2453. Destroy Sequential Targets

Medium Array Hash Table Counting

Leetcode Link

Problem Description

In this LeetCode problem, we are given an array `nums`, which represents targets on a number line, and a positive integer `space`. We also have a machine that can destroy targets when seeded with a number from `nums`. Once seeded with `nums[i]`, the machine destroys all targets that can be expressed as `nums[i] + c * space`, where `c` is any non-negative integer. The goal is to destroy the maximum number of targets possible by seeding the machine with the minimum value from `nums`.

To put it simply, we want to find the smallest number in `nums` that, when used to seed the machine, results in the most targets being destroyed. Each time the machine is seeded, it destroys targets that fit the pattern based on the `space` value and the chosen seed from `nums`.

Intuition

The intuition behind the solution is to leverage modular arithmetic to find which seed enables the machine to destroy the most targets. Here's the thought process for arriving at the solution:

1. The key insight is that targets are destroyed in intervals of `space` starting from `nums[i]`. So, if two targets `a` and `b` have the same remainder when divided by `space`, seeding the machine with either `a` or `b` will destroy the same set of targets aligned with that spacing.
2. To understand which starting target affects the most other targets, we can count how many targets each possible starting value (based on the remainder when divided by `space`) would destroy.
3. A `Counter` dictionary can be used to maintain the count of how many targets share the same `value % space`.
4. We iterate through `nums` and update the maximum count of targets destroyed (`mx`) and the associated minimum seed value (`ans`), keeping track of the seed that destroys the most targets.
5. If a new maximum count is found, we update `ans` to that new seed value. If we find an equal count, we choose the seed with the lower value to minimize the seed.

By leveraging the frequency of targets, modular arithmetic, and choosing the smallest value with the most targets destroyed, we efficiently solve the problem.

Solution Approach

The implementation of the solution utilizes Python's `Counter` class from the `collections` module along with a straightforward iteration over the list of `nums`. Here's a detailed walk-through of the implementation steps:

1. `cnt = Counter(v % space for v in nums)`: The first step involves creating a `Counter` object to count the occurrences of each unique remainder when dividing the targets in `nums` by `space`. This helps us understand the frequency distribution of how the targets line up with different offsets from 0 when considering the `space` intervals. Each unique remainder represents a potential starting point for the machine to destroy targets.

2. `ans = mx = 0`: Two variables are initialized - `ans` for storing the minimum seed value needed to destroy the maximum number of targets, and `mx` for storing the maximum number of targets that can be destroyed from any given seed value (initially both are set to 0).

3. The for loop - `for v in nums`: This loop iterates through each value `v` in `nums` and checks how many targets can be destroyed if the machine is seeded with `v`. This is found with reference to the remainder `v % space`.

4. `t = cnt[v % space]`: For a given value `v`, the number of targets that can be destroyed from this seed `v` is retrieved from the `Counter` object we created. This gives us the count `t`.

5. `if t > mx or (t == mx and v < ans)`: This conditional block is the core logic that checks if the current count `t` of destroyable targets is greater than the maximum calculated so far `mx` or if it is equal to `mx` but the seed value `v` is smaller than the current answer `ans`.

6. `ans = v` and `mx = t`: If the condition is true, then we update `ans` with `v` because we found a better (smaller) seed value that destroys an equal or greater number of targets compared to the previously stored answer. Likewise, `mx` is updated with `t`, reflecting the new maximum count of destroyable targets.

7. `return ans`: Lastly, after going through all the values in `nums`, the value of `ans` now contains the minimum seed value that can destroy the maximum number of targets when the loop is complete. This value is returned as the final answer.

This approach smartly combines modular arithmetic with frequency counting and an iterative comparison to yield both the maximum destruction and the minimum seed value needed in a highly optimized way.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we are given `nums = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]` and `space = 2`.

Step 1: Calculate remainders and count frequencies.

