2702. Minimum Operations to Make Numbers Non-positive

Hard Array Binary Search Leetcode Link

Problem Description

In this problem, we have an array of non-negative integers called `nums` and two positive integers `x` and `y`. We are allowed to perform operations on the array with the aim of reducing all the elements to zero or less. In each operation, we can choose an index `i` and decrease the element at that index `nums[i]` by `x` and simultaneously decrease all other elements by `y`. The task is to find and return the minimum number of such operations required to achieve the target condition where every element of the array is less than or equal to zero.

Intuition

To arrive at the solution, we need to recognize that decreasing all elements in the array besides the chosen index `i` by `y` in each operation will set a pace for how quickly the array can reach `<= 0`. This pace is constrained by the smallest element since it will hit zero first due to the decrement by `y` in each operation. The chosen element decremented by `x` will go down faster, so we want to figure out how often we need to 'hit' an element with the `x` decrement to bring it down to zero along with the rest of the elements.

A sound approach here is to perform a binary search on the number of operations `t` we would need since the answer will range between `0` and the maximum element in `nums`. For any guess `t`, we calculate if it's possible to bring all elements to less than or equal to zero within `t` operations. To do this, for each `v` in nums, we check if `v > t * y`, meaning the decay by `y` alone wouldn't be enough. If so, we need additional decrements of `x - y` to bring it down, which we calculate using `ceil((v - t * y) / (x - y))`. If the sum of additional decrements for all elements is less than or equal to `t`, we've found a viable candidate for the number of operations, and we can try to lower our guess. If not, we need to increase it. The first `t` that satisfies the condition for all elements in `nums` will be our answer.

This binary search loop continues until we narrow down the number of operations needed to a single value, which is then returned as the minimum number of operations required.

Solution Approach

The solution uses a binary search algorithm to efficiently narrow down the minimum number of operations. The reasoning behind using binary search is that we're dealing with a monotonic condition: if we can bring down all elements to zero or less in `t` operations, we can also do it in `t+1`, `t+2`, etc. So, we try to find the smallest `t` for which the condition holds true.

Here are the steps of the implemented solution:

1. Define an inner function `check(t: int) -> bool` that will verify if we can achieve the target with `t` operations. It performs the following steps:
    - Initialize a counter variable `cnt` to 0.
    - Loop through each element `v` in `nums`.
    - Within the loop, check if `v` is greater than `t * y`. If it is, calculate the extra decrements needed by subtracting `t * y` from `v`, and then dividing the rest by `x - y` using `ceil`. This ensures that we take into account the integer division rounding.
    - Increment `cnt` by the number computed. This represents the number of times `x` decrement should be applied to each element.
    - If the sum of decrements `cnt` is less than or equal to `t`, return `True`; otherwise return `False`.
2. Conduct the binary search between `l = 0` and `r = max(nums)`, as the answer is in this range:
    - While `l < r`, calculate `mid` which is the midpoint of `l` and `r`.
    - Call `check(mid)` to verify if `t = mid` is a viable number of operations.
    - If `check(mid)` returns `True`, we can potentially reduce the number of operations, so set `r` to `mid`.
    - Else, if `check(mid)` returns `False`, we need more operations to reach the target, so set `l` to `mid + 1`.
3. The loop ends when `l` is equal to `r`, meaning we've found the smallest possible `t` where `check(t)` is `True`. Return `l` as the minimum number of operations required.

The reason we use `(l + r) >> 1` to calculate the midpoint is that it is equivalent to `(l + r) / 2`, but faster in terms of execution as it is a bit shift operation.

The time complexity of this solution is `O(nlog(maxElement))`, where `n` is the number of elements in `nums`. The log factor comes from the binary search, and for each step of the search, we iterate through the array once to check if the current `t` is viable. The space complexity is `O(1)` since we only use a few variables for the binary search and the checks.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have an array `nums = [3, 19, 10]` and the integers `x = 7` and `y = 5`.

The target condition is to reduce all elements to zero or less. According to the solution approach, we can perform a binary search to find the minimum number of operations, `t`.

