribute either. Then 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, \dots, p'_1 . (note that $p' < k$, so it will fit into m bits). Here I claim the condition for x_1 to equal the single element is $p'_1 = 1$. Quick proof: if the r -th bit of x_1 is '1', we can safely say the r -th bit of the single element is also '1'. We are left to prove that if the r -th bit of x_1 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. If we write p' in its binary form: p'_m, \dots, p'_1 , then by definition the r -th bit of x_1 will equal p'_1 , which is '1'. This contradicts with the presumption that the r -th bit of x_1 is '0'. Since this is true for all bits in x_1 , we can conclude x_1 will equal the single element if $p'_1 = 1$. Similarly we can show x_j will equal the single element if $p'_j = 1$ ($j = 1$ to m). Now it's clear what we should return. Just express $p' = p \% k$ in its binary form, and return any of the corresponding x_j as long as $p'_j = 1$.

In total, the algorithm will run in $O(n \cdot \log k)$ time and $O(\log k)$ space.

Hope this helps!

written by [fun4LeetCode](#) original link [here](#)

Solution 3

this kind of question the key idea is design a counter that record state. the problem can be every one occurs K times except one occurs M times. for this question, $K=3$, $M=1$ (or 2) . so to represent 3 state, we need two bit. let say it is a and b, and c is the incoming bit. then we can design a table to implement the state move.

current		incoming	next
a	b	c	a b
0	0	0	0 0
0	1	0	0 1
1	0	0	1 0
0	0	1	0 1
0	1	1	1 0
1	0	1	0 0

like circuit design, we can find out what the next state will be with the incoming bit.(we only need find the ones) then we have for a to be 1, we have

current		incoming	next
a	b	c	a b
1	0	0	1 0
0	1	1	1 0

and this is can be represented by

$$a = a \& \sim b \& \sim c + \sim a \& b \& c$$

and b can do the same we , and we find that

$$b = \sim a \& b \& \sim c + \sim a \& \sim b \& c$$

and this is the final formula of a and b and just one of the result set, because for different state move table definition, we can generate different formulas, and this one is may not the most optimised. as you may see other's answer that have a much simple formula, and that formula also corresponding to specific state move table. (if you like ,you can reverse their formula to a state move table, just using the same way but reversely)

for this questions we need to find the except one as the question don't say if the one appears one time or two time , so for ab both

```
01 10 => 1
00 => 0
```

we should return $a|b$; this is the key idea , we can design any based counter and find the occurs any times except one . here is my code. with comment.

```

public class Solution {

    public int singleNumber(int[] nums) {
        //we need to implement a tree-time counter(base 3) that if a bit appears
        three time ,it will be zero.
        //#curent income ouput
        //# ab      c/c      ab/ab
        //# 00      1/0      01/00
        //# 01      1/0      10/01
        //# 10      1/0      00/10
        // a=~abc+a~b~c;
        // b=~a~bc+~ab~c;
        int a=0;
        int b=0;
        for(int c:nums){
            int ta=(~a&b&c)|(a&~b&~c);
            b=(~a&~b&c)|(~a&b&~c);
            a=ta;
        }
        //we need find the number that is 01,10 => 1, 00 => 0.
        return a|b;
    }
}

```

this is a general solution . and it comes from the Circuit Design on course digital logic.

written by [ziyihao](#) original link [here](#)

From [LeetCoder](#).