1133. Largest Unique Number

Easy Array Hash Table Sorting

Leetcode Link

Problem Description

The problem presents us with a simple task: Given an array of integers called `nums`, we are to find the highest value integer that appears exactly once in the array. If all integers appear more than once or if the array is empty, the function should return `-1`.

For example, if the input array is `[5, 7, 3, 9, 4, 9, 8, 3, 1]`, the function should return `8`, as it is the largest integer that appears only once.

This is essentially a frequency problem where we need to count how many times each number appears, and then find the largest number that has a frequency of 1.

Intuition

The intuition behind the solution is to keep track of the frequency of each element in the array and then search for the elements with a frequency of 1, starting from the largest potential number and moving downwards. The search stops when we find the first number that meets this criterion, as that would be the largest unique number.

To apply this solution approach efficiently:

1. We use the `Counter` class from Python's `collections` module to quickly count the frequency of each integer in the input array.
2. After we have the frequency of each number, we iterate from the maximum possible integer value down to 0. This ensures that the first integer we find with a frequency of 1 is the largest such integer.
3. We use a generator expression within the `next` function which goes through the numbers in the decreasing order checking for the condition `cnt[x] == 1`.
4. If no integer with a frequency of 1 is found, the `next` function returns `-1` as specified by its default parameter.

With this approach, we can efficiently solve the problem in linear time with respect to the number of elements in the array, which is quite optimal for this type of problem.

Solution Approach

The implementation consists of a few straightforward steps:

1. Import the `Counter` from the `collections` module. The `Counter` is a subclass of `dict` specifically designed to count hashable objects. It's an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values.

2. The `cnt = Counter(nums)` creates a `Counter` object with the frequency of each integer from the `nums` array. For example, if `nums` is `[1, 2, 2, 3]`, `cnt` would be `Counter({2: 2, 1: 1, 3: 1})`.

3. The core of the implementation is the line `return next((x for x in range(1000, -1, -1) if cnt[x] == 1), -1)`. This line uses a generator expression within the `next` function.

   - The generator expression gives us a way to iterate through each integer from `1000` down to `0` (`range(1000, -1, -1)`). We're starting from `1000` because, according to the constraints of the problem, the values in `nums` will not exceed `1000`.
   - For each number `x` in this range, we check if `cnt[x] == 1`. This condition is true only for numbers that occur exactly once in `nums`. If the condition is met, `x` is yielded by the generator.
   - The `next` function is used to find the first item in the sequence that satisfies the condition. If such an item is found, it's returned immediately, making the process efficient because we don't need to count or iterate through the whole range if we've already found our largest unique number.
   - The second argument to `next` is `-1`, which acts as the default value returned if the generator does not yield any value (which would be the case if there are no unique numbers).

This compact and efficient implementation bypasses the need for sorting or additional loops, as it directly makes use of the `Counter` to access the frequency and uses the `range` function in descending order to find the largest unique number.

Example Walkthrough

To illustrate the solution approach, let's take a small example. Suppose our input array is `[4, 6, 2, 6, 4]`.

1. First, we import the `Counter` class from the `collections` module.
2. We then create a `Counter` object to count the frequency of each integer in our array:
   ```
   from collections import Counter
   nums = [4, 6, 2, 6, 4]
   cnt = Counter(nums)  # Counter({4: 2, 6: 2, 2: 1})
   ```
   After creating the `Counter` object, it shows that both 4 and 6 occur twice, and 2 occurs once.
3. We then proceed to find the highest unique integer in `nums` by iterating from the highest possible value (1000) to the lowest in the array, checking if the frequency equals 1. For this example, our array doesn't go up to 1000, so we would just be interested in the range from the highest value in `nums` which is 6, down to the smallest 2:
   ```
   highest_unique = next((x for x in range(6, 1, -1) if cnt[x] == 1), -1)
   ```
4. In our `range`, we start checking from 6 to 2:
   - Check if `cnt[6] == 1`: This is `False` as `cnt[6]` is `2`.
   - Move to the next value, `5`, but it does not exist in our `Counter`, so move on.
   - Check if `cnt[4] == 1`: This is `False` as `cnt[4]` is `2`.
   - Check if `cnt[3] == 1`: As `3` is not in our `Counter`, we move on.
   - Check if `cnt[2] == 1`: This is `True` as `cnt[2]` is `1`.
5. Since we have found that `2` is the largest value that occurs exactly once, we don't need to check any further. We can now return `2` as the result.

