1802. Maximum Value at a Given Index in a Bounded Array

Medium | Greedy | Binary Search | Leetcode Link

Problem Description

In this problem, we are tasked with constructing an array `nums` with the following constraints:

- The length of the array `nums` is equal to the given integer `n`.
- Each element in the array is a positive integer.
- The absolute difference between any two consecutive elements is at most `1`.
- The sum of all elements in `nums` does not exceed a given integer `maxSum`.
- Among all possible `nums` arrays that satisfy the above conditions, we want to maximize the value of `nums[index]`.

Our objective is to find out what that maximized `nums[index]` is, given the parameters `n` (the length of the array), `index` (the specific position in the array we want to maximize), and `maxSum` (the maximum allowed sum of all elements in the array).

Intuition

The intuition behind the solution is to leverage binary search to efficiently find the maximum possible value of `nums[index]`.

1. We know that `nums[index]` must be a positive integer and that the sum of all elements in the array must not exceed `maxSum`. This means that `nums[index]` has an upper bound given by `maxSum`.
2. The idea is to perform a binary search, starting with the lowest possible value for `nums[index]` (which is 1) and the maximum possible value, which would be `maxSum` (assuming all other values in the array are 1).
3. For each possible value of `nums[index]` we test in our binary search, we calculate the sum of elements that would be required to form a valid array if `nums[index]` were that value. To do this, the `sum` function is used, which calculates the sum of elements in a portion of the array that slopes upwards or downwards by 1 with each step away from `nums[index]`.
4. If the calculated sum is less than or equal to `maxSum` while maintaining the constraints of the problem, it means we can potentially increase `nums[index]`. On the other hand, if the sum exceeds `maxSum`, then `nums[index]` must be lower.

By using this method, when we eventually narrow down to a single value through binary search, we find the maximum value of `nums[index]` that can exist within an array satisfying all the described constraints.

Solution Approach

The solution provided uses a binary search algorithm to find the maximum value of `nums[index]`. The binary search algorithm is a classic approach to efficiently search for an element in a sorted array by repeatedly dividing the search interval in half.

The following steps are taken in this implementation:

1. **Initialize Search Range**: The search for the maximum value of `nums[index]` begins by setting the `left` bound to `1`, which is the smallest possible value for any element in the array, and `right` bound to `maxSum`, the highest possible value for the `nums[index]` (assuming all other elements are at the minimum value of 1).
2. **Binary Search Loop**: A while loop runs as long as the `left` bound is less than `right`. The loop calculates the `mid` value as the average of `left` and `right`, setting up the next guess for `nums[index]`.
3. **Calculate Required Sum**: In each iteration, the program calculates the required sum for the array if `nums[index]` were equal to `mid`. This is done using a custom `sum` function, which accounts for the sum of the pyramid-like sequence that forms when values decrease by 1 on each side of the `index`.
   - **`sum(x, cnt)` Function**: This function calculates the sum of the first `cnt` terms of an arithmetic series that starts at `x` and decreases by 1 each term until it reaches 1 or runs out of terms. If `x` is greater than `cnt`, the sum is the sum of `cnt` terms starting at `x` and subtracting down to `(x - cnt + 1)`. If `x` is less than or equal to `cnt`, then the sum includes all numbers down to 1, and the remaining terms are 1s. The formula is based on the sum of the first `n` natural numbers `n(n + 1)/2` and adjusted for the start being `x` instead of `1`.

   The function calculates two sums:

   - The sum for the left side from `nums[index]` to the start of the array.
   - The sum for the right side from `nums[index]` to the end of the array.
4. **Update Search Bounds**: Depending on whether the sum of the sequence with `nums[index]` equal to `mid` exceeds `maxSum` or not, we adjust the binary search range accordingly:
   - If the total sum does not exceed `maxSum`, it is safe to move the `left` bound up to `mid` because a larger or equal `nums[index]` is viable.
   - If the total sum exceeds `maxSum`, the `right` bound is set to `mid - 1` because we need a smaller `nums[index]` to reduce the total sum.
5. **Determine the Maximum Value**: After exit from the loop, the maximum possible value for `nums[index]` is found, which is pointed by `left`. At this point, `left` is the largest value that did not violate the sum constraint. Since the constraints ensure that the sequence is increasing then decreasing around `nums[index]` and that the maximum sum does not exceed `maxSum`, `left` indeed maximizes `nums[index]`.

