137. Single Number II

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)

`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;
}
```

Notes

`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.

0_1510941016426_137. Single Number II .png

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 `0` s, 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):

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:

```
public int singleNumber(int[] nums) {
    int x1 = 0;

    for (int i : nums) {
        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:

```
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).
}
```

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:

```
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).
}
```

Lastly I would like to thank those for providing feedbacks to make this post better. Hope it helps and happy coding!

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

Show All (https://discuss.leetcode.com/category/145)

Java

```
class Solution {
    public int singleNumber(int[] nums) {

    }
}
```

Notes

Custom Testcase (Contribute)

Run

Submit