Single Number III

Given an array of numbers `nums`, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given `nums = [1, 2, 1, 3, 2, 5]`, return `[3, 5]`.

Note:

1. The order of the result is not important. So in the above example, `[5, 3]` is also correct.
2. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 1

Once again, we need to use XOR to solve this problem. But this time, we need to do it in two passes:

- In the first pass, we XOR all elements in the array, and get the XOR of the two numbers we need to find. Note that since the two numbers are distinct, so there must be a set bit (that is, the bit with value '1') in the XOR result. Find out an arbitrary set bit (for example, the rightmost set bit).
- In the second pass, we divide all numbers into two groups, one with the aforementioned bit set, another with the aforementinoed bit unset. Two different numbers we need to find must fall into thte two distrinct groups. XOR numbers in each group, we can find a number in either group.

Complexity:

- Time: $O(n)$
- Space: $O(1)$

A Corner Case:

- When `diff == numeric_limits<int>::min()`, `-diff` is also `numeric_limits<int>::min()`. Therefore, the value of `diff` after executing `diff &= -diff` is still `numeric_limits<int>::min()`. The answer is still correct.

C++:

```cpp
class Solution
{
public:
    vector<int> singleNumber(vector<int>& nums)
    {
        // Pass 1 :
        // Get the XOR of the two numbers we need to find
        int diff = accumulate(nums.begin(), nums.end(), 0, bit_xor<int>());
        // Get its last set bit
        diff &= -diff;

        // Pass 2 :
        vector<int> rets = {0, 0}; // this vector stores the two numbers we will
return
        for (int num : nums)
        {
            if ((num & diff) == 0) // the bit is not set
            {
                rets[0] ^= num;
            }
            else // the bit is set
            {
                rets[1] ^= num;
            }
        }
        return rets;
    }
};
```

Java:

```java
public class Solution {
    public int[] singleNumber(int[] nums) {
        // Pass 1 :
        // Get the XOR of the two numbers we need to find
        int diff = 0;
        for (int num : nums) {
            diff ^= num;
        }
        // Get its last set bit
        diff &= -diff;

        // Pass 2 :
        int[] rets = {0, 0}; // this array stores the two numbers we will return
        for (int num : nums)
        {
            if ((num & diff) == 0) // the bit is not set
            {
                rets[0] ^= num;
            }
            else // the bit is set
            {
                rets[1] ^= num;
            }
        }
        return rets;
    }
}
```

Thanks for reading :)

Acknowledgements:

- Thank **@jianchao.li.fighter** for introducing this problem and for your encouragement.
- Thank **@StefanPochmann** for your valuable suggestions and comments. Your idea of `diff &= -diff` is very elegent! And yes, it does not need to XOR for both group in the second pass. XOR for one group suffices. I revise my code accordingly.
- Thank **@Nakagawa_Kanon** for posting this question and presenting the same idea in a previous thread (prior to this thread).
- Thank **@caijun** for providing an interesting test case.

written by zhiqing_xiao original link here

Solution 2

If you were stuck by this problem, it's easy to find a solution in the discussion. However, usually, the solution lacks some explanations.

I'm sharing my understanding here:

The two numbers that appear only once must differ at some bit, this is how we can distinguish between them. Otherwise, they will be one of the duplicate numbers.

Let’s say the at the ith bit, the two desired numbers differ from each other. which means one number has bit i equaling: 0, the other number has bit i equaling 1.

Thus, all the numbers can be partitioned into two groups according to their bits at location i. the first group consists of all numbers whose bits at i is 0. the second group consists of all numbers whose bits at i is 1.

Notice that, if a duplicate number has bit i as 0, then, two copies of it will belong to the first group. Similarly, if a duplicate number has bit i as 1, then, two copies of it will belong to the second group.

by XoRing all numbers in the first group, we can get the first number. by XoRing all numbers in the second group, we can get the second number.

written by douglasleer original link here

Solution 3

```
vector<int> singleNumber(vector<int>& nums) {
    int aXorb = 0;  // the result of a xor b;
    for (auto item : nums) aXorb ^= item;
    int lastBit = (aXorb & (aXorb - 1)) ^ aXorb;  // the last bit that a diffs b
    int intA = 0, intB = 0;
    for (auto item : nums) {
        // based on the last bit, group the items into groupA(include a) and grou
pB
        if (item & lastBit) intA = intA ^ item;
        else intB = intB ^ item;
    }
    return vector<int>{intA, intB};
}
```

written by lchen77 original link here