By the end of this process, the solution has efficiently zeroed in on the largest possible value for `nums[index]` in compliance with all the problem's constraints using binary search.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the mentioned constraints.

Suppose we have the following inputs:

- n = 5 (length of the array)
- index = 2 (position in the array we want to maximize)
- maxSum = 10 (maximum allowed sum of all elements in the array)

Step by Step Process:

1. **Initialize Search Range**:
   - We start with `left = 1` and `right = maxSum = 10`.
2. **Binary Search Loop**:
   - Begin with `left = 1` and `right = 10`.
   - Calculate `mid = (left + right) / 2`, let's assume integer division, so with `left = 1` and `right = 10`, `mid = 5`.
3. **Calculate Required Sum**:
   - We calculate the sum for `nums[index] = mid`.
   - Using the `sum` function, we calculate the sum for the left part (from start to index) and the right part (from index to end).
   - For the left part, we need `2` values (elements at index 0 and 1), and for the right part, we also need `2` values (elements at index 3 and 4).
   - `sumLeft = sum(mid, 3) = sum(5, 3) = 5 + 4 + 1 (as the third term would be 0) = 10`.
   - `sumRight = sum(mid, 2) = sum(5, 2) = 5 + 4 = 9`.
   - Total sum = `sumLeft + sumRight - mid` (we subtract `mid` because it's counted in both the left and right sums) = `10 + 9 - 5 = 14`.
   - This sum exceeds `maxSum`; therefore, we need to reduce `mid`.
4. **Update Search Bounds**:
   - Since `14` exceeds `maxSum`, we set `right = mid - 1 = 4`.
5. **Loop Continuation**:
   - Adjust `mid` with the new bounds, `left = 1` and `right = 4`.
   - New `mid = (1 + 4) / 2 = 2`.
   - Calculate new sums with `mid = 2`.
   - `sumLeft = sum(2, 3) = 2 + 1 + 1 = 4`.
   - `sumRight = sum(2, 2) = 2 + 1 = 3`.
   - Total sum = `sumLeft + sumRight - mid = 4 + 3 - 2 = 5`.
   - This sum fits within `maxSum`, so we try to increase `mid` by moving `left` up.
6. **Update Search Bounds Again**:
   - As `5` is less than `maxSum`, we now set `left = mid = 2`.
   - Now `left = 2` and `right = 4`.
7. **Finishing the Search**:
   - Continue the binary search until `left` and `right` meet.
   - Suppose in the next iteration `mid = 3` does not exceed `maxSum` but `mid = 4` does, we will stop with `left` at `3`.
8. **Determination**:
   - The binary search concludes when `left` equals `right`, which is the value just before the sum exceeded `maxSum`.
   - We find `left` to be `3`, so `nums[index] = 3` is the largest possible value that does not violate the constraints.

This example illustrates the solution approach, showing how a binary search systematically narrows down the maximum value for `nums[index]` while adhering to the problem's constraints. By calculating sums that would form a valid array configuration for each guess and adjusting our bounds accordingly, we efficiently pinpoint the solution.

Python Solution

```python
class Solution:
    def maxValue(self, n: int, index: int, maxSum: int) -> int:

        # Define a local function to calculate the sum of the
        # arithmetic series that starts at `start_value`, has `count` number of elements
        def calculate_sum(start_value, count):
            if start_value >= count:
                # If the start value is larger than or equal to count,
                # calculate the sum of the first `count` numbers in the arithmetic series
                # descending from `start_value`
                return (start_value + start_value - count + 1) * count // 2
            else:
                # If start_value is less than count, then the series is not long
                # enough to decrease down to 1. It bottoms out at 1 after `start_value` steps
                # Then we have to count the remaining `count - start_value` times 1.
                return (start_value + 1) * start_value // 2 + count - start_value

        left, right = 1, maxSum  # Set the search range between 1 and maxSum
        while left < right:      # Use binary search to find maximum value
            mid = (left + right + 1) >> 1  # Calcualte the middle point

            # Check if the sum of both sides with `mid` as the peak value is <= maxSum
            if calculate_sum(mid - 1, index) + calculate_sum(mid, n - index - 1) + mid <= maxSum:
                left = mid   # If it's less than or equal to maxSum, this is a new possible solution
            else:
                right = mid - 1  # If it exceeds maxSum, we discard the mid value and go lower

        return left  # At the end of the loop, `left` is our maximum value

# Example of how to use the class
# solution = Solution()
# result = solution.maxValue(10, 5, 54)
# print(result)  # The results would print the maximum value that can be achieved

