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

Problem Description

Greedy Binary Search

Medium

• The length of the array nums is equal to the given integer n.

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

- 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

nums [index] that can exist within an array satisfying all the described constraints.

nums [index] (assuming all other elements are at the minimum value of 1).

• The sum for the right side from nums[index] to the end of the array.

left indeed maximizes nums[index].

all the problem's constraints using binary search.

○ We start with left = 1 and right = maxSum = 10.

 \circ sumRight = sum(mid, 2) = sum(5, 2) = 5 + 4 = 9.

Adjust mid with the new bounds, left = 1 and right = 4.

This sum fits within maxSum, so we try to increase mid by moving left up.

• Begin with left = 1 and right = 10.

Suppose we have the following inputs:

specific position in the array we want to maximize), and maxSum (the maximum allowed sum of all elements in the array).

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

Intuition

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).

elements in a portion of the array that slopes upwards or downwards by 1 with each step away from nums[index].

- 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
- 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

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: 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

Binary Search Loop: A while loop runs as long as the left bound is less than right. The loop calculates the mid value as

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

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

Update Search Bounds: Depending on whether the sum of the sequence with nums[index] equal to mid exceeds maxSum or

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,

By the end of this process, the solution has efficiently zeroed in on the largest possible value for nums[index] in compliance with

the average of left and right, setting up the next guess for nums [index].

values decrease by 1 on each side of the index. o 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

- 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.
- 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.

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

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).

○ Total sum = sumLeft + sumRight - mid (we subtract mid because it's counted in both the left and right sums) = 10 + 9 - 5 = 14.

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).

Step by Step Process:

Example Walkthrough

• n = 5 (length of the array)

Initialize Search Range:

Calculate Required Sum:

Binary Search Loop:

• This sum exceeds maxSum; therefore, we need to reduce mid. **Update Search Bounds:**

• sumLeft = sum(mid, 3) = sum(5, 3) = 5 + 4 + 1 (as the third term would be 0) = 10.

Calculate mid = (left + right) / 2, let's assume integer division, so with left = 1 and right = 10, mid = 5.

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

∘ Since 14 exceeds maxSum, we set right = mid - 1 = 4. **Loop Continuation:**

 \circ sumLeft = sum(2, 3) = 2 + 1 + 1 = 4.

 \circ New mid = (1 + 4) / 2 = 2.

Calculate new sums with mid = 2.

Update Search Bounds Again:

∘ Now left = 2 and right = 4.

Finishing the Search:

Solution Implementation

else:

solution = Solution()

Java

Example of how to use the class

result = solution.maxValue(10, 5, 54)

public int maxValue(int n, int index, int maxSum) {

int mid = (left + right + 1) >>> 1;

int left = 1, right = maxSum;

left = mid;

right = mid - 1;

private long sum(long x, int count) {

// Perform binary search

while (left < right) {</pre>

} else {

return left;

if (x >= count) {

} else {

// Define search boundaries for binary search

// Calculate midpoint and avoid integer overflow

// Otherwise, search in the lower half

return (x + x - count + 1) * count / 2;

return (x + 1) * x / 2 + count - x;

if $(sum(mid - 1, index) + sum(mid, n - index - 1) <= maxSum) {$

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

// 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'

// 'count - x' ones (since we cannot decrement below 1)

Python

class Solution:

Determination:

 \circ sumRight = sum(2, 2) = 2 + 1 = 3. \circ Total sum = sumLeft + sumRight - mid = 4 + 3 - 2 = 5.

• The binary search concludes when left equals right, which is the value just before the sum exceeded maxSum.

This example illustrates the solution approach, showing how a binary search systematically narrows down the maximum value for

• We find left to be 3, so nums[index] = 3 is the largest possible value that does not violate the constraints.

 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.

○ As 5 is less than maxSum, we now set left = mid = 2.

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.

def maxValue(self, n: int, index: int, maxSum: int) -> int:

descending from `start value`

def calculate sum(start value, count):

if start value >= count:

Define a local function to calculate the sum of the

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

if calculate sum(mid -1, index) + calculate sum(mid, n - index -1) + mid <= maxSum:

right = mid - 1 # If it exceeds maxSum, we discard the mid value and go lower

left = mid # If it's less than or equal to maxSum, this is a new possible solution

calculate the sum of the first `count` numbers in the arithmetic series

arithmetic series that starts at `start_value`, has `count` number of elements

Check if the sum of both sides with `mid` as the peak value is <= maxSum

// If the calculated sum is within the allowed range, search in the upper half

// Otherwise, we calculate the triangular sum up to 'x' and add the remaining

// Main function to find the maximum value that can be inserted at a given index to not exceed maxSum

if (calculateSum(midValue -1, index) + calculateSum(midValue, n - index - 1) <= maxSum - midValue) {

function maxValue(n: number, index: number, maxSum: number): number {

// Binary search to find the max value possible to achieve sum up to maxSum

const midValue = Math.floor((minValue + maxValue + 1) / 2);

minValue = midValue; // Solution exists, go right

// Check if the sum of values on both sides fits within maxSum

maxValue = midValue - 1; // Solution doesn't fit, go left

arithmetic series that starts at `start_value`, has `count` number of elements

calculate the sum of the first `count` numbers in the arithmetic series

enough to decrease down to 1. It bottoms out at 1 after `start value` steps

if calculate sum(mid -1, index) + calculate sum(mid, n - index - 1) + mid <= maxSum:

right = mid - 1 # If it exceeds maxSum, we discard the mid value and go lower

left = mid # If it's less than or equal to maxSum, this is a new possible solution

If the start value is larger than or equal to count.

left, right = 1, maxSum # Set the search range between 1 and maxSum

while left < right: # Use binary search to find maximum value</pre>

mid = (left + right + 1) >> 1 # Calcualte the middle point

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

print(result) # The results would print the maximum value that can be achieved

return (start_value + start_value - count + 1) * count // 2

If start value is less than count, then the series is not long

return (start_value + 1) * start_value // 2 + count - start_value

Check if the sum of both sides with `mid` as the peak value is <= maxSum

Then we have to count the remaining `count - start value` times 1.

let maxValue = maxSum; // set the bounds for binary search

// minValue holds the maximum value possible for the array

def maxValue(self, n: int, index: int, maxSum: int) -> int:

descending from `start value`

def calculate sum(start value, count):

if start value >= count:

Define a local function to calculate the sum of the

let minValue = 1:

} else {

return minValue;

else:

else:

solution = Solution()

Example of how to use the class

result = solution.maxValue(10, 5, 54)

class Solution:

while (minValue < maxValue) {</pre>

If the start value is larger than or equal to count.

left, right = 1, maxSum # Set the search range between 1 and maxSum

while left < right: # Use binary search to find maximum value</pre>

mid = (left + right + 1) >> 1 # Calcualte the middle point

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

print(result) # The results would print the maximum value that can be achieved

return (start_value + start_value - count + 1) * count // 2

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.

```
// If 'x' is greater than 'count', we can simply calculate a triangular sum
```

C++

```
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) {</pre>
            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) {</pre>
                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
// 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;
```

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

which is 0(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 0(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.

determines the number of iterations needed to find the solution. In each iteration, the sum function is called twice, each call of