2439. Minimize Maximum of Array

Medium Greedy Array Binary Search Dynamic Programming Prefix Sum Leetcode Link

Problem Description

In this problem, you are given an array `nums` with `n` non-negative integers. The focus is on the array's integers' transformation through a series of operations to minimize the value of the maximum integer in the array. Each operation includes three steps:

1. Select an integer `i` where `i` is between 1 and `n-1` (inclusive), and `nums[i]` is greater than 0.
2. Decrease `nums[i]` by 1.
3. Increase `nums[i - 1]` by 1.

The objective is to find out the smallest possible value of the largest number in the array after performing any number of these operations.

Intuition

The intuition behind the solution is to use binary search to find the minimum possible value of the maximum integer in the array after all the operations. Since we are asked for the minimum possible maximum value, we can guess that there is some maximum value that we cannot go below no matter how many operations we perform.

We perform binary search between the lowest possible maximum value, which is 0, and the current maximum value in the array. For each guess (possible maximum value) in the binary search, we check if we can achieve that maximum value after performing the operations. Here's the step-by-step reasoning:

1. Start with a binary search between 0 and the maximum value present in the array `nums`.
2. Calculate the middle value between the current range as our guess for the possible maximum value.
3. Define a `check` function that will determine if it is possible to achieve the guessed maximum value with the array.
4. In the `check` function, we work backwards from the end of the array towards the beginning (right to left). For each element, we calculate the surplus or deficit compared to our guessed maximum and carry this difference to the previous element.
5. The conditions that tell us if we can achieve the guessed maximum are:
   - If at any point the required decrease is so great we cannot make up the difference with the carried surplus, the check fails.
   - If we can carry the difference through to the first element, we then check if the total sum of increases needed for the first element does not exceed our guessed maximum.
6. If the `check` function returns `true`, it means we can achieve the guessed maximum or even lower, so we continue searching towards the lower half of the binary search range.
7. If the `check` function returns `false`, the guessed maximum is too low, and we have to adjust our range towards the higher values.
8. Repeat this process until the binary search narrows down to the smallest possible maximum integer value that satisfies the condition (the left and right pointers converge).

The solution uses the fact that after all possible optimizations, the array `nums` will become non-decreasing from the beginning to the end. This is because the maximum possible number in an optimal solution cannot be less than the average of the numbers in the array. Thus, binary search optimizes the process by eliminating the need to test each value sequentially.

Solution Approach

The solution utilizes a binary search algorithm to narrow down the possible maximum value that can be achieved in the array after performing the operations. Here's an in-depth walkthrough of the algorithm, elaborating on the reference solution provided:

1. **Binary Search Initialization:** Define two pointers, `left` and `right`, which represent the range within which the final answer could be. Initialize `left` to `0`, assuming the minimum possible value in the array could be 0 after all operations, and initialize `right` to the maximum value in `nums`, which is the upper limit of the answer.

2. **Binary Search Loop:** While `left` is less than `right`, calculate the middle of the current `left` and `right` boundary, `mid = (left + right) >> 1`. (`>> 1` is a bitwise operation equivalent to dividing by 2, shifting the bits to the right once.)

3. **Check Function:** The `check` function is defined to test whether a certain maximum value `mx` can be attained after performing the allowed operations. It works as follows:

   - Initialize a `d` variable to 0, which represents the cumulative difference needed to reach the guessed maximum `mx` from the end of the array.
   - Iterate in reverse (from the last element to the second, hence `nums[:0:-1]`) to calculate `d`. At each step, update `d` by adding the difference between the current element `x` and `mx` only if that difference is positive. In other words, if the element `x` is larger than `mx`, we need to 'push' some of its value to its left neighbor.
   - Finally, add `d` to the first element `nums[0]` and check if it still does not exceed `mx`. If it does not, return `True`; otherwise, `False`.
4. **Binary Search Logic:** Use the result of the `check` function inside the binary search loop. If `check(mid)` returns `True`, the current guess can be achieved, and perhaps a lower maximum value is possible, so we update `right` to `mid`. Conversely, if `check(mid)` returns `False`, we must increase our guess and move the lower boundary up, setting `left` to `mid + 1`.

5. **Binary Search Conclusion:** The binary search concludes when `left` equals `right`, indicating that the search range has been narrowed down to a single element—this element is the minimum possible maximal value that can be achieved in the `nums` array after performing any number of operations.

