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

Given an integer array 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. You can return the answer in **any order**.

You must write an algorithm that runs in linear runtime complexity and uses only constant extra space.

**Example 1:**

**Input:** nums = [1,2,1,3,2,5]

**Output:** [3,5]

**Explanation:**  [5, 3] is also a valid answer.

**Example 2:**

**Input:** nums = [-1,0]

**Output:** [-1,0]

**Example 3:**

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

**Output:** [1,0]

**Constraints:**

2 <= nums.length <= 3 \* 10^4

-2^31 <= nums[i] <= 2^31 - 1

Each integer in nums will appear twice, only two integers will appear once.

**Attempt 1: 2023-03-17**

**Solution 1: Brute Force (30 min)**

**Wrong Solution**

**Error out when set 'j = i + 1', test out by nums = [1,2,1,3,2,5], if only allow j > i, then when check the 2nd 1, it will not able to trace back to compare with the 1st 1, then 2nd 1 will be wrongly set**

class Solution {

public int[] singleNumber(int[] nums) {

List<Integer> tmp = new ArrayList<>();

int n = nums.length;

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

for(int j = i + 1; j < n; j++) {

if(nums[i] == nums[j]) {

break;

} else if(j == n - 1) {

tmp.add(nums[i]);

}

}

}

return tmp.stream().mapToInt(i -> i).toArray();

}

}

**Correct Solution**

class Solution {

    public int[] singleNumber(int[] nums) {

        List<Integer> tmp = new ArrayList<>();

        int n = nums.length;

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < n; j++) {

                if(i != j && nums[i] == nums[j]) {

                    break;

                } else if(j == n - 1) {

                    tmp.add(nums[i]);

                }

            }

        }

        return tmp.stream().mapToInt(i -> i).toArray();

    }

}

Time Complexity: O(N^2), where N is the number of elements in the given array.

We iterate over the entire array for each element until we find two elements

that occur only once. In the worst case, we iterate over entire array for N times

for N elements giving the overall time complexity of O(N\*N) = O(N^2)

Space Complexity: O(1), only constant extra space is used

**Solution 2: Sorting (30 min)**

**Wrong Solution 1**

**Error out when test by nums = [0,1,1,2], output = [0,1,1,2], expect = [2,0], the reason is assuming only pattern as nums[i] = nums[i + 1] for all elements appear twice not intercept by or divide two single element, like [1,1,2,2,3,5] pattern, but [0,1,1,2] is the case which [1,1] divide [0] and [2].**

public int[] singleNumber(int[] nums) {

Arrays.sort(nums);

List<Integer> tmp = new ArrayList<>();

int n = nums.length;

for(int i = 0; i < n - 1; i += 2) {

if(nums[i] != nums[i + 1]) {

tmp.add(nums[i]);

tmp.add(nums[i + 1]);

}

}

return tmp.stream().mapToInt(i -> i).toArray();

}

**Wrong Solution 2**

**Error out when test by nums = [1,2,1,3,2,5], output = [1,2,3,5], expect = [3,5], the reason is we only check ith element with its left side adjacent number as (i - 1)th element, but after sorting, the nums = [1,1,2,2,3,5], if the checking condition is 'i > 0 && nums[i] == nums[i - 1]', for the first element as 0 will be a problem, it has duplicates but not detect by if condition, it will directly go into 'else' branch as 'tmp.add(nums[i])', which is wrong, we have to also check ith element with its right side adjacent number as (i + 1)th element to avoid only one side check and corner case like first / last element on nums.**

class Solution {

    public int[] singleNumber(int[] nums) {

        Arrays.sort(nums);

        List<Integer> tmp = new ArrayList<>();

        int n = nums.length;

        for(int i = 0; i < n; i++) {

            if(i > 0 && nums[i - 1] == nums[i]) {

                continue;

            } else {

                tmp.add(nums[i]);

            }

        }

        return tmp.stream().mapToInt(i -> i).toArray();

    }

}

**Correct Solution**

class Solution {

    public int[] singleNumber(int[] nums) {

        Arrays.sort(nums);

        List<Integer> tmp = new ArrayList<>();

        int n = nums.length;

        for(int i = 0; i < n; i++) {

            if((i > 0 && nums[i - 1] == nums[i]) || (i < n - 1 && nums[i] == nums[i + 1])) {

                continue;

            } else {

                tmp.add(nums[i]);

            }

        }

        return tmp.stream().mapToInt(i -> i).toArray();

    }

}

Time Complexity: O(NlogN), required for sorting

Space Complexity: O(sort), the space required by internal sorting algorithm used.

