<https://leetcode.com/problems/beautiful-towers-i/description/>

You are given an array heights of n integers representing the number of bricks in n consecutive towers. Your task is to remove some bricks to form a **mountain-shaped** tower arrangement. In this arrangement, the tower heights are non-decreasing, reaching a maximum peak value with one or multiple consecutive towers and then non-increasing.

Return the **maximum possible sum** of heights of a mountain-shaped tower arrangement.

**Example 1:**

Input: heights = [5,3,4,1,1]

Output: 13

**Explanation:**

We remove some bricks to make heights = [5,3,3,1,1], the peak is at index 0.

**Example 2:**

Input: heights = [6,5,3,9,2,7]

Output: 22

**Explanation**:

We remove some bricks to make heights = [3,3,3,9,2,2], the peak is at index 3.

**Example 3:**

Input: heights = [3,2,5,5,2,3]

Output: 18

**Explanation:**

We remove some bricks to make heights = [2,2,5,5,2,2], the peak is at index 2 or 3.

**Constraints:**

1 <= n == heights.length <= 10^3

1 <= heights[i] <= 10^9

**Attempt 1: 2024-08-31**

**Solution 1: Brute Force (60 min)**

class Solution {

    public long maximumSumOfHeights(int[] heights) {

        long result = 0;

        int n = heights.length;

        for(int i = 0; i < n; i++) {

            long sum = heights[i];

            long lastHeight = heights[i];

            // Calculate sum for left branch

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

                lastHeight = Math.min(lastHeight, heights[j]);

                sum += lastHeight;

            }

            lastHeight = heights[i];

            // Calculate sum for right branch

            for(int j = i + 1; j < n; j++) {

                lastHeight = Math.min(lastHeight, heights[j]);

                sum += lastHeight;

            }

            result = Math.max(result, sum);

        }

        return result;

    }

}

Time Complexity: O(N^2)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/beautiful-towers-i/solutions/4085159/brute-force-approach-very-simple-and-easy-to-understand-solution/>

Enumerate A[i] to be the highest based on its neighbour height, if travese from left to right, A[i - 1] height should not exceed A[i] (can equal), A[i + 1] height should not exceed A[i] (can equal)

From A[i] to A[0], the height for each pile is non-increasing.

From A[i] to A[n - 1], the height for each pile is non-increasing.

Sum up the heights for A[i] to be the highest, and return the biggest heights sum.

public long maximumSumOfHeights(List<Integer> maxHeights) {

long ans = 0;

for (int i = 0; i < maxHeights.size(); ++i) {

long sum = maxHeights.get(i), last = maxHeights.get(i);

// Calculate sum of heights to the left

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

last = Math.min(maxHeights.get(j), last);

sum += last;

}

last = maxHeights.get(i);

// Calculate sum of heights to the right

for (int j = i + 1; j < maxHeights.size(); ++j) {

last = Math.min(maxHeights.get(j), last);

sum += last;

}

ans = Math.max(ans, sum);

}

return ans;

}

Time O(n^2), Space O(1).

**Solution 2: Monotonic Increasing Stack (60 min)**

**We can test this solution for easier understand as heights = {2, 5, 3, 6, 2}**

class Solution {

    public long maximumSumOfHeights(int[] heights) {

        // Monotonic increasing stack to store index

        Stack<Integer> stack = new Stack<>();

        int n = heights.length;

        // Store maximum sum of height at each index

        long[] sum = new long[n];

        long result = 0;

        long curSum = 0;

        // Scan from left to right, logic similar to L84.Largest Rectangle in Histogram

        // Special handling when stack is empty, it means no previous index stored for

        // monotonic increasing heights in stack, the rectangle width is 'curIndex - (-1)'

        for(int i = 0; i < n; i++) {

            while(!stack.isEmpty() && heights[stack.peek()] > heights[i]) {

                int j = stack.pop();

                curSum -= ((long) heights[j] \* (j - (stack.isEmpty() ? -1 : stack.peek())));

            }

            curSum += ((long) heights[i] \* (i - (stack.isEmpty() ? -1 : stack.peek())));

            stack.push(i);

            sum[i] = curSum;

        }

        stack.clear();

        curSum = 0;

        // Scan from right to left, logic similar to L84.Largest Rectangle in Histogram

        // Special handling when stack is empty, it means no previous index stored for

        // monotonic increasing heights in stack, the rectangle width is 'n - curIndex'

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

            while(!stack.isEmpty() && heights[stack.peek()] > heights[i]) {

                int j = stack.pop();

                curSum -= ((long) heights[j] \* ((stack.isEmpty() ? n : stack.peek()) - j));

            }

            curSum += ((long) heights[i] \* ((stack.isEmpty() ? n : stack.peek()) - i));

            stack.push(i);

            // For each element, the total sum is the combination of left sum and right sum,

            // and subtracting heights[i] (since it's counted twice) through both left to

            // right and right to left traversal

            result = Math.max(result, sum[i] + curSum - heights[i]);

        }

        return result;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/beautiful-towers-i/solutions/4082905/brute-force-solution/>

**Note: This solution a bit different since it introduce preposing -1 for left to right traversal and n for right to left traversal.**

stack will saves the indices for an increasing sequence.

