2150. Find All Lonely Numbers in the Array

Medium **Array** **Hash Table** **Counting**

Problem Description

In this problem, you are given an integer array called `nums`. A number `x` is considered `lonely` if it meets two conditions:

1. It appears only once in the array (`nums`).
2. Neither `x + 1` (the next consecutive number) nor `x - 1` (the previous consecutive number) appear in the array.

Your objective is to find all the lonely numbers within the `nums` array and return them. The order in which you return these lonely numbers does not matter, meaning they can be in any sequence.

Intuition

The intuition behind the solution to this problem is to determine the occurrence of each number as well as the occurrence of its immediate neighbors (i.e., `num - 1` and `num + 1`). To efficiently manage this, a `Counter` from Python's collections module is useful because it enables us to count the occurrences of each number in `nums` with ease.

Once we have the counts of each number, we can iterate through the items in the counter, which consist of the number and its count. During the iteration, we apply the conditions that define loneliness:

- The count of the number `num` should be exactly 1 (it appears only once).
- The count of `num - 1` should be 0 (no adjacent smaller number).
- The count of `num + 1` should be 0 (no adjacent larger number).

Solution Approach

The solution uses a hash table, specifically the `Counter` class from Python's collections module, to keep track of the frequency with which each number appears in the `nums` array. This is a common pattern when dealing with problems that require you to track the occurrences of elements because `Counter` provides an efficient way to store and query frequencies.

Here's a step-by-step explanation of the implementation:

1. Initialize the `Counter` with the `nums` array. The `Counter` will create a hash table where the keys are the numbers from the array and the values are the counts of each number.

```
counter = Counter(nums)
```

2. Create an empty list `ans` that will eventually contain all the lonely numbers.

```
ans = []
```

3. Iterate over the items in the `counter`. For each iteration, you have a `num` which is the number from the array and `cnt` which is the count of how many times that number appears.

```
for num, cnt in counter.items():
```

4. Inside the loop, you apply three checks for each number based on the definition of a `lonely` number:
   - Check if the current number `num` appears only once (`cnt == 1`).
   - Check if the number immediately smaller than the current number (`num - 1`) does not appear in the array (`counter[num - 1] == 0`).
   - Check if the number immediately larger than the current number (`num + 1`) does not appear in the array (`counter[num + 1] == 0`).
5. If all three checks pass, it means that `num` is a lonely number, and you append it to the `ans` list.

```
if cnt == 1 and counter[num - 1] == 0 and counter[num + 1] == 0:
    ans.append(num)
```

6. After the loop finishes, the `ans` list contains all the lonely numbers and the function returns this list.

```
return ans
```

By using `Counter`, we achieve a solution that is simple and efficient with respect to time complexity since we look up the counts in constant time O(1) and we only iterate through the elements of the array once, making the overall time complexity O(N), where N is the number of elements in `nums`.

Example Walkthrough

Let's consider an example to illustrate the solution approach. Suppose we have the following `nums` array:

```
nums = [4, 10, 5, 8, 20, 15, 11, 10]
```

Follow these steps to find all the lonely numbers:

1. Initialize the `Counter` with the `nums` array:

```
counter = Counter([4, 10, 5, 8, 20, 15, 11, 10])
# counter now looks like: {4:1, 10:2, 5:1, 8:1, 20:1, 15:1, 11:1}
```

2. Create an empty list `ans` to hold the lonely numbers:

```
ans = []
```

3. Iterate over the items in the `counter`:

```
for num, cnt in counter.items():
    # First iteration: num = 4, cnt = 1
    # Second iteration: num = 10, cnt = 2
    # and so on...
```

4. Apply the checks for loneliness:
   - In the first iteration, `num` is 4 and `cnt` is 1.
     - Check if `4` appears only once: Yes (`cnt == 1`).
     - Check if `3` appears in the array: No (`counter[3] == 0`).
     - Check if `5` appears in the array: Yes (`counter[5] != 0`), so `4` is not lonely since its immediate larger neighbor exists.
   - Skipping to the next number meeting first condition which is `5`.
     - Check if `5` appears only once: Yes (`cnt == 1`).
     - Check if `4` appears in the array: Yes (`counter[4] != 0`), so `5` is not lonely since its immediate smaller neighbor exists.
   - Skipping ahead to numbers that meet the first condition (`20` and `15`).
     - For `num = 20`, it appears only once, and neither `19` nor `21` appear in the array. So, `20` is lonely.
     - For `num = 15`, it appears only once, and neither `14` nor `16` appear in the array. So, `15` is lonely.
   - Continue this process for the rest of the numbers.
5. After applying the loneliness checks and iterating through all the counter's items, find that `20` and `15` meet the criteria for being lonely numbers:

```
ans.append(20)
ans.append(15)
```

6. Finally, return the `ans` list, which contains all the lonely numbers found in `nums`:

```
return ans
# The final returned ans list is: [20, 15]
```

The final output for our example is `[20, 15]`, which means `20` and `15` are the lonely numbers in the given `nums` array. They appear only once, and neither their immediate smaller nor larger neighbors appear in the array.