**Solution 3: Hash Table (10 min)**

**Style 1: Traditional frequency table way**

class Solution {

    public int[] singleNumber(int[] nums) {

        Map<Integer, Integer> map = new HashMap<>();

        List<Integer> tmp = new ArrayList<>();

        int n = nums.length;

        for(int i = 0; i < n; i++) {

            map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

        }

        for(int num : map.keySet()) {

            if(map.get(num) == 1) {

                tmp.add(num);

            }

        }

        return tmp.stream().mapToInt(i -> i).toArray();

    }

}

Time Complexity: O(N), we iterate over the array once and the hashmap once which contains N-2 elements. Thus the overall time complexity comes out to

O(N + N - 2) ≈ O(2N-2) ≈ O(N)

Space Complexity: O(N), required for maintaining the hashmap.

**Style 2: Traditional frequency table way**

class Solution {

    public int[] singleNumber(int[] nums) {

        Set<Integer> set =  new HashSet<>();

        int n = nums.length;

        for(int i = 0; i < n; i++) {

            // set.contains return true if this set contains the specified element

            if(set.contains(nums[i])) {

                set.remove(nums[i]);

            } else {

                set.add(nums[i]);

            }

        }

        return set.stream().mapToInt(i -> i).toArray();

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Solution 4: Bit Manipulation (60 min)**

class Solution {

    public int[] singleNumber(int[] nums) {

        int xor1 = 0;

        int xor2 = 0;

        int i = 0;

        // XOR all elements in array

        for(int num : nums) {

            xor1 ^= num;

        }

        // Find the lowest set bit in xor1

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

            if((xor1 & 1 << bit) != 0) {

                i = bit;

                break;

            }

        }

        // XOR all elements having ith bit set

        for(int num : nums) {

            if((num & 1 << i) != 0) {

                xor2 ^= num;

            }

        }

        // Return final result

        return new int[]{xor1 ^ xor2, xor2};

    }

}

Time Complexity: O(N). We iterate over the array exactly twice.

Space Complexity: O(1), only constant extra space is used.

**Refer to**

**5 Simple Solutions w/ Explanation | Brute-Force + Sort + Hashmap + Hashset + Xor O(1)**

<https://leetcode.com/problems/single-number-iii/solutions/1561827/c-python-5-simple-solutions-w-explanation-brute-force-sort-hashmap-hashset-xor-o-1/>

✔️ **Solution - I (Brute-Force)**

Let's start with the most obvious brute-force approach. We can select each number in the given array and re-iterate whole array to see if there's a duplicate. If we don't find a duplicate, then we will push that element into our resultant array and return at the end.

class Solution {

public:

vector<int> singleNumber(vector<int>& nums) {

vector<int> ans;

for(int i = 0; i < size(nums) && size(ans) < 2; i++)

for(int j = 0; j < size(nums); j++)

if(i != j and nums[i] == nums[j]) break;

else if(j == size(nums)-1) ans.push\_back(nums[i]);

return ans;

}

};

Time Complexity : O(N^2), where N is the number of elements in the given array.

We iterate over the entire array for each element until we find two elements that

occur only once. In the worst case, we iterate over entire array for N times for

N elements giving the overall time complexity of O(N\*N) = O(N^2)

Space Complexity : O(1), only constant extra space is used

✔️ **Solution - II (Sort)**

We can sort and then each duplicate element will be adjacent. We can iterate over array and check if an element occurs only once by comparing it with its adjacent elements

class Solution {

public:

vector<int> singleNumber(vector<int>& nums) {

sort(begin(nums), end(nums));

vector<int> ans;

for(int i = 0; i < size(nums); i++)

if((i && nums[i] == nums[i-1]) || (i+1 < size(nums) && nums[i] == nums[i+1])) continue;

else ans.push\_back(nums[i]);

return ans;

}

};

Time Complexity: O(NlogN), required for sorting

Space Complexity: O(sort), the space required by internal sorting algorithm used.

✔️ **Solution - III (HashMap)**

A more efficient way to solve this problem is to maintain a hashmap.