Using `cnt = Counter(v % space for v in nums)` will result in:

- Remainder 0: {4, 6, 2} (3 targets, because 4, 6, and 2 are divisible by 2)
- Remainder 1: {1, 3, 5, 9} (7 targets, because 1, 3, 5, 5, 3, 5, and 9 leave a remainder of 1 when divided by 2)

So our `Counter` object `cnt` would look like this: `{1: 7, 0: 3}`.

Step 2: Initialize variables for answer and maximum count.

set `ans = 0` and `mx = 0`.

Step 3: Iterate through `nums` and determine the targets that can be destroyed using each number as a seed.

For `v = 3`, `t = cnt[3 % 2] = cnt[1] = 7`. Since `t > mx`, we update `ans` to `3` and `mx` to `7`.

For the next value `v = 1`, `t = cnt[1 % 2] = cnt[1] = 7`. Now, `t == mx`, but since `1 < ans`, we update `ans` to `1`.

We continue this process for all `nums`. Throughout this process, `ans` will potentially be updated when we find a `v` such that either:

- It destroys more targets, or
- It destroys an equal number of targets and is a smaller number than the current `ans`.

Going through the rest of `nums`, we find that no other numbers will update `ans`, as they either have a remainder of 0 or are larger than 1 with a remainder of 1.

Finally, after checking all elements in `nums`, the final values of `ans` and `mx` will be `1` and `7`, respectively.

Step 4: Return the answer.

The function would then return `ans`, which is `1`, as the smallest number from `nums` that can be used to seed the machine to destroy the maximum number of targets, which, in this example, is 7 targets.

By following the implementation steps outlined in the description, we have found the minimum seed value that achieves the maximum destruction of targets according to the given `space`.

Python Solution

```python
from collections import Counter

class Solution:
    def destroyTargets(self, nums: List[int], space: int) -> int:
        # Create a counter to keep track of the frequency of the modulus values
        modulus_frequencies = Counter(value % space for value in nums)

        # Initialize variables to store the most frequent element and its frequency
        most_frequent_element = 0
        max_frequency = 0

        # Loop through the list of numbers
        for value in nums:
            # Get the frequency of the current element's modulus
            current_frequency = modulus_frequencies[value % space]

            # Check if current element's modulus frequency is higher than the max frequency found
            # Or if it's the same but the value is smaller (as per problem's requirement)
            if current_frequency > max_frequency or (current_frequency == max_frequency and value < most_frequent_element):
                # Update the most frequent element and the max frequency
                most_frequent_element = value
                max_frequency = current_frequency

        # Return the most frequent element after checking all elements
        return most_frequent_element

```

Java Solution

```java
class Solution {
    public int destroyTargets(int[] nums, int space) {
        // Creating a map to store the frequency of each space-residual ('residue').
        Map<Integer, Integer> residueFrequency = new HashMap<>();
        for (int value : nums) {
            // Compute the space-residual of the current number.
            int residue = value % space;
            // Increment the count of this residue in the map.
            residueFrequency.put(residue, residueFrequency.getOrDefault(residue, 0) + 1);
        }

        // 'bestNumber' will hold the result, 'maxFrequency' the highest frequency found.
        int bestNumber = 0, maxFrequency = 0;
        for (int value : nums) {
            // Get the frequency of the current number's space-residual.
            int currentFrequency = residueFrequency.get(value % space);
            // Update 'bestNumber' and 'maxFrequency' if a higher frequency is found,
            // or if the frequency is equal and the value is less than the current 'bestNumber'.
            if (currentFrequency > maxFrequency || (currentFrequency == maxFrequency && value < bestNumber)) {
                bestNumber = value;
                maxFrequency = currentFrequency;
            }
        }

        // Return the resultant number i.e., smallest number with the highest space-residual frequency.
        return bestNumber;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>

class Solution {
public:
    // Function to determine the value to be destroyed based on given rules.
    int destroyTargets(vector<int>& nums, int space) {
        // Create a map to store the frequency of each number modulo space.
        unordered_map<int, int> frequency;
        // Fill the frequency map.
        for (int value : nums) {
            ++frequency[value % space];
        }

        int result = 0; // Store the number whose group to be destroyed.
        int maxFrequency = 0; // Keep track of the max frequency found.

        // Iterate over the numbers to find the value with the
        // highest frequency and still respect the rules for ties.
        for (int value : nums) {
            int currentFrequency = frequency[value % space];
            // Check if the current value's frequency is greater than the max frequency so far
            // or if the frequency is the same but the value is smaller (for tie-breaking).
            if (currentFrequency > maxFrequency || (currentFrequency == maxFrequency && value < result)) {
                result = value; // Update the result with the current value.
                maxFrequency = currentFrequency; // Update the max frequency.
            }
        }

        // Return the number whose group will be destroyed.
        return result;
    }
};

