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

https://discuss.leetcode.com/topic/11877/detailed-explanation-and-generalization-of-the-bitwise-operation-method-for-single-numbers

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."

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 changes 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 we need to change the value of x2, what must the value of x1 be 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 only if x1 and i are both 1, or mathematically, x2 = x2 ^ (x1 & i). Similarly xm will change 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 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: km,..., k1. we can construct the mask as follows:

mask = ~(x1' & x2' & ... xm'), where xj' = xj if kj = 1 and xj' = ~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:

```
for (int i : array) {
xm ^= (xm-1 & ... & x1 & i);
xm-1 ^= (xm-2 & ... & x1 & i);
.....
x1 ^= i;
mask = ~(x1' & x2' & ... xm') where xj' = xj if kj = 1 and xj' = ~xj if kj = 0 (j = 1 to m).
xm &= mask;
......
x1 &= 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 >= 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. 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 x1 to xm 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 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 equal 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 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 != 0) times will contribute. If p > k, then the first k*[p/k] (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, ..., p'1. (note that p' < k, so it will fit into m bits). Here I claim the condition for x1 to equal the single element is p'1 = 1. Quick proof: if the r-th bit of x1 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 x1 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, ..., p'1, then by definition the r-th bit of x1 will equal p'1, which is '1'. This contradicts with the presumption that the r-th bit of x1 is '0'. Since this is true for all bits in x1, we can conclude x1 will equal the single element if p'1 = 1. Similarly we can show xj 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 xj as long as p'j = 1.

In total, the algorithm will run in O(n*logk) time and O(logk) space.

Hope this helps!

Here is a list of few quick examples to show how the algorithm works:

1. 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:

```java
public int singleNumber(int[] A) {
     int x1 = 0;
     for (int i : A) {
        x1 ^= i;
     }
     return x1;
}
```

2. 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:

```java
public int singleNumber(int[] A) {
     int x1 = 0;
     int x2 = 0;
     int mask = 0;

     for (int i : A) {
        x2 ^= x1 & i;
        x1 ^= i;
        mask = ~(x1 & x2);
        x2 &= mask;
        x1 &= mask;
     }

     return x1;  // 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, so we should return x2.
}
```

3. 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:

```java
public int singleNumber(int[] A) {
     int x1 = 0;
     int x2 = 0;
     int x3  = 0;
     int mask = 0;

     for (int i : A) {
        x3 ^= x2 & x1 & i;
        x2 ^= x1 & i;
        x1 ^= i;
        mask = ~(x1 & ~x2 & x3);
        x3 &= mask;
        x2 &= mask;
        x1 &= mask;
     }

     return x1;  // p = 3, in binary form p = '011', then p1 = p
2 = 1,
                  // so we can return either x1 or x2;
                  // But if p = 4, in binary form p = '100', then
only p3 = 1,
                  // which implies we can only return x3.
}
```

You can easily come up with other examples. If you have any questions about the explanation, please let me know. I would appreciate your feedback. Thanks!