We iterate over the array and store the number of times an element occurs in the array using the hashmap.

Then, we iterate over the hashmap and push the elements into the resultant array that occurs only once.

class Solution {

public:

vector<int> singleNumber(vector<int>& nums) {

vector<int> ans;

unordered\_map<int, int> mp;

for(const int n : nums) mp[n]++;

for(auto [n, freq] : mp)

if(freq == 1)

ans.push\_back(n);

return ans;

}

};

Time Complexity: O(N), we iterate over the array once and the hashmap once which contains

N-2 elements. Thus the overall time complexity comes out to O(N + N - 2) ≈ O(2N-2) ≈ O(N)

Space Complexity: O(N), required for maintaining the hashmap.

✔️ **Solution - IV (HashSet)**

This approach is very similar to previous one. But since the elements occur only once or twice, we can optimize the previous approach slightly by using hashset instead.

The first time we encounter an element, we add it into hashset

If we find an element that is already present in hashset, we remove it.

Finally, our hashset will only contain elements occuring once.

class Solution {

public:

vector<int> singleNumber(vector<int>& nums) {

unordered\_set<int> s;

for(const int n : nums)

if(s.count(n)) s.erase(n);

else s.insert(n);

return vector<int>(begin(s), end(s));

}

};

Time Complexity: O(N)

Space Complexity: O(N)

✔️ **Solution - V (Xor)**

Both the above solutions solve the question but neither do it both in optimal time and space complexity. I would suggest to first solve **136. Single Number** if you haven't already, before moving on to the solution.

We use the Xor function to solve this problem in linear time and constant space complexity. First, let's note the properties of Xor -

A ⊕ B = B ⊕ A

A ⊕ ( B ⊕ C ) = ( A ⊕ B ) ⊕ C

A ⊕ 0 = A

A ⊕ A = 0

The last property is the main one that will be helpful to solve this question. If we Xor all the elements of given array, all the **duplicate elements will cancel out each other** out to 0 because they occur exactly twice and the **result - Xor1 will be Xor of the elements that occur only once.**

**But how do we differentiate and find the two elements from Xor1?**

For this, we need to dig deeper into how Xor works on bit-level. The **Xor property-4** basically tells us that two same bits will cancel each other out to 0 (**1 ⊕ 1 = 0** and **0 ⊕ 0 = 0**). So, all the set bits in the Xor always occur odd number of times in the array.

Thus, we select any arbitrary set-bit of the Xor obtained in above iteration - let's choose the lowest set-bit - i. Then we can re-iterate over the array and Xor all elements having ith bit set. All the duplicates will again cancel out each other and the resultant Xor - **Xor2** will be one of the elements that occur only once.

We can then perform **Xor1 ⊕ Xor2**, which will cancel out the first element occuring once and we get the 2nd element as well :)

**Dry Run**

nums = [1,2,1,3,2,5]

We need to find a and b such that both of them occur once in the array.

1. Xor1 = 1 ⊕ 2 ⊕ 1 ⊕ 3 ⊕ 2 ⊕ 5 = 6

To verify, calculate a ⊕ b = 3 ⊕ 5 = 6 which is equal to Xor1

2. The lowest set-bit in Xor1 (110) is 2nd bit.

3. We Xor all elements having 2nd bit set.

So, Xor2 = 2 ⊕ 3 ⊕ 2 = 3

So, we get one of the number occuring exactly once - 3

4. We get the 2nd number by doing Xor1 ⊕ Xor2 = 6 ⊕ 3 = 5

Thus, we can see that this method correctly finds the two elements occuring once.

**Code**

class Solution {

public:

vector<int> singleNumber(vector<int>& nums) {

int Xor1 = 0, Xor2 = 0, i;

for(const int n : nums) Xor1 ^= n; // Xor all elements of array

for(int bit = 0; bit < 32; bit++) // finding the lowest set bit in Xor1

if(Xor1 & 1 << bit) {

i = bit;

break;

}

for(const int n : nums) // Xor-ing all elements having ith bit set

if(n & 1 << i)

Xor2 ^= n;

return {Xor1 ^ Xor2, Xor2}; // final result

}

};

Time Complexity: O(N). We iterate over the array exactly twice.

Space Complexity: O(1), only constant extra space is used.

**Refer to**

[L136.Single Number](note://WEBa55331b3e24a06aea335247aaa722bbf)