Solution Implementation

Python

```
from collections import Counter
from typing import List

class Solution:
    def findLonely(self, nums: List[int]) -> List[int]:
        # Create a counter for all numbers in the list
        num_counter = Counter(nums)
        # Initialize an empty list to store lonely numbers
        lonely_numbers = []

        # Iterate through the items in the counter
        for num, count in num_counter.items():
            # Check if the number appears only once and neither (num-1) nor (num+1) appear
            if count == 1 and num_counter[num - 1] == 0 and num_counter[num + 1] == 0:
                # If conditions are met, append the number to the lonely_numbers list
                lonely_numbers.append(num)

        # Return the list of lonely numbers
        return lonely_numbers
```

Java

```
import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;

class Solution {
    // This method finds all elements in the array that stand alone without
    // any adjacent or duplicate elements
    public List<Integer> findLonely(int[] nums) {
        // Create a HashMap to store the frequency of each number in the array
        Map<Integer, Integer> frequencyMap = new HashMap<>();

        // Iterate through the array and populate the frequency map
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Initialize a list to store the lonely numbers
        List<Integer> lonelyNumbers = new ArrayList<>();

        // Iterate through the map to find lonely numbers
        frequencyMap.forEach((number, count) -> {
            // A number is lonely if it appears exactly once and neither of its
            // adjacent numbers (number - 1, number + 1) are present in the array.
            if (count == 1 && !frequencyMap.containsKey(number - 1)
                && !frequencyMap.containsKey(number + 1)) {
                lonelyNumbers.add(number);
            }
        });

        // Return the list of lonely numbers
        return lonelyNumbers;
    }
}
```

C++

```
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Function to find 'lonely' integers in the array.
    // An integer is 'lonely' if its count is one and no adjacent integers (num - 1, num + 1) are present.
    vector<int> findLonely(vector<int>& nums) {
        // Create a hash map to keep track of the frequency of each number in the array.
        unordered_map<int, int> frequencyMap;

        // Iterate over the array and increment the count for each number in the frequencyMap.
        for (int num : nums) {
            ++frequencyMap[num];
        }

        // Prepare the answer vector to store the 'lonely' numbers.
        vector<int> lonelyNumbers;

        // Iterate through the elements of frequencyMap
        for (const auto& element : frequencyMap) {
            int number = element.first;        // The number itself
            int count = element.second;        // Frequency of the number

            // Check if the count is 1 (unique number) and both adjacent numbers do not exist.
            if (count == 1 && !frequencyMap.count(number - 1) && !frequencyMap.count(number + 1)) {
                // If conditions are satisfied, add number to the 'lonely' numbers list.
                lonelyNumbers.push_back(number);
            }
        }

        // Return the list of 'lonely' numbers.
        return lonelyNumbers;
    }
};
```

TypeScript

```
function findLonely(nums: number[]): number[] {
    // Initialize a hashMap to store the frequency of each number in the array
    let frequencyMap: Map<number, number> = new Map();

    // Populate the frequency map with counts of each number
    for (let num of nums) {
        frequencyMap.set(num, (frequencyMap.get(num) || 0) + 1);
    }

    // Initialize an array to store lonely numbers
    let lonelyNumbers: Array<number> = [];

    // Iterate through the entries of the frequency map
    for (let [num, count] of frequencyMap.entries()) {
        // Check if the current number is lonely, i.e., count is 1 and neither num-1 nor num+1 exist in the map
        // If conditions are met, add the number to the lonelyNumbers array
        if (count === 1 && !frequencyMap.has(num - 1) && !frequencyMap.has(num + 1)) {
            lonelyNumbers.push(num);
        }
    }

    // Return the array containing all lonely numbers
    return lonelyNumbers;
}
```

```
from collections import Counter
from typing import List

class Solution:
    def findLonely(self, nums: List[int]) -> List[int]:
        # Create a counter for all numbers in the list
        num_counter = Counter(nums)
        # Initialize an empty list to store lonely numbers
        lonely_numbers = []

        # Iterate through the items in the counter
        for num, count in num_counter.items():
            # Check if the number appears only once and neither (num-1) nor (num+1) appear
            if count == 1 and num_counter[num - 1] == 0 and num_counter[num + 1] == 0:
                # If conditions are met, append the number to the lonely_numbers list
                lonely_numbers.append(num)

        # Return the list of lonely numbers
        return lonely_numbers
```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by three factors: the creation of the `counter` which counts the occurrences of each element, the iteration through the `counter` dictionary, and the checks performed for each number.

1. Creating the `counter` from the `nums` list takes `O(n)` time where `n` is the number of elements in the `nums` list because it requires one pass through all elements.
2. Iterating through the `counter` dictionary's items also takes `O(n)` in the worst case, which is when all elements in `nums` are unique. We need to clarify this is "worst case" because the counter may have fewer than `n` keys if there are duplicate values in `nums`.
3. For each element in the `counter`, we check if the `cnt` is 1 and if the adjacent numbers `num - 1` and `num + 1` are not present. These checks are constant-time operations, `O(1)`, because dictionary lookup is an `O(1)` operation on average due to hash table implementation.

Combining these factors, the overall time complexity is `O(n) + O(n) * O(1)`, which simplifies to `O(n)`.

Space Complexity

The space complexity involves the space taken up by the `counter` dictionary and the `ans` list.

1. The `counter` dictionary will hold at most `n` key-value pairs if all elements in `nums` are unique, resulting in a space complexity of `O(n)`.
2. The `ans` list can also hold at most `n` elements in the worst case, where every element is lonely. Therefore, the space complexity for the `ans` list is also `O(n)`.

Therefore, the combined space complexity of the algorithm is also `O(n)`, where `n` is the number of elements in the `nums` list.