<https://leetcode.com/problems/find-all-lonely-numbers-in-the-array/description/>

You are given an integer array nums. A number x is lonely when it appears only once, and no adjacent numbers (i.e. x + 1 and x - 1) appear in the array.

Return all lonely numbers in nums. You may return the answer in any order.

**Example 1:**

**Input:** nums = [10,6,5,8]

**Output:** [10,8]

**Explanation:**

- 10 is a lonely number since it appears exactly once and 9 and 11 does not appear in nums.

- 8 is a lonely number since it appears exactly once and 7 and 9 does not appear in nums.

- 5 is not a lonely number since 6 appears in nums and vice versa.

Hence, the lonely numbers in nums are [10, 8].Note that [8, 10] may also be returned.

**Example 2:**

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

**Output:** [1,5]

**Explanation:**

- 1 is a lonely number since it appears exactly once and 0 and 2 does not appear in nums.

- 5 is a lonely number since it appears exactly once and 4 and 6 does not appear in nums.

- 3 is not a lonely number since it appears twice.Hence, the lonely numbers in nums are [1, 5].

Note that [5, 1] may also be returned.

**Constraints:**

1 <= nums.length <= 10^5

0 <= nums[i] <= 10^6

**Attempt 1: 2024-12-29**

**Solution 1: Hash Table (10 min)**

class Solution {

public List<Integer> findLonely(int[] nums) {

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

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

for(int num : nums) {

freq.put(num, freq.getOrDefault(num, 0) + 1);

}

for(int num : nums) {

if(freq.get(num) == 1 && !freq.containsKey(num - 1) && !freq.containsKey(num + 1)) {

result.add(num);

}

}

return result;

}

}

Time Complexity: O(n)

Space Complexity: O(1)

**Solution 2: Sorting (10 min)**

class Solution {

    public List<Integer> findLonely(int[] nums) {

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

        Arrays.sort(nums);

        for(int i = 0; i < nums.length; i++) {

            boolean lonely = true;

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

                lonely = false;

            }

            if(i < nums.length - 1 && (nums[i] == nums[i + 1] || nums[i] + 1 == nums[i + 1])) {

                lonely = false;

            }

            if(lonely) {

                result.add(nums[i]);

            }

        }

        return result;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Why HashMap as O(n) but not slower than sorting way as O(nlogn) ?**

**Refer to**

<https://leetcode.com/problems/find-all-lonely-numbers-in-the-array/solutions/1711759/java-100-tc-100-sc-sorted-array/>

class Solution {

public List<Integer> findLonely(int[] nums) {

Arrays.sort(nums);

ArrayList<Integer> list = new ArrayList<>();

for (int i = 1; i < nums.length - 1; i++) {

if (nums[i - 1] + 1 < nums[i] && nums[i] + 1 < nums[i + 1]) {

list.add(nums[i]);

}

}

if (nums.length == 1) {

list.add(nums[0]);

}

if (nums.length > 1) {

if (nums[0] + 1 < nums[1]) {

list.add(nums[0]);

}

if (nums[nums.length - 2] + 1 < nums[nums.length - 1]) {

list.add(nums[nums.length - 1]);

}

}

return list;

}

}

**Refer to**

<https://leetcode.com/problems/find-all-lonely-numbers-in-the-array/solutions/1711759/java-100-tc-100-sc-sorted-array/comments/1235257>

**Its highly likely that's the case. The HashMap is resized multiple times (worst case 13 times), and each of the n lookups and n put operations we perform on the map has an amortized (not actual) cost of O(1).**

**Looks like you're right on. Playing around with modifying the initial size and load factor of the map led to vastly different runtimes.**

**Refer to**

[L1838.Frequency of the Most Frequent Element (Ref.L2968)](note://WEBe75023aadd133f6c8d91ff09f6ec8794)