The `next` function will yield `2` and since it's the first number that satisfies our condition, this is the value that would be returned from our function call.

If no such unique value is found, the default `-1` will be returned, signaling that there are no elements that occur exactly once.

Thus, in our small example, the function would return `2` as the highest value integer that appears exactly once.

Python Solution

```python
from collections import Counter

class Solution:
    def largestUniqueNumber(self, nums: list[int]) -> int:
        # Count the occurrences of each number in nums
        # using a Counter, which is a dictionary subclass
        number_counts = Counter(nums)

        # Traverse the range from 1000 (inclusive) to -1 (exclusive)
        # in descending order to find the largest unique number
        for num in range(1000, -1, -1):
            # Check if the number appears exactly once
            if number_counts[num] == 1:
                # If the number is unique, return it
                return num

        # Return -1 if no unique number is found
        return -1
```

Java Solution

```java
class Solution {

    // Method to find the largest unique number in an array
    public int largestUniqueNumber(int[] nums) {
        // Array to store the count of each number, assuming the values are within [0, 1000]
        int[] count = new int[1001];

        // Loop through each number in the given array 'nums' and increment its count
        for (int num : nums) {
            count[num]++;
        }

        // Iterate from the largest possible value (1000) down to 0
        for (int i = 1000; i >= 0; i--) {
            // Check if the count of the current number is exactly 1 (unique)
            if (count[i] == 1) {
                // If a unique number is found, return it as it will be the largest one due to the reverse iteration
                return i;
            }
        }

        // If no unique number is found, return -1 as specified by the problem
        return -1;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to find the largest unique number from the vector
    int largestUniqueNumber(vector<int>& nums) {
        // Initialize an array to count occurrences of each number, given the maximal value of 1000.
        int frequency[1001] = {}; // Indexed from 0 to 1000

        // Populate the frequency array with the count of each number from 'nums'.
        for (int num : nums) {
            ++frequency[num];
        }

        // Iterate from the end of the frequency array (starting from the largest possible value - 1000)
        // to find the first number with a frequency of 1 (unique number).
        for (int i = 1000; i >= 0; --i) {
            if (frequency[i] == 1) {
                // If a unique number is found, return it as the largest unique number
                return i;
            }
        }

        // If no unique number is found, return -1.
        return -1;
    }
};
```

Typescript Solution

```typescript
function largestUniqueNumber(nums: number[]): number {
    // Initialize an array of size 1001 to count the occurrences of each number.
    const count = new Array(1001).fill(0);

    // Iterate over the input array and increment the count at the index equal to the number.
    for (const num of nums) {
        ++count[num];
    }

    // Iterate backward from the largest possible number (1000)
    // to find the first number that has a count of 1.
    for (let i = 1000; i >= 0; --i) {
        if (count[i] === 1) {
            return i; // Return the largest unique number.
        }
    }

    // If no unique number is found, return -1.
    return -1;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code consists of two parts: creating the counter and finding the largest unique number.

1. **Creating the Counter:** The `Counter` function from `collections` module is used to count the frequency of each element in the input list `nums`. The time complexity of this operation is `O(n)`, where `n` is the length of the input list `nums`, as it requires a single pass over all elements to count their frequencies.

2. **Finding the Largest Unique Number:** The generator expression inside `next` iterates from `1000` to `0`, which is a constant range, and checks if the count of each number is exactly `1`. The worst-case time complexity of this operation is `O(1)` because we're iterating over a fixed range independent of the input size.

Combining both, the overall time complexity is `O(n + 1)`, which simplifies to `O(n)` because asymptotic analysis drops constant terms.

Space Complexity

The space complexity also consists of two parts: the space used by the `Counter` and the space for the generator expression.

1. **Counter Space:** The `Counter` object will hold at most `n` unique numbers and their counts, so in the worst case, where all numbers in `nums` are unique, the space complexity is `O(n)`.

2. **Generator Expression Space:** The generator expression does not create an additional list; it simply iterates over the range and yields values one by one. Therefore, its space complexity is `O(1)`.

Overall, the space complexity of the algorithm is `O(n)`, dominated by the space required for the `Counter`.