This solution effectively combines binary search with a greedy strategy in the `check` function to ensure that all potential operations are taken into account from right to left, simulating the effect that the operations would have on an array and determining the viability of a selected maximum value. The use of binary search drastically reduces the number of checks needed compared to a linear search and finds the optimal solution efficiently.

Example Walkthrough

Assume we have an array `nums` = [3, 1, 5, 6, 8, 7] with `n` = 6 non-negative integers. We need to minimize the value of the maximum integer in the array after performing our operations.

1. Initialize our binary search range with `left` = 0 and `right` = max(`nums`) = 8.

2. Conduct a binary search. The first `mid` value will be (0 + 8) / 2 = 4.

3. Run the check function with `mx` = 4:

   - Start iterating from the end:
     - `nums[5]` = 7, `d` += 7 - 4 = 3 (since 7 is greater than 4)
     - `nums[4]` = 8, `d` += 8 - 4 = 7 (total `d` becomes 10)
     - `nums[3]` = 6, `d` += 6 - 4 = 12 (total `d` becomes 12)
     - `nums[2]` = 5, `d` += 5 - 4 = 13 (total `d` becomes 13)
     - `nums[1]` does not contribute since 1 is not greater than 4

     Now, adding the total `d` (13) to `nums[0]` which is 3, we have 3 + 13 = 16, which is greater than `mx` = 4, so our check function returns `False`.
4. Since the check function returned `False`, we can't attain a maximum with 4. We adjust our binary search range, setting `left` to `mid + 1`, which is 5.

5. With a new search range `left` = 5 and `right` = 8, we calculate a new `mid` value of (5 + 8) / 2 = 6 and repeat the `check` function. We keep iterating this process until we find the `mid` value for which the `check` function returns `True`.

6. When we finally find the smallest `mid` for which the `check` function returns `True`, we have found the smallest possible value of the largest number in the array after performing the operations. For this example, let's assume through the binary search process, we find that the smallest maximum we can achieve is `5`.

7. The solution approach successfully minimizes the maximum value in the array with the optimal use of operations detailed in the given problem.

Python Solution

```python
class Solution:
    def minimizeArrayValue(self, nums: List[int]) -> int:

        # Function to check if a given maximum value can be achieved
        # by distributing the values across the nums array.
        def is_valid(max_value):
            # 'additional' keeps track of the value that needs to be distributed
            # to achieve the max_value in the nums array.
            additional = 0
            # Iterating backwards to check if it's possible to reduce
            # all values to at most max_value by distribution.
            for num in nums[:-1][::-1]:
                additional = max(0, additional + num - max_value)
            # Check if the first element plus the distributable amount is less than
            # or equal to the maximum value after distribution.
            return nums[0] + additional <= max_value

        # Binary search to find the minimum possible maximum value in the array
        # after redistribution.
        left = 0              # Minimum possible value
        right = max(nums)     # Maximum possible value in the original array

        # Perform binary search to find the minimum maximum value.
        while left < right:
            mid = (left + right) // 2
            # If the mid value is a valid maximum, we try to see if there's a smaller maximum.
            if is_valid(mid):
                right = mid
            # If the mid value is not valid, we search for a larger value that might be valid.
            else:
                left = mid + 1
        # The binary search loop exits when left == right, which is the minimum maximum value.
        return left

```

Java Solution

```java
class Solution {
    private int[] nums; // Array to store the input numbers

    // Method to find the minimum possible max value of the array after modifications
    public int minimizeArrayValue(int[] nums) {
        this.nums = nums; // Initialize the global array with the input array
        int left = 0; // Start of the search range
        int right = findMaxValue(nums); // End of the search range, which is the max value in nums

        // Binary search to find the minimum possible value
        while (left < right) {
            int mid = (left + right) / 2; // Middle value of the current search range
            if (canBeMinimizedTo(mid)) {
                right = mid; // If can minimize to `mid`, continue search on the left half
            } else {
                left = mid + 1; // Otherwise, search on the right half
            }
        }

        return left; // The minimum possible max value after minimization
    }

    // Method to check if we can minimize the array to a certain max value
    private boolean canBeMinimizedTo(int maxVal) {
        long deficit = 0; // Tracks the required decrease to achieve maxVal
        // Iterate backward through the array
        for (int i = nums.length - 1; i > 0; --i) {
            // Accumulate the deficit from the end of the array to the start
            deficit = Math.max(0, deficit + nums[i] - maxVal);
        }
        // Check if we can minimize the first element with the accumulated deficit
        return nums[0] + deficit <= maxVal;
    }

    // Helper method to find the maximum value in an array
    private int findMaxValue(int[] nums) {
        int maxValue = nums[0]; // Initialize with the first element of the array
        // Iterate over the array to find the maximal value
        for (int num : nums) {
            maxValue = Math.max(maxValue, num);
        }
        return maxValue; // Return the found maximum
    }
}