Step by step, here's how the solution works with this example:

1. We estimate the bounds [`l`, `r`] for the binary search. The upper bound `r` can be the maximum element in `nums`, which is 19. We start with `l = 0`.

2. We define our `check` function, which will determine if a given number of operations `t` is sufficient to reduce all the elements in `nums` to zero or less.

3. We begin the binary search. With `l = 0` and `r = 19`, we calculate the midpoint `mid` to check. `mid` is `(0 + 19) >> 1 = 9`.

4. We call `check(9)` to verify if `t = 9` is a viable number of operations:
    - For `nums[0] = 3`, since `3 <= 9 * 5`, we don't need any extra decrements because `y` decrements alone will bring it down to zero.
    - For `nums[1] = 19`, since `19 > 9 * 5`, we do the math `ceil((19 - 9 * 5) / (7 - 5)) = ceil((19 - 45) / 2) = ceil(-26 / 2) = 0`. So no extra decrements are needed.
    - For `nums[2] = 10`, since `10 > 9 * 5`, we do `ceil((10 - 9 * 5) / (7 - 5)) = ceil((10 - 45) / 2) = ceil(-35 / 2) = 0`.
    - Sum of extra decrements needed is `0` which is `<= 9`, so `check(9)` returns `True`.

5. Since `check(9)` returns `True`, we can potentially reduce the number of operations. Now our new upper bound `r` is set to our current `mid`, which is 9. We repeat the binary search with new bounds `l = 0` and `r = 9` and find the midpoint again.

6. After some iterations, we might find that lower values of `t` also pass the check function (in this example, `t = 1` could also work since none of the elements need extra decrements). Binary search will continue until `l == r`.

7. Once we find the smallest `t` such that `check(t)` is `True`, that will be our answer for the minimum number of operations required.

This detailed example walkthrough shows us how the binary search can efficiently find the minimum number of operations to reduce all array elements to zero or less. In this instance, given `x` and `y` are much larger than the array elements, even one operation is sufficient—hence the answer is `t = 1`.

Python Solution

```python
from math import ceil
from typing import List

class Solution:
    def minOperations(self, nums: List[int], x: int, y: int) -> int:
        # Helper function to check if the operation can be completed with 't' operations.
        def is_possible_with_t_operations(t: int) -> bool:
            # Initialize count of operations.
            operations = 0
            # Iterate through each number in the list.
            for value in nums:
                # If the value is greater than 't' times 'y', operations are needed.
                if value > t * y:
                    # Calculate the number of operations needed for the current value, and add to the total.
                    operations += ceil((value - t * y) / (x - y))
            # Return True if the counted operations are less than or equal to 't', else return False.
            return operations <= t

        # Initialize left and right pointers for the binary search.
        left, right = 0, max(nums)
        # Perform binary search to find the minimum number of operations required.
        while left < right:
            # Get the middle value between left and right.
            mid = (left + right) >> 1  # Using bitwise shift to divide by 2.
            # If the operation is possible with 'mid' operations, move the right boundary to 'mid'.
            if is_possible_with_t_operations(mid):
                right = mid
            # Otherwise, move the left boundary past 'mid'.
            else:
                left = mid + 1
        # The left pointer will now point to the minimum number of operations needed.
        return left