For example,

[1,2,2,3,3,3,6,6,6,6]

stack will be [-1, 0, 2, 5, 9].

Firstly, we iterate A[i] from A[0] to A[n - 1],

and assume A[i] is the highest.

We need to pop all higher element from stack,

and update the sum of all heights cur,

and push i into the stack.

We do the above process again from A[n - 1] to A[0],

then sum up the left part and right part,

we can have the result with A[i] as the highest.

Finally return the biggest heights sum.

public long maximumSumOfHeights(List<Integer> A) {

int n = A.size();

long[] left = new long[n];

Stack<Integer> stack = new Stack<>();

stack.push(-1);

long res = 0, cur = 0;

for (int i = 0; i < n; i++) {

while (stack.size() > 1 && A.get(stack.peek()) > A.get(i)) {

int j = stack.pop();

cur -= 1L \* (j - stack.peek()) \* A.get(j);

}

cur += 1L \* (i - stack.peek()) \* A.get(i);

stack.push(i);

left[i] = cur;

}

stack.clear();

stack.push(n);

cur = 0;

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

while (stack.size() > 1 && A.get(stack.peek()) > A.get(i)) {

int j = stack.pop();

cur -= 1L \* -(j - stack.peek()) \* A.get(j);

}

cur += 1L \* -(i - stack.peek()) \* A.get(i);

stack.push(i);

res = Math.max(res, left[i] + cur - A.get(i));

}

return res;

}

**Refer to chatGPT**

A more efficient solution using a monotonic stack can be derived to solve the problem in **O(n)** time complexity. The monotonic stack approach helps us efficiently compute the maximum possible sum of heights for a mountain-shaped tower arrangement by keeping track of valid height transitions.

**Problem Restatement**

Given an array of tower heights, the goal is to form a mountain shape where:

Heights increase to a peak and then decrease.

We want to maximize the sum of the heights in this arrangement.

**Approach Using Monotonic Stack**

To achieve this efficiently, we can use a **monotonic stack** to calculate the maximum possible sum of heights for any mountain peak:

Left Contribution: Use a monotonic stack to find the left boundary for each element, ensuring the sequence is non-decreasing towards the peak.

Right Contribution: Use a monotonic stack to find the right boundary for each element, ensuring the sequence is non-increasing after the peak.

**Monotonic Stack-Based Solution**

This solution uses two monotonic stacks: one to process the left side of each element and another for the right side. The idea is to compute the contribution from both sides of each potential peak efficiently.

class Solution {

public long maximumSumOfHeights(int[] heights) {

int n = heights.length;

long[] leftSum = new long[n];

long[] rightSum = new long[n];

long[] left = new long[n];

long[] right = new long[n];

// Monotonic stack for left side calculation

java.util.Stack<Integer> stack = new java.util.Stack<>();

for (int i = 0; i < n; i++) {

while (!stack.isEmpty() && heights[stack.peek()] > heights[i]) {

stack.pop();

}

if (!stack.isEmpty()) {

left[i] = left[stack.peek()];

}

leftSum[i] = left[i] + (i - (stack.isEmpty() ? -1 : stack.peek())) \* (long) heights[i];

left[i] = leftSum[i];

stack.push(i);

}

// Monotonic stack for right side calculation

stack.clear();

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

while (!stack.isEmpty() && heights[stack.peek()] > heights[i]) {

stack.pop();

}

if (!stack.isEmpty()) {

right[i] = right[stack.peek()];

}

rightSum[i] = right[i] + ((stack.isEmpty() ? n : stack.peek()) - i) \* (long) heights[i];

right[i] = rightSum[i];

stack.push(i);

}

// Calculate the maximum sum

long result = 0;

for (int i = 0; i < n; i++) {

result = Math.max(result, leftSum[i] + rightSum[i] - heights[i]);

}

return result;

}

}

**Explanation**

Left Sum Calculation:

We use a monotonic stack to track the nearest smaller height on the left. If the stack is empty, the current element is considered the leftmost.

For each element, leftSum[i] is calculated as the sum of heights up to i in a non-decreasing order from the left.

Right Sum Calculation:

A similar approach is used with a stack to calculate rightSum[i] as the sum of heights from i to the right in a non-increasing order.

Combining Results:

For each element, the total sum is the combination of leftSum[i] and rightSum[i], subtracting heights[i] (since it's counted twice).

**Complexity Analysis**

Time Complexity: O(n). Each element is pushed and popped from the stack at most once.

Space Complexity: O(n). The space is used for the stacks and the additional arrays to keep track of left and right sums.

This solution efficiently calculates the maximum sum of a mountain-shaped tower arrangement using a monotonic stack, providing a significant improvement over the brute-force O(n^2) approach.

**Refer to**

[L941.Valid Mountain Array (Ref.L1671)](note://WEB2db04147a5e836b22f6719e5810e3e91)

[L1671.Minimum Number of Removals to Make Mountain Array (Ref.L300)](note://A717168457D646DF8CE8A9054DD61C12)

[L84.Largest Rectangle in Histogram](note://E2FC22D9715540C5A8F06F60FBBE886F)