2865. Beautiful Towers I

Array

Monotonic Stack

Problem Description

Medium Stack

You are given an array maxHeights which consists of n integers. Imagine the responsibility of constructing n towers along a straight line where the i-th tower is positioned at coordinate i and may vary in height, up to a maximum height specified by maxHeights[i]. The goal is to build these towers in such a way that they form a beautiful configuration. A beautiful configuration is defined by two criteria:

The height sequence of the towers (heights) must form a mountain array, meaning there exists an index i in heights such

The height of each tower (heights[i]) must be at least 1 and at most maxHeights[i].

For any index k greater than i, heights[k] is greater than or equal to heights[k+1] (descending order).

- that: For any index j less than i, heights[j] is less than or equal to heights[j+1] (ascending order).
- The challenge is to find the configuration of towers that is beautiful and yields the maximal possible sum of heights across all
- towers. Intuition

The approach to solve this problem involves iterating through each possible peak of the mountain (the highest point where the

value given by maxHeights[i]. We then expand this peak to the left and right to construct the ascending and descending parts of the mountain. While expanding to the left, we ensure that the heights of the towers are both less than or equal to maxHeights[j] and the height of the previous tower to maintain a non-decreasing slope. Similarly, when expanding to the right, we maintain heights that are less than or equal to maxHeights[j] and the previous tower to ensure a non-increasing slope. The total sum for this configuration

towers will transition from ascending to descending heights). For each potential peak position i, we determine the maximum sum

of heights for a mountain configuration where the peak is at i. To achieve this, we initialize the peak tower's height with 'x' - the

is computed by accumulating the heights as we expand from the peak to both the left and right ends. After exploring all potential peak positions, we keep track of the maximum sum of heights that we have encountered. This value represents the highest sum of tower heights for any beautiful configuration possible, and thus, is the answer we return.

Solution Approach The provided solution uses a straightforward brute force approach to simulate the construction of the mountain from each potential peak position. The approach does not use complex data structures or advanced algorithms; it relies on basic iteration

and condition checking. Here's a step-by-step breakdown of how the solution is implemented:

with non-increasing tower heights. Each y value is added to the total sum t.

configuration, and variable n to store the number of towers (the length of the maxHeights array).

The function maximumSumOfHeights begins with initializing variable ans to store the maximum sum found for any beautiful

returned.

directly to find the optimal configuration.

by trying each position for the peak.

The sum for this peak would be 1.

It then enters a loop to consider each element x in maxHeights as the peak of the mountain (tallest tower). The variable i refers to the index where the peak is located. Inside the loop, two additional loops are used:

total sum starting with the peak. For every element left of the peak (j index), we update y to the minimum of the current y and

sum t. • The second inner loop increases from the peak's index i+1 to the end of the array, similarly initializing y to the height of the peak. As we iterate to the right, we update y with the minimum of the current y and maxHeights[j] to ensure the mountain array condition is maintained

∘ The first inner loop decreases from the peak's index i-1 to the start of the array. We initialize y as the height of the peak tower and t as the

maxHeights[j] to ensure that the tower heights are non-decreasing as we move towards the peak. We then add the height y to the total

After both loops, we have constructed the tallest possible mountain whose peak is at index i and we compare its total sum t

- with the current maximum ans, updating ans if t is greater. Finally, after all iterations, ans contains the maximum sum that can be achieved by a beautiful configuration of towers and is
- previous tower (to maintain a valid mountain shape), and the cumulative sum of the current configuration, respectively. No extra data structures are used; the solution operates directly on the input array, and no sorting or other modifications are

needed. The simplicity of the problem allows for a direct brute force method, exploring every possibility and comparing sums

Throughout the implementation, variables ans, y, and t are used to keep track of the current best answer, the height of the

Example Walkthrough

maxHeights = [2, 1, 4, 3]In this scenario, we have four positions to place towers, and we can set their heights with the constraints provided in the maxHeights array. We need to build the towers such that they form a mountain array. Let's examine the construction step-by-step

To the left of the peak, there are no taller towers, so we can't extend in that direction. • To the right, we need to construct descending towers. However, given the height 1 at the next position, we cannot make a descending

sequence from 2. • The sequence fails to form a mountain, and the sum here would simply be 2.

 \circ The mountain array [2, 1, 4, 3] forms a valid, beautiful configuration with a sum of 2 + 1 + 4 + 3 = 10.