```

C++ Solution

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

class Solution {
public:
    int minimizeArrayValue(std::vector<int>& nums) {
        // Initialize the search range for the minimum possible max value
        int minPossibleValue = 0;
        int maxPossibleValue = *max_element(nums.begin(), nums.end());

        // Define a lambda function to check if a given max value is possible
        // by distributing values from right to left
        auto canDistribute = [&](int maxValue) {
            long excess = 0;
            // Iterate over the array from the end to start (excluding the first element)
            for (int i = nums.size() - 1; i > 0; --i) {
                // Calculate the excess that needs to be distributed to the left
                excess = std::max(0l, excess + nums[i] - maxValue);
            }
            // The first element plus any excess should not exceed the max value
            return nums[0] + excess <= maxValue;
        };

        // Perform a binary search to find the minimum max value that satisfies
        // the distribution criteria
        while (minPossibleValue < maxPossibleValue) {
            int mid = (minPossibleValue + maxPossibleValue) / 2;

            // If the current midway value works, try to see if we can lower it
            if (canDistribute(mid))
                maxPossibleValue = mid;
            // If it doesn't work, we need to try a higher value
            else
                minPossibleValue = mid + 1;
        }

        // The left pointer will point to the smallest max value that works, so return it
        return minPossibleValue;
    }
};

```

Typescript Solution

```typescript
// TypeScript code equivalent of the provided C++ code

// Import array related utilities
import { max } from 'lodash';

// Define a method to minimize the array value
function minimizeArrayValue(nums: number[]): number {
    // Initialize the search range for the minimum possible max value
    let minPossibleValue: number = 0;
    let maxPossibleValue: number = max(nums) as number;

    // Define a lambda function to check if a given max value is possible
    // by distributing values from right to left
    const canDistribute = (maxValue: number): boolean => {
        let excess: number = 0;
        // Iterate over the array from the end to start (excluding the first element)
        for (let i: number = nums.length - 1; i > 0; --i) {
            // Calculate the excess that needs to be distributed to the left
            // If positive, add it to the current excess, otherwise keep the current excess
            excess = Math.max(0, excess + nums[i] - maxValue);
        }
        // The first element plus any excess should not exceed the max value
        return nums[0] + excess <= maxValue;
    };

    // Perform a binary search to find the minimum max value that satisfies
    // the distribution criteria
    while (minPossibleValue < maxPossibleValue) {
        let mid: number = Math.floor((minPossibleValue + maxPossibleValue) / 2);

        // If the current midway value works, try to see if we can lower it
        if (canDistribute(mid)) {
            maxPossibleValue = mid;
        }
        // If it doesn't work, we need to try a higher value
        else {
            minPossibleValue = mid + 1;
        }
    }

    // The minPossibleValue will point to the smallest max value that works, so return it
    return minPossibleValue;
}

// Example usage:
// const nums: number[] = [10, 20, 30];
// const minimizedValue: number = minimizeArrayValue(nums);
// console.log(minimizedValue); // Output will vary based on function implementation

```

Time and Space Complexity

The given code implements a binary search algorithm to minimize the maximum value in the array after applying the specified operation. Here's the analysis of its time and space complexity:

Time Complexity:

The while loop in the code is running a binary search over the range `0` to `max(nums)`, which has at most `log(max(nums))` iterations since we're halving the search space in every step. The `check` function is called inside the loop and it iterates through the array in reverse, which takes `O(n)` time. Therefore, the time complexity of the code is `O(n * log(max(nums)))`.

Space Complexity:

The space complexity of the code is `O(1)`. The algorithm uses a constant number of variables (`d`, `mx`, `left`, `right`, `mid`) and the space occupied by these does not depend on the size of the input array `nums`. The function `check` also does not use any additional space that scales with the size of the input, as it works in place.