```

Typescript Solution

```typescript
// Import statements for TypeScript (if needed in your environment)

interface FrequencyMap {
    [key: number]: number;
}

// Function to determine the value to be destroyed based on given rules.
function destroyTargets(nums: number[], space: number): number {
    // Create a map to store the frequency of each number modulo space.
    const frequency: FrequencyMap = {};
    // Fill the frequency map.
    nums.forEach(value => {
        const modValue = value % space;
        frequency[modValue] = (frequency[modValue] || 0) + 1;
    });

    let result = 0; // Store the number whose group is to be destroyed.
    let maxFrequency = 0; // Keep track of the max frequency found.

    // Iterate over the numbers to find the value with the
    // highest frequency and still adhere to the rules for ties.
    nums.forEach(value => {
        const modValue = value % space;
        const currentFrequency = frequency[modValue];
        // Check if the current value's frequency is greater than max frequency so far
        // or if the frequency is the same but the value is smaller (for tie-breaking).
        if (currentFrequency > maxFrequency || (currentFrequency === maxFrequency && value < result)) {
            result = value; // Update the result with the current value.
            maxFrequency = currentFrequency; // Update the max frequency.
        }
    });

    // Return the number whose group will be destroyed.
    return result;
}

// Example usage:
// const nums = [2, 12, 4, 6, 8];
// const space = 10;
// console.log(destroyTargets(nums, space)); // Output would depend on the input array 'nums'

```

Time and Space Complexity

The given Python code defines a method `destroyTargets`, which takes a list of integers `nums` and an integer `space`, and returns the value of the integer from the list that has the highest number of occurrences mod `space`, with the smallest value chosen if there is a tie.

To analyze the time complexity:

1. We start by creating a counter `cnt` for occurrences of each `v % space`, where `v` is each value in `nums`. The `Counter` operation has a time complexity of `O(n)` where `n` is the number of elements in the list `nums`, as it requires going through each element in the list once.

2. Next, we run a for loop through each value in `nums`, which means the loop runs `n` times.

3. Inside the loop, we perform a constant time operation checking if `t` (which is `cnt[v % space]`) is greater than `mx` or if it is equal to `mx` and `v` is less than `ans`, and perform at most two assignments if the condition is true.

Therefore, the loop has a time complexity of O(n), where each iteration is O(1) but we do it n times. When you combine the loop with the initial counter creation, the overall time complexity remains O(n) because both are linear operations with respect to the size of `nums`.

For space complexity:

1. The `Counter` object `cnt` will at most have a unique count for each possible value of `v % space`, which is at most `space` different values. So the space complexity for `cnt` is `O(space)`.

2. No additional data structures grow with input size `n` are used; therefore, the space complexity is not directly dependent on `n`.

Considering both the Counter object and some auxiliary variables (`ans`, `mx`, `t`), the overall space complexity is `O(space)`.

In summary:

- Time Complexity: `O(n)`
- Space Complexity: `O(space)`