• Ascending from the left, we start with 2, can increase to 1, but then we cannot go higher because the peak is at 3.

Let's walk through a small example to illustrate the solution approach using the following maxHeights array:

Trying the third position as the peak (i = 2). The peak height x = 4. • Ascending from the left, we can have the first tower at height 2, next cannot exceed 1, so it stays 1. • Descending from the peak to the right, the next and final tower can be up to 3.

Out of all the trials, the third position peak yields the highest sum of tower heights, which is 10. Therefore, the solution would

The simple brute force approach worked effectively for this example by checking each possible peak and ensuring that all criteria

of a mountain array are met. By calculating the sums for each valid configuration and keeping track of the maximum, we identified

 No towers to the right since this is the end of the array. • The sequence [2, 1, 3] is not a valid mountain as it lacks a descending part, making this an invalid peak position.

def maximumSumOfHeights(self, maxHeights: List[int]) -> int:

left_min_height = current_height

right_min_height = current_height

temp_sum = current_height

Initialize the variable to hold the maximum sum of heights

encountered heights so far to the temporary sum

encountered heights so far to the temporary sum

for right_index in range(i + 1, num_heights):

temp_sum += right_min_height

// Function to calculate the maximum sum of heights.

for (int i = 0; i < sequenceLength; ++i) {</pre>

minHeight = maxHeightSequence[i];

maxSum = max(maxSum, tempSum);

// Return the maximum sum found

return maxSum;

long long maximumSumOfHeights(vector<int>& maxHeightSequence) {

long long maxSum = 0; // Initialize variable to hold the maximum sum

// Loop through the entire sequence to find the maximum sum of heights

// Extend to the left of position 'i' and accumulate heights

// Extend to the right of position 'i' and accumulate heights

// Update maxSum if the current temporary sum is greater

for (int leftIndex = i - 1; leftIndex >= 0; --leftIndex) {

// Reset minHeight for checking to the right of 'i'

int sequenceLength = maxHeightSequence.size(); // Get the length of the sequence

long long tempSum = maxHeightSequence[i]; // Start with the current height

tempSum += minHeight; // Add the minimum height to the temporary sum

for (int rightIndex = i + 1; rightIndex < sequenceLength; ++rightIndex) {</pre>

tempSum += minHeight; // Add the minimum height to the temporary sum

int minHeight = maxHeightSequence[i]; // Initialize the minimum height seen so far

minHeight = min(minHeight, maxHeightSequence[leftIndex]); // Update the min height

minHeight = min(minHeight, maxHeightSequence[rightIndex]); // Update the min height

Trying the fourth position as the peak (i = 3). The peak height x = 3.

Trying the first position as the peak (i = 0). The peak height x = 2.

Trying the second position as the peak (i = 1). The peak height x = 1.

Since the peak is the smallest possible tower, we cannot create a proper mountain.

- return 10 as the maximum sum for creating a beautiful tower configuration.
- the optimal construction. Solution Implementation

Initialize temporary variables for moving left and right from the current index

Temporary sum for current position including the current height itself

Move to the left of the current position and add the minimum of all

Move to the right of the current position and add the minimum of all

right_min_height = min(right_min_height, maxHeights[right_index])

max_sum = 0 # Calculate the length of the maxHeights list once as it is used multiple times num_heights = len(maxHeights) # Iterate through each height in the maxHeights list for i, current_height in enumerate(maxHeights):

for left_index in range(i - 1, -1, -1): left_min_height = min(left_min_height, maxHeights[left_index]) temp sum += left_min_height

Python

class Solution:

```
# Update the maximum sum if the temporary sum is greater than the current maximum
            max_sum = max(max_sum, temp_sum)
       # Return the maximum sum after considering all positions
       return max_sum
Java
class Solution {
   // Method to calculate the maximum sum of heights, considering that each position
   // can act as a pivot, and the sum includes the minimum height from the pivot to each side.
    public long maximumSumOfHeights(List<Integer> maxHeightList) {
        long maxSum = 0; // This will store the maximum sum of heights we find
        int listSize = maxHeightList.size(); // The size of the provided list
       // Iterate through each element in the list to consider it as a potential pivot
        for (int i = 0; i < listSize; ++i) {</pre>
            int currentHeight = maxHeightList.get(i); // Height at the current pivot
            long tempSum = currentHeight; // Initialize temp sum with current pivot's height
           // Calculate the sum of heights to the left of the pivot
            for (int j = i - 1; j >= 0; --j) {
                currentHeight = Math.min(currentHeight, maxHeightList.get(j)); // Update to the smaller height
                tempSum += currentHeight; // Add this height to the running total for the pivot
            currentHeight = maxHeightList.get(i); // Reset height for current pivot
            // Calculate the sum of heights to the right of the pivot
            for (int j = i + 1; j < listSize; ++j) {</pre>
                currentHeight = Math.min(currentHeight, maxHeightList.get(j)); // Update to the smaller height
                tempSum += currentHeight; // Add this height to the running total for the pivot
            // Update maxSum if the sum for the current pivot is greater than the previous maximum
           maxSum = Math.max(maxSum, tempSum);
       // Return the maximum sum of heights found
       return maxSum;
```

```
TypeScript
```

