2789. Largest Element in an Array after Merge Operations

Medium Greedy Array Prefix Sum

Leetcode Link

Problem Description

The problem presents us with an array of positive integers, and it allows us to perform a specific type of operation an unlimited number of times. In this operation, we can look for an element (`nums[i]`) such that it is less than or equal to the element directly after it (`nums[i + 1]`). When such an element is found, we sum up `nums[i]` and `nums[i + 1]` and replace `nums[i + 1]` with their sum, then remove `nums[i]` from the array. Our objective is to apply this operation as many times as required to maximize the value of the largest element in the array and return that maximum value.

The challenge is to carry out this operation strategically so that the largest possible value can be obtained, considering that the operation changes the array's length and its element values.

Intuition

To solve this problem, we need to identify a pattern or rule that helps us optimize the largest value. The key insight here is to recognize that combining a smaller number with a larger number will always result in a larger sum than combining two small numbers. Hence, to maximize the final value, we should target the smallest element that can be merged with a larger element next to it.

Starting from the right end of the array (the last element), we work our way backward. If we find that `nums[i]` is less than or equal to `nums[i + 1]`, we perform the operation specified, i.e., we sum those two elements and replace `nums[i + 1]` with the sum while effectively removing `nums[i]`. This is because any operation done to the left of `nums[i + 1]` will not affect the final value of `nums[i + 1]` once it's merged with `nums[i]`. Therefore, we can safely consolidate from right to left. Once we have gone through the array in reverse, we return the largest number.

This reverse traversal allows the largest number to "absorb" the sum of the smaller numbers before it, resulting in the maximal value we are looking for.

Solution Approach

The solution approach involves iterating through the array in reverse order and performing the specified operation to maximize the value of the elements. This is achieved by leveraging a simple algorithm without the need for additional data structures or complex patterns.

Here's a step-by-step walkthrough of the implementation:

1. A loop goes through the array elements starting from the second to last element towards the first (`nums[len(nums) - 2]` to `nums[0]`). We use a reverse loop by starting from `len(nums) - 2` and decrementing the index `i` until we reach `0`. This reverse order is crucial because it allows us to perform the accumulation from right to left, which follows the problem's requirement that `nums[i]` should be less than or equal to `nums[i + 1]`.

2. For every element in the loop, we check the condition `if nums[i] <= nums[i + 1]`. If true, this means we can perform the operation according to the problem statement.

3. When the condition is satisfied, we replace `nums[i + 1]` with the sum of `nums[i]` and `nums[i + 1]` using `nums[i] += nums[i + 1]`. The element `nums[i]` is not explicitly removed because it is not needed anymore, and the subsequent iterations will only work with the newly updated `nums[i + 1]`.

4. Once all possible operations have been performed (which is when the loop exits), we use the `max()` function to find the largest element in the array. The largest element is then returned as the result of the function `maxArrayValue`.

Code Implementation:

```
class Solution:
    def maxArrayValue(self, nums: List[int]) -> int:
        for i in range(len(nums) - 2, -1, -1):  # Start from the second-last element to the first
            if nums[i] <= nums[i + 1]:
                nums[i+1] += nums[i]            # Perform the accumulation operation
        return max(nums)                        # Find and return the largest element
```

This approach is efficient because it avoids unnecessary iterations and directly modifies the array in place, leading to the desired outcome with minimal computation.

Note: In the explanation, the removal operation is implicitly understood as it does not require an explicit method call for the logic to work but rather, we directly update the element that will eventually become the maximum.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using the following input array: `[1, 3, 1, 5]`.

1. According to our solution, we need to start iterating from the second to last element, which is `1`. The element after the second to last is `3`. Since `1 <= 5`, we perform the operation, thus, the array becomes `[1, 3, 6]` because we've added `1` and `5` and removed the second `1`.

2. Moving to the next element in reverse, which is `3` now. The element after `3` is `6`. Since `3 <= 6`, we perform the operation again, and the array is now `[1, 9]` because we've added `3` and `6` and removed `3`.

