1005. Maximize Sum Of Array After K Negations

Easy Greedy Array Sorting Leetcode Link

Problem Description

When given an integer array `nums` and an integer `k`, we're tasked to carry out a specific modification process to the array a total of `k` times. This modification involves choosing an index `i` and then flipping the sign of the number at that index (turning a positive number into a negative one or vice versa). The goal is to maximize the sum of elements in the array after making exactly `k` sign changes.

Intuition

The thinking process behind the solution is to first increase the sum of the array. We do this by flipping the sign of negative numbers because converting negatives to positives contributes to an increase in the total sum. The priority is to flip the smallest negative numbers (largest absolute value), as this will have the most significant impact on the sum.

The solution approach involves:

1. Using a `Counter` from the `collections` module to count the occurrences of each number in the array. This is a more efficient way to handle duplicates and manage the operations rather than sorting or manually manipulating the array.
2. We first consider all negative numbers, starting with the smallest (closest to -100, since the problem limits numbers to the range [-100, 100]). If a negative number exists, we flip it to positive and decrease our `k` accordingly. This is only beneficial if `k` remains non-zero after the operation.
3. If we still have `k` operations left after dealing with all negative numbers, we now look at `k`'s parity (whether it is even or odd). If `k` is even, we can ignore the rest of the operations since flipping any number twice will just return it to its original state (a no-op in terms of the array sum). If `k` is odd, we need to perform one more flip to maximize the sum.
4. In the case where `k` is odd, we flip the smallest positive number (including zero, if present). We do this because, after all negatives have been flipped (if `k` allowed), this will have the smallest impact on decreasing the sum (since we have to perform an odd number of flips).
5. After modifying the array according to the above rules, we calculate and return the sum of the final numbers, which represents the largest sum we can achieve.

The usage of `Counter` and flipping based on the smallest absolute values allows us to perform the minimum number of operations to achieve the highest sum.

Solution Approach

The given Python solution to maximize the sum of the integer array after `k` negations makes use of several programming concepts, including:

- A **counter**: A `Counter` from the `collections` module is utilized to maintain a count of each distinct element present in the array.
- **Looping through a range of numbers**: A loop is employed to iterate through a range of numbers from -100 to -1 (inclusive), corresponding to potential negative numbers in `nums`.
- **Conditional checks and updates**: Inside this loop, the algorithm checks for the presence of a negative number `x` in `nums` (determined by `if cnt[x]`) and calculates how many times this number should be flipped using `min(cnt[x], k)`. If a flip is possible, the counter for `x` is decreased, while the counter for `-x` (the positive counterpart) is increased. The number of remaining flips `k` is decremented by the number of flips made, and if `k` drops to zero, the loop breaks as no more flips are permitted.
- **Handling the parity of the number of flips** `k`: After negating as many negative numbers as possible, if there is an odd number of flips remaining (`k & 1`), and there are no zeros in the array to negate (since negating zero does not affect the sum), the algorithm looks for the smallest positive number (in the range of 1 to 100) to negate.
- **Summation of the array**: Ultimately, the sum of the elements in the array is calculated using list comprehension and the items in the counter. The product of `x * v` for each number `x` and its count `v` is summed up to obtain the final result.

The algorithm is efficient as it prioritizes flipping the most significant negative numbers, handles the parity of `k` wisely, and performs a minimal number of operations by skipping unnecessary flips when `k` is even. Additionally, by employing a `Counter`, the solution avoids redundant re-computation by smartly tracking and updating the count of each number after each operation.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the array `nums = [3, -4, -2, 5]` and `k = 3`. Our goal is to maximize the sum of the array after making exactly `k` flips.

