2366. Minimum Replacements to Sort the Array

Hard Greedy Array Math Leetcode Link

Problem Description

You are provided with an array of integers called `nums`. The goal is to sort this array into non-decreasing order (where each number is less than or equal to the next) by performing a specific operation as many times as needed. The operation you can perform involves replacing a single element in the array with any two elements that sum up to the original element.

For instance, if you have an element `6` in the array, you can replace it with two elements `2` and `4` since `2 + 4 = 6`. The challenge lies in figuring out the minimum number of such operations required to sort the entire array.

Intuition

The solution to this problem hinges on a key observation: replacing any number with two smaller numbers can potentially make sorting the array harder, as it introduces more elements that need to be in non-decreasing order. Therefore, we should aim to perform each replacement in a way that either maintains the current order or requires the least amount of subsequent operations.

To minimize operations, we work our way from the end of the array (the largest elements in a sorted array) to the beginning, adjusting each element so that it's not larger than the one already considered and positioned at the end.

As we move backwards:

1. We want the current element to be less than or equal to the element after it (since the array must be in non-decreasing order).
2. If the current element is already less than or equal to the next element, no operation is needed, and we move to the previous element.
3. If the current element is larger, we need to split it into smaller numbers in a way that requires the minimum number of splits while ensuring that each new number is not larger than the next element.

The algorithm keeps track of the maximum allowed value (`mx`) for each replacement, which initially is the value of the last element. It computes the minimum number of splits (`k`) needed for the current element to satisfy the non-decreasing order constraint, and updates the answer (`ans`) with the number of operations needed (which is `k - 1`, since replacing one number with two requires one operation). After this, it updates the maximum allowed value for the next element accordingly.

The process continues until we've considered all elements of the array, and the minimum number of operations required to achieve a non-decreasing array is returned.

Solution Approach

To implement the solution, we follow a reverse iteration. We initiate by setting the variable `mx` to the value of the last element in the array since we aim to ensure that all preceding numbers are less than or equal to this value.

The main logic resides in a for-loop that starts from the second-to-last element and moves towards the first element (index 0). Here's a breakdown:

1. **For-loop iteration**: The loop starts at index `n - 2` because we're comparing each element with the one after it. We decrement our index with every iteration, essentially moving from the end of the array to the start.

2. **Conditional Operation**: For each number, we check if it's less than or equal to `mx`. If it is, we're already maintaining the non-decreasing order, so we set `mx` to the current number (since this can be the new maximum for the upcoming previous elements) and continue to the next iteration without any further action.

   ```
   if nums[i] <= mx:
       mx = nums[i]
       continue
   ```

3. **Calculation of Splits (`k`)**: When the current number is greater than `mx`, we calculate `k`, the minimum number of parts we need to split the current number into. This must be done such that each part does not exceed the value of `mx`. The calculation of `k` is given by `(nums[i] + mx - 1) // mx`. We add `mx - 1` before integer division to ensure that all parts will be as large as possible without exceeding `mx`.

4. **Updating Answer**: The number of new elements added (which is equal to the number of operations performed) will be `k - 1`. This value is added to `ans`, the variable tallying the minimum number of operations.

   ```
   ans += k - 1
   ```

5. **Updating `mx` for the next iteration**: We update `mx` to the largest possible value that a part can take after the split, which is `nums[i] // k`. This new value of `mx` will now act as the upper limit for the next (actually the previous since we're iterating in reverse) number in the array.

6. **Returning the result**: Once the loop has been completed, the variable `ans` will hold the minimum number of operations needed to sort the array, which is returned as the result.

The algorithm does not require any additional data structures; it operates in-place, using only a few additional variables for keeping track of the state (`ans`, `mx`, `k`). It's a simple, yet efficient solution with a time complexity of O(n) as it requires a single iteration through the array.

Example Walkthrough

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

```
nums = [10, 5, 13]
```

We need to sort this array into non-decreasing order by replacing elements into sums as needed. Let's walk through the procedure step by step.

1. We start by setting `mx` to the value of the last element in `nums`, which is `13`. This is the maximum allowed value for its preceding elements.

2. We then begin iterating from the second-to-last element, which is `5` at index `1`.

   - For `nums[1]` which is `5`, since `5` ≤ `mx` (`13`), we do not need to perform any operations. We set `mx` to `5` (since `5` now becomes the maximum allowed value for the preceding element) and continue.

3. Next, we check `nums[0]`, which is `10`. Since `10` > `mx` (`5` now), we need to perform operations. Calculating `k` gives us:

   ```
   k = (nums[0] + mx - 1) // mx = (10 + 5 - 1) // 5 = 2
   ```

   Since `k` is `2`, it means we need to split `10` into two parts, each not exceeding `5` (the current `mx`). We can split it into `[5, 5]`. The number of new elements added is `k - 1` which is `1`. Thus, we have `1` operation performed.

4. We update `ans` with the number of operations performed, so `ans` becomes `1`.

5. We update `mx` to the largest value possible after the splits, which is `nums[0] // k = 10 // 2 = 5`.

6. Having iterated over all elements, we conclude that we needed a minimum of `1` operation to sort the array into non-decreasing order. So the answer (`ans`) is `1`.

This is the result of using the approach described in the solution. The process is time-efficient as it iterates through the array just once, making the complexity O(n).