3. Now, we only have two elements left: `1` and `9`. The first element `1` is indeed less than or equal to the second element `9`, so we do the final operation, leaving us with `[10]`.

4. With no more elements left to process, we finish our loop. The resulting array has only one element, `10`, which is the maximal sum we could achieve through our operations.

5. The `maxArrayValue` function now returns `10` as the result because it is the largest (in this case, the only) element in the final array.

In this example, we maximized the value of the largest element in the array `[1, 3, 1, 5]` by following the solution approach and ended up with a maximum value of `10`.

Python Solution

```
from typing import List

class Solution:
    def maxArrayValue(self, nums: List[int]) -> int:
        # Loop through the array in reverse order except for the last element.
        for i in range(len(nums) - 2, -1, -1):
            # If the current element is less than or equal to the next one,
            # modify the current element by adding the next element's value to it.
            if nums[i] <= nums[i + 1]:
                nums[i] += nums[i + 1]

        # After modifying the array, return the maximum value in the array.
        return max(nums)
```

Java Solution

```
class Solution {
    public long maxArrayValue(int[] nums) {
        // Initialize the variables.
        // 'n' holds the length of the array 'nums'.
        int n = nums.length;

        // 'maxValue' will keep track of the maximum value found in the array.
        long maxValue = nums[n - 1];

        // 'tempValue' holds the value of the running sum or the current element being considered.
        long tempValue = nums[n - 1];

        // Iterate backwards through the array starting from the second-to-last element.
        for (int i = n - 2; i >= 0; --i) {
            // If the current element is less than or equal to tempValue,
            // it means we can add it to 'tempValue' to potentially form a larger sum.
            if (nums[i] <= tempValue) {
                tempValue += nums[i];
            } else {
                // If the current element is larger than 'tempValue',
                // we restart the running sum at the current element.
                tempValue = nums[i];
            }
            // Update the maximum value found with the larger of the current 'maxValue'
            // and the 'tempValue' obtained from this round of the iteration.
            maxValue = Math.max(maxValue, tempValue);
        }

        // Return the maximum value found.
        return maxValue;
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm> // For std::max

class Solution {
public:
    long long maxArrayValue(vector<int>& nums) {
        int size = nums.size(); // Use 'size' for the size of nums array
        // Initialize 'maxValue' with the last element of the array
        long long maxValue = nums[size - 1];
        // 'tempSum' holds the temporary cumulative sum from the end of the array
        long long tempSum = nums[size - 1];

        // Start iterating from the second last element to the first
        for (int i = size - 2; i >= 0; --i) {
            // If the current element is less than or equal to 'tempSum'
            if (nums[i] <= tempSum) {
                // Add it to 'tempSum'
                tempSum += nums[i];
            } else {
                // If the current element is larger, start a new 'tempSum'
                tempSum = nums[i];
            }
            // Update 'maxValue' if 'tempSum' is greater
            // than the current 'maxValue'
            maxValue = std::max(maxValue, tempSum);
        }
        // Return the maximum 'tempSum' found
        return maxValue;
    }
};
```

Typescript Solution

```
/**
 * Function to calculate the maximum value in an array after performing
 * specific in-place transformations.
 * For each element, starting from the second to last and moving backwards,
 * if the current element is less than or equal the next element,
 * it will be incremented by the value of the next element.
 * After the transformation, the function returns the maximum value in the array.
 * @param nums The array of numbers to be transformed.
 * @returns The maximum value after the transformation.
 */
function maxArrayValue(nums: number[]): number {
    // Iterate over the array from the second to last element to the beginning
    for (let i = nums.length - 2; i >= 0; --i) {
        // If the current element is less than or equal to the next one,
        // add the next element's value to the current one
        if (nums[i] <= nums[i + 1]) {
            nums[i] += nums[i + 1];
        }
    }

    // Return the maximum value in the modified array
    return Math.max(...nums);
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the `nums` list. This is because there is a single loop that iterates backwards through the list, doing a constant amount of work for each element by adding to the previous element if the condition is met.

The space complexity of the algorithm is `O(1)` as it operates in-place, modifying the input list directly. No additional data structures that grow with the input size are used.