```

Java Solution

```java
class Solution {

    // Method to find the maximum integer value that can be placed in position 'index'
    // of an array of length 'n' such that the total sum does not exceed 'maxSum'
    // and the array is a 0-indexed array with non-negative integers.
    public int maxValue(int n, int index, int maxSum) {
        // Define search boundaries for binary search
        int left = 1, right = maxSum;

        // Perform binary search
        while (left < right) {
            // Calculate midpoint and avoid integer overflow
            int mid = (left + right + 1) >>> 1;

            // If the calculated sum is within the allowed range, search in the upper half
            if (sum(mid - 1, index) + sum(mid, n - index - 1) <= maxSum) {
                left = mid;
            } else {
                // Otherwise, search in the lower half
                right = mid - 1;
            }
        }

        // At this point, 'left' is the maximum value that can be placed at 'index'
        return left;
    }

    // Helper method to calculate the sum of the values we could place in the array
    // if we start from 'x' and decrement by 1 until we reach 1, limited by 'count'
    private long sum(long x, int count) {
        // If 'x' is greater than 'count', we can simply calculate a triangular sum
        if (x >= count) {
            return (x + x - count + 1) * count / 2;
        } else {
            // Otherwise, we calculate the triangular sum up to 'x' and add the remaining
            // 'count - x' ones (since we cannot decrement below 1)
            return (x + 1) * x / 2 + count - x;
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Helper function to calculate sum in a range with certain conditions
    // If x is greater or equal to count, it calculates the sum of an arithmetic sequence,
    // Otherwise, it calculates the partial sum and adds the remaining terms
    long calculateSum(long x, int count) {
        if (x >= count) {
            // Full arithmetic sequence
            return (x + x - count + 1) * count / 2;
        } else {
            // Partial arithmetic sequence + remaining elements
            return (x + 1) * x / 2 + count - x;
        }
    }

    // Main function to find the maximum value that can be inserted at a given index
    int maxValue(int n, int index, int maxSum) {
        int minValue = 1, maxValue = maxSum; // set the bounds for binary search

        // Binary search to find the max value possible to achieve sum up to maxSum
        while (minValue < maxValue) {
            int midValue = (minValue + maxValue + 1) >> 1;

            // Check if the sum of values on both sides fits within maxSum
            if (calculateSum(midValue - 1, index) + calculateSum(midValue, n - index - 1) <= maxSum) {
                minValue = midValue; // Solution exists, go right
            } else {
                maxValue = midValue - 1; // Solution doesn't fit, go left
            }
        }

        // minValue holds the maximum value possible for the array
        return minValue;
    }
};

```

Typescript Solution

```typescript
// Helper function to calculate sum in a range with certain conditions
// If x is greater or equal to count, it calculates the sum of an arithmetic sequence,
// Otherwise, it calculates the partial sum and adds the remaining terms
function calculateSum(x: number, count: number): number {
    if (x >= count) {
        // Full arithmetic sequence
        return (x + x - count + 1) * count / 2;
    } else {
        // Partial arithmetic sequence + remaining elements
        return (x + 1) * x / 2 + count - x;
    }
}

// Main function to find the maximum value that can be inserted at a given index to not exceed maxSum
function maxValue(n: number, index: number, maxSum: number): number {
    let minValue = 1;
    let maxValue = maxSum; // set the bounds for binary search

    // Binary search to find the max value possible to achieve sum up to maxSum
    while (minValue < maxValue) {
        const midValue = Math.floor((minValue + maxValue + 1) / 2);

        // Check if the sum of values on both sides fits within maxSum
        if (calculateSum(midValue - 1, index) + calculateSum(midValue, n - index - 1) <= maxSum - midValue) {
            minValue = midValue; // Solution exists, go right
        } else {
            maxValue = midValue - 1; // Solution doesn't fit, go left
        }
    }

    // minValue holds the maximum value possible for the array
    return minValue;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(log(maxSum))`. The binary search algorithm runs between 1 and `maxSum`, which determines the number of iterations needed to find the solution. In each iteration, the `sum` function is called twice, each call of which is `O(1)` because the operations involve simple arithmetic and a conditional check, and thus don't depend on the size of `n` or `maxSum`.

The space complexity of the code is `O(1)`. There are only a fixed number of variables used (`left`, `right`, `mid`, and within the `sum` function), and no extra space that scales with the input size is required. Therefore, the amount of memory used is constant.