We start by using a `Counter` to count occurrences:

```
Counter({3: 1, -4: 1, -2: 1, 5: 1})
```

Step by step, we perform the following operations:

1. Flip the smallest negative number. The smallest (in terms of value) negative number is `-4`. So, we flip it to `4`. Now, `nums = [3, 4, -2, 5]`, `k = 2`, and the `Counter` updates to `{3: 1, -4: 0, 4: 1, -2: 1, 5: 1}`.
2. We still have flips left (`k > 0`), so we flip the next smallest negative number `-2` to `2`. Now `nums = [3, 4, 2, 5]`, `k = 1`, and the `Counter` becomes `{3: 1, -2: 0, 2: 1, 4: 1, 5: 1}`.
3. With `k` now equal to `1` (which is odd), we need to perform one more flip. We look for the smallest positive number, which is `2`, and flip it back to `-2` (if there were a zero, we would flip that instead as it would not affect the sum). Now `nums = [3, 4, -2, 5]` and `k = 0`. The `Counter` updates to `{3: 1, 2: 0, -2: 1, 4: 1, 5: 1}`.
4. No more flips remain; `k = 0`. We calculate the sum of the array using the `Counter`. The array now looks like `[3, 4, -2, 5]`, yielding a sum of `10`.

So the maximum sum we can achieve with `k = 3` flips for the array `[3, -4, -2, 5]` is `10`. The solution approach effectively prioritizes the flips to maximize the sum by first flipping the smallest negatives and then handling the case where an odd number of flips is left by flipping the smallest positive number.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def largestSumAfterKNegations(self, nums: List[int], k: int) -> int:
        # Count the occurrences of each number in the array.
        num_counter = Counter(nums)

        # Iterate over negative numbers since negating negatives can increase total sum.
        for x in range(-100, 0):
            if num_counter[x]:
                # Determine the minimum between the count of the current number and k.
                num_negations = min(num_counter[x], k)
                num_counter[x] -= num_negations  # Decrease the count for the current number.
                num_counter[-x] += num_negations  # Increase the count for the opposite positive number.
                k -= num_negations  # Decrease k by the number of negations performed.
                if k == 0:  # Break if all negations have been used up.
                    break

        # If there are an odd number of negations left and no zero in the list,
        # it's optimal to flip the smallest positive number (if it exists).
        if k % 2 == 1 and num_counter[0] == 0:
            for x in range(1, 101):  # Look for the smallest positive number.
                if num_counter[x]:
                    num_counter[x] -= 1  # Decrease its count.
                    num_counter[-x] += 1  # Increase the count for its negative.
                    break

        # Calculate the final sum.
        return sum(x * occurrence for x, occurrence in num_counter.items())

```

Java Solution

```java
class Solution {

    public int largestSumAfterKNegations(int[] nums, int k) {
        // Create a frequency map to store the occurrence of each number
        Map<Integer, Integer> frequency = new HashMap<>();
        for (int num : nums) {
            frequency.merge(num, 1, Integer::sum);
        }

        // Negate K numbers starting with the smallest (most negative) number
        for (int i = -100; i < 0 && k > 0; ++i) {
            if (frequency.getOrDefault(i, 0) > 0) {
                // Determine the number of negations allowed for the number i
                int negations = Math.min(frequency.get(i), k);
                // Decrease the count for this number after negations
                frequency.merge(i, -negations, Integer::sum);
                // Increase the count for its positive pair after negations
                frequency.merge(-i, negations, Integer::sum);
                // Decrease the number of remaining negations
                k -= negations;
            }
        }

        // If there is an odd number of negations left and 0 is not present,
        // we must negate the smallest positive number
        if ((k % 2 == 1) && frequency.getOrDefault(0, 0) == 0) {
            for (int i = 1; i <= 100; ++i) {
                if (frequency.getOrDefault(i, 0) > 0) {
                    // Negate one occurrence of the smallest positive number
                    frequency.merge(i, -1, Integer::sum);
                    // Add one occurrence of its negation
                    frequency.merge(-i, 1, Integer::sum);
                    break;
                }
            }
        }

        // Calculate the sum of all numbers after the negations
        int sum = 0;
        for (Map.Entry<Integer, Integer> entry : frequency.entrySet()) {
            sum += entry.getKey() * entry.getValue();
        }

        return sum;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int largestSumAfterKNegations(vector<int>& nums, int k) {
        unordered_map<int, int> countMap; // Map to store the frequency of each number

        // Count the frequency of each number in the array
        for (int number : nums) {
            ++countMap[number];
        }