};

C++

public:

class Solution {

```
// This function calculates the maximum sum of heights based on the rules provided in the maxHeights array.
  function maximumSumOfHeights(maxHeights: number[]): number {
      // Initialize the answer variable that holds the maximum sum of heights encountered.
      let maximumSum = 0;
      // Get the total number of elements in the maxHeights array.
      const arrayLength = maxHeights.length;
      // Iterate over the maxHeights array.
      for (let currentIndex = 0; currentIndex < arrayLength; ++currentIndex) {</pre>
          // Get the height at the current index.
          const currentHeight = maxHeights[currentIndex];
          // Initialize the minimum height variables for left and right directions.
          let minHeightToLeft = currentHeight;
          let totalSum = currentHeight;
          // Iterate over the previous elements from the current index to calculate the sum.
          for (let leftIndex = currentIndex - 1; leftIndex >= 0; --leftIndex) {
              // Identify the new minimum height to the left.
              minHeightToLeft = Math.min(minHeightToLeft, maxHeights[leftIndex]);
              // Add the minimum height to the left to the total sum.
              totalSum += minHeightToLeft;
          // Reinitialize the minimum height to the current height for the right direction.
          let minHeightToRight = currentHeight;
          // Iterate over the next elements from the current index to calculate the sum.
          for (let rightIndex = currentIndex + 1; rightIndex < arrayLength; ++rightIndex) {</pre>
              // Identify the new minimum height to the right.
              minHeightToRight = Math.min(minHeightToRight, maxHeights[rightIndex]);
              // Add the minimum height to the right to the total sum.
              totalSum += minHeightToRight;
          // Compare the current total sum with the previous maximum sum and update maximumSum if necessary.
          maximumSum = Math.max(maximumSum, totalSum);
      // Return the maximum sum of heights after considering all possible positions.
      return maximumSum;
class Solution:
   def maximumSumOfHeights(self, maxHeights: List[int]) -> int:
       # Initialize the variable to hold the maximum sum of heights
       max_sum = 0
       # Calculate the length of the maxHeights list once as it is used multiple times
       num_heights = len(maxHeights)
       # Iterate through each height in the maxHeights list
        for i, current_height in enumerate(maxHeights):
            # Initialize temporary variables for moving left and right from the current index
            left_min_height = current_height
            right_min_height = current_height
            # Temporary sum for current position including the current height itself
            temp_sum = current_height
            # Move to the left of the current position and add the minimum of all
            # encountered heights so far to the temporary sum
            for left_index in range(i - 1, -1, -1):
                left_min_height = min(left_min_height, maxHeights[left_index])
                temp sum += left min height
            # Move to the right of the current position and add the minimum of all
            # encountered heights so far to the temporary sum
            for right_index in range(i + 1, num_heights):
                right_min_height = min(right_min_height, maxHeights[right_index])
                temp_sum += right_min_height
            # Update the maximum sum if the temporary sum is greater than the current maximum
```

return max_sum

Time Complexity

max_sum = max(max_sum, temp_sum)

not depend on the input size and remains constant.

Return the maximum sum after considering all positions

The time complexity of the provided code is $0(n^2)$, where n is the length of the maxHeights list. This is because for each element x in maxHeights, the code iterates over the elements to its left and then to its right in separate for-loops. Each of these nested

Time and Space Complexity

loops runs at most n-1 times in the worst case, leading to roughly 2 * (n-1) operations for each of the n elements, thus the quadratic time complexity. **Space Complexity** The space complexity of the provided code is 0(1) as it only uses a constant amount of extra space. Variables ans, n, i, x, y, t,

and j are used for computations, but no additional space that scales with the input size is used. Therefore, the space used does