```

Java Solution

```java
class Solution {
    private int[] numbers;
    private int x;
    private int y;

    // This method aims to find the minimum number of operations required.
    // It uses a binary search approach to find the minimum 't'.
    public int minOperations(int[] numbers, int x, int y) {
        this.numbers = numbers;
        this.x = x;
        this.y = y;

        // Initializing search range for 't'.
        int left = 0;
        int right = 0;

        // Find the largest number in the array to set the upper bound for 't'.
        for (int value : numbers) {
            right = Math.max(right, value);
        }

        // Perform binary search to find the minimum 't'.
        while (left < right) {
            int mid = (left + right) >>> 1;  // Calculate the mid value for 't'.
            if (isFeasible(mid)) {
                right = mid;  // Found new possible minimum, adjust right bound.
            } else {
                left = mid + 1;  // Target not feasible, adjust left bound.
            }
        }
        return left;  // Return the minimum number of operations required.
    }

    // This helper method checks if it's possible to achieve the goal
    // with 't' operations, where each operation reduces a number in the array by x until it's no smaller than t * y.
    private boolean isFeasible(int target) {
        long count = 0;  // Initialize the count of operations.

        // Check each number in the array to see if operations are needed.
        for (int value : numbers) {
            if (value > (long) target * y) {
                // Calculate operations required for current value and add to count.
                count += (value - (long) target * y + x - y - 1) / (x - y);
            }
        }
        // Check if the count of operations is less than or equal to target to meet the constraint.
        return count <= target;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    int minOperations(std::vector<int>& nums, int increment, int decrement) {
        // Using binary search to find the minimum number of operations
        // Initialize the left (l) and right (r) boundaries for binary search
        int left = 0, right = *std::max_element(nums.begin(), nums.end());

        // Lambda to check if a given number of operations (ops) is sufficient
        auto isSufficient = [&](int ops) {
            long long requiredOps = 0;
            for (int value : nums) {
                if (value > 1LL * ops * decrement) {
                    // Calculate additional operations needed for this value
                    requiredOps += (value - 1LL * ops * decrement + increment - decrement - 1) / (increment - decrement);
                }
            }
            return requiredOps <= ops; // Return true if the total required operations are within the limit
        };

        // Binary search to find the minimum number of operations required
        while (left < right) {
            int mid = (left + right) >> 1; // Calculate the mid-point
            if (isSufficient(mid)) {
                right = mid; // If mid operations are sufficient, we look for a potential smaller number
            } else {
                left = mid + 1; // If mid operations are not sufficient, we need more operations
            }
        }

        return left; // Once the search is narrowed down to a single element, left will be the minimum number of operations
    }
};

```

Typescript Solution

```typescript
function minOperations(nums: number[], x: number, y: number): number {
    // Initialize pointers l and r for the binary search
    let left = 0;
    let right = Math.max(...nums);

    // Define a check function that will return true if the operation is possible within t actions
    const check = (threshold: number): boolean => {
        // Initialize a counter for the number of operations performed
        let operations = 0;

        // Iterate over each value in the array
        for (const value of nums) {
            // If the value is greater than the threshold times y, calculate the necessary operations
            if (value > threshold * y) {
                // Add the ceil of the division to the operations' counter
                operations += Math.ceil((value - threshold * y) / (x - y));
            }
        }

        // Return true if the total operations are less than or equal to the threshold
        return operations <= threshold;
    };

    // Use binary search to find the minimum number of operations to reach a certain state
    while (left < right) {
        // Find the mid-point of the current search range
        const mid = (left + right) >> 1;

        // If the current mid value satisfies the condition, adjust the right pointer
        if (check(mid)) {
            right = mid;
        } else {
            // Otherwise, adjust the left pointer
            left = mid + 1;
        }
    }

    // Return the minimum number of operations after binary search completes
    return left;
}

```

Time and Space Complexity

The given code snippet is implementing a binary search algorithm to find the minimum number of operations needed to reduce certain elements in the array `nums` so that no element is greater than `t * y`. To determine if a given target `t` is feasible, the `check` function is employed, which computes the number of operations needed for each element in `nums`. If the element is greater than `t * y`, the number of operations to reduce it is added to `cnt`.

The `check` function is called with a time complexity of `O(n)` where `n` is the number of elements in `nums`. This is because it iteratively checks each element once.

The binary search runs from 0 to the maximum value in `nums`. The time complexity for binary search is `O(log m)`, where `m` is the range within which the binary search operates. Here `m` corresponds to the maximum value in `nums`.

The overall time complexity of the code is therefore `O(n log m)`, since the binary search is the outer loop and the `check` function is called within each iteration of this binary search.

As for the space complexity, the space used is constant and does not depend on the input size, besides the input array itself. No additional data structures that grow with input size are created. Thus, the space complexity is `O(1)`.