        // Negate the negative numbers if possible, starting from the smallest
        for (int x = -100; x < 0 && k > 0; ++x) {
            if (countMap[x]) { // If there are occurrences of 'x'
                int times = min(countMap[x], k); // Find the min between count and remaining k
                countMap[x] -= times; // Decrease the count for 'x'
                countMap[-x] += times; // Increase the count for '-x', effectively negating 'x'
                k -= times; // Decrease k by the number of negations performed
            }
        }

        // If there are remaining negations 'k' and there's no zero in the array
        if (k % 2 == 1 && !countMap[0]) {
            // Find the smallest positive number to negate
            for (int x = 1; x <= 100; ++x) {
                if (countMap[x]) {
                    --countMap[x]; // Decrement the count of this number
                    ++countMap[-x]; // Increment the count of its negation
                    break; // Only negate once, then break
                }
            }
        }

        // Calculate the final sum after the possible negations
        int sum = 0;
        for (const auto& [value, frequency] : countMap) {
            sum += value * frequency; // Sum = number * its frequency
        }

        return sum; // Return the final sum
    }
};

```

Typescript Solution

```typescript
// This function calculates the largest sum we can achieve by negating K elements
// in the given array 'nums'.
function largestSumAfterKNegations(nums: number[], k: number): number {
    // Create a frequency map to count occurrences of each number
    const frequency: Map<number, number> = new Map();

    // Fill the frequency map with the numbers from 'nums'
    for (const number of nums) {
        frequency.set(number, (frequency.get(number) || 0) + 1);
    }

    // Process the numbers from -100 to -1
    for (let number = -100; number < 0 && k > 0; ++number) {
        // If the current number has a frequency greater than 0 and we still have k negations
        if (frequency.get(number)! > 0) {
            // Determine how many negations we can perform (limited by k and the frequency)
            const negations = Math.min(frequency.get(number) || 0, k);

            // Decrease the frequency of the current number by the negations performed
            frequency.set(number, (frequency.get(number) || 0) - negations);

            // Increase the frequency of the number's positive counterpart
            frequency.set(-number, (frequency.get(-number) || 0) + negations);

            // Decrease the count of remaining negations
            k -= negations;
        }
    }

    // Check for remaining negations; if we have an odd number and no zero is present in 'nums'
    if ((k & 1) === 1 && (frequency.get(0) || 0) === 0) {
        // Apply the negation to the smallest positive number
        for (let number = 1; number <= 100; ++number) {
            if (frequency.get(number)! > 0) {
                // Decrease the frequency of the smallest positive number by 1
                frequency.set(number, (frequency.get(number) || 0) - 1);

                // Increase the frequency of the number's negative counterpart by 1
                frequency.set(-number, (frequency.get(-number) || 0) + 1);
                break;
            }
        }
    }

    // Calculate the sum of all numbers, taking their frequencies into account
    let totalSum = 0;
    for (const [num, freq] of frequency.entries()) {
        totalSum += num * freq;
    }

    // Return the computed total sum
    return totalSum;
}

```

Time and Space Complexity

Time Complexity

The given code has a time complexity of `O(N + K)` where `N` is the size of the input list `nums`.

Here's the breakdown:

- Counting the elements into the Counter object `cnt` takes `O(N)` time.
- The first `for` loop runs for at most 100 iterations (from -100 to -1), which is a constant, hence `O(1)`. However, within this loop, operations take place `min(cnt[x], k)` times, which could approach `k`. Therefore, in the worst case where all elements are negative and k is large, it could be `O(K)`.
- The check for odd `k` when there's no zero in the array, and then the subsequent for loop to find the smallest positive number, runs in at most 100 iterations again. Therefore, it consumes constant `O(1)` time.
- Finally, summing up the elements in the `cnt` takes `O(N)` as it has to iterate through all elements once.

Therefore, we combine these to find `O(N + K + 1 + 1)`, which simplifies to `O(N + K)`.

Space Complexity

The space complexity of the code is `O(N)` because:

- The Counter object `cnt` contains at most `N` unique integers, which is equal to the size of the input list `nums`.
- No other data structures are used that scale with the size of the input.

Thus, the code requires additional space proportional to the number of unique elements in `nums`, which at maximum could be all the elements in `nums`, so `O(N)`.