Python Solution

```python
class Solution:
    def minimumReplacement(self, nums: List[int]) -> int:
        # Initialize the count of replacements to 0.
        replacement_count = 0

        # Get the number of elements in the nums list.
        num_elements = len(nums)

        # Set the current maximum to the last element in the nums list.
        # This maximum represents the highest number we can decrease to without making any replacements.
        current_max = nums[-1]

        # Iterate through the list in reverse order, starting from the second to last element.
        for i in range(num_elements - 2, -1, -1):
            # If the current element is less than or equal to the current maximum,
            # no replacements are needed. Update the current maximum to this element.
            if nums[i] <= current_max:
                current_max = nums[i]
            else:
                # If the current element is greater than the current maximum,
                # we calculate the minimum number of replacements required.
                # This is done by dividing the current element by the current maximum,
                # and rounding up to ensure we get a value no larger than the current max.
                # Then compute how many replacements are needed to reach this value.
                replacements_needed = (nums[i] + current_max - 1) // current_max
                replacement_count += replacements_needed - 1   # Increase the count of replacements

                # Update the current maximum to the value obtained by evenly dividing
                # the current element by the number of replacements needed.
                # This will be the new threshold for further calculations on previous elements.
                current_max = nums[i] // replacements_needed

        # Return the total count of replacements required to make the array non-increasing.
        return replacement_count

```

Java Solution

```java
class Solution {

    // Method to find the minimum number of replacements to make the array non-increasing.
    public long minimumReplacement(int[] nums) {
        // Initialize the answer to accumulate the number of replacements.
        long replacements = 0;
        // Get the number of elements in the array.
        int n = nums.length;
        // Initialize the max value to the last element in the array (it's already in correct position).
        int maxValue = nums[n - 1];

        // Loop through the array from second-to-last to the first element.
        for (int i = n - 2; i >= 0; --i) {
            // If the current element is less than or equal to the max value,
            // it is already in right position, thus move to the previous element.
            if (nums[i] <= maxValue) {
                maxValue = nums[i]; // Update the max value to the current element.
                continue;
            }

            // If the current element is larger than the max value, calculate the number of parts
            // this element needs to be split into to maintain non-increasing order.
            int parts = (nums[i] + maxValue - 1) / maxValue;
            // Update the replacements count by adding the number of new elements added
            // (parts - 1 means how many splits we do, which equals to additional numbers introduced).
            replacements += parts - 1;
            // The new max value should be the average of the current element
            // after replacing it with 'parts' equal or almost equal numbers.
            maxValue = nums[i] / parts;
        }

        // Return the total number of replacements done to make the array non-increasing.
        return replacements;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    long long minimumReplacement(std::vector<int>& nums) {
        long long operations = 0; // Stores the total number of operations required
        int size = nums.size(); // Size of the input vector
        int maxElement = nums[size - 1]; // Initialize maxElement with the last item in the vector

        // Iterate from the second to last element to the beginning
        for (int i = size - 2; i >= 0; --i) {
            // If the current element is less than or equal to maxElement,
            // it's already in the correct order, update maxElement if necessary
            if (nums[i] <= maxElement) {
                maxElement = nums[i];
                continue;
            }

            // Calculate the minimum number of replacements needed for nums[i]
            // such that each replaced number is less than or equal to maxElement
            int replacements = (nums[i] + maxElement - 1) / maxElement;

            // The actual replacements will be one less than the calculated
            // replacements since we are also including the current element
            operations += replacements - 1;

            // Update maxElement to the value of the largest possible replaced number
            maxElement = nums[i] / replacements;
        }

        // Return the total number of operations required to make the array
        // non-decreasing by replacing some numbers with multiple numbers.
        return operations;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the minimum number of replacements needed such that for every i,
 * nums[i] is greater than or equal to nums[i + 1].
 *
 * @param {number[]} nums - Array of numbers to be modified.
 * @return {number} - The minimum number of replacements needed.
 */
function minimumReplacement(nums: number[]): number {
    // Get the length of the array
    const length = nums.length;

    // Initialize variable to store the current lowest number from the back
    let currentMin = nums[length - 1];

    // Initialize variable to store the answer
    let replacements = 0;

    // Iterate from the second-to-last element down to the first element
    for (let i = length - 2; i >= 0; --i) {
        // If the current element is less than or equal to the current lowest number, no need to replace
        if (nums[i] <= currentMin) {
            currentMin = nums[i];
            continue;
        }
        // Calculate how many times the current number needs to be divided
        // to be less than or equal to the current lowest number.
        const factor = Math.ceil(nums[i] / currentMin);

        // Accumulate the total number of replacements needed
        replacements += factor - 1;

        // Update the current lowest number to be the divided number
        currentMin = Math.floor(nums[i] / factor);
    }

    // Return the total number of replacements
    return replacements;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code provided is `O(n)` where `n` is the length of the input list `nums`. This is because the algorithm iterates through the list once in reverse, beginning from the penultimate element to the first element. The operations within each iteration of the loop take constant time, such as comparison, arithmetic operations, and variable assignments. There are no nested loops or additional function calls that would change the linearity of the time complexity.

Space Complexity

The space complexity of the code provided is `O(1)`. The algorithm uses a fixed amount of extra space regardless of the input size. Only a few single-value variables (`ans`, `n`, `mx`) are used for storage, and their space does not scale with the size of the input `nums`. No additional data structures that would grow with the size of the input are used, so the space usage remains constant.