2908. Minimum Sum of Mountain Triplets I

Problem Description

<u>Array</u>

Easy

You are given an array of integers, where the array is indexed from 0 (0-indexed). The goal is to find a specific type of triplet within this array, referred to as a "mountain" triplet. A triplet (i, j, k) is defined as a mountain if it satisfies two conditions:

- The indices are in increasing order: i < j < k. • The values at these indices increase and then decrease, forming a peak: nums[i] < nums[j] and nums[k] < nums[j].
- - Your task is to find such a mountain triplet whose sum of elements (nums[i] + nums[j] + nums[k]) is as small as possible and

return this minimum sum. If no mountain triplet exists in the array, the function should return -1. Intuition

The problem is to find three numbers that form a peak (or a mountain) with the smallest possible sum. We could try to look at every possible triplet in the array, but that would be slow as it would require checking n choose 3 triplets, where n is the size of

the array. To optimize the process, we can leverage the sequential nature of the triplets and use the concept of preprocessing. The main insight is that for a valid mountain triplet, the middle element must be greater than the elements to its left and right. So if we fix

the middle element, we only need to find the smallest element to its left and the smallest to its right to minimize the sum.

The solution preprocesses the array to find the minimum value to the right of each element. It does so by traversing the array from right to left and keeps updating the minimum value seen so far. This information is stored in an array named right, where right[i] contains the minimum value in nums[i+1..n-1].

When we enumerate through each possible peak element nums [i], we keep track of the minimum value found to its left so far in a variable left. At the same time, we use a variable ans to maintain the current smallest sum found. For each potential middle element nums[i], if it is greater than both the minimum to its left (left) and the minimum to its right

(right[i+1]), it could be the peak of a minimum sum triplet. We update ans if the sum of these three elements is less than the

Solution Approach The solution makes intelligent use of a prefix and suffix strategy combined with a single pass through the array. The overall

approach is to prepare in advance the information needed for a quick lookup and to use this information to find the minimum

First, we initialize an array right to keep track of the minimum values to the right of each index. This array is initially filled with

inf (representing infinity), ensuring that if no smaller element exists to the right, the default large value prevents it from

triplet:

possible sum efficiently. Here's a breakdown of the implementation:

incorrectly contributing to a potential minimum triplet.

current ans.

We then fill the right array by iterating over nums backwards (from right to left). In each step, we assign to right [i] the minimum value between the current element nums [i] and the previously recorded minimum in right [i + 1]. This ensures that after this loop, for each index i, right[i] will contain the smallest element found in the subarray starting just after i until the end.

We initialize two more variables before the main loop: left and ans. Both are set to inf. The left variable will hold the

- minimum value to the left of the current index as we scan the array. The ans variable will keep track of the minimum sum of any valid mountain triplet found so far. The main loop iterates over the elements of the nums array, attempting to find the smallest sum of a mountain triplet with the current element as the peak. For each element nums [i], we check two conditions to see if it can be the peak of a mountain
- If both conditions are satisfied, we then calculate the potential minimum sum of the mountain triplet, which is left + nums[i] + right[i + 1]. If this sum is less than what is stored in ans, we update ans with the new minimum sum. In the same iteration, we also update the left variable, setting it to be the minimum between its current value and the current

element nums [i]. This ensures that left is always the smallest number to the left of the current element.

Let's illustrate the solution approach using a small example. Suppose we are given the following array of integers:

left < nums[i]: Is the current element greater than the smallest element found to its left?

right[i + 1] < nums[i]: Is the current element greater than the smallest element found to its right?

mountain triplet was found, and we return -1. Otherwise, we return the value of ans, which is the minimum sum of a mountain triplet.

This solution cleverly avoids the need for a complex three-level nested loop, which would result in a less efficient algorithm with

higher time complexity. Instead, it utilizes dynamic programming-like preprocessing and a single pass for a time-efficient linear

After the loop completes, we check if ans has been updated from its initial value of inf. If ans is still inf, it means no valid

scan of the array. **Example Walkthrough**

Filling the right array: We fill right by iterating from right to left starting at index n - 2, where n is the length of nums. We have:

values to avoid problems if no smaller right-hand side element is found. right = [inf, inf, inf, inf, inf, inf].

Initializing the right array: We first create an auxiliary array right with the same length as nums and initialize it with infinity

Now, our right array looks like this: [1, 1, 1, 3, 3, inf].

the answer, respectively.

mountain triplets.

from typing import List

class Solution:

nums = [2, 3, 1, 4, 3, 2]

• For i = 0: left is inf, and the current element is 2, which is not greater than left or right[i + 1]. Thus, we skip this. We update left to 2. • For i = 1: left is 2, and the current element is 3, which is greater than left and right[i + 1]. The sum here would be 2 + 3 + 1 = 6. Since ans is inf, we update ans to 6. We update left to 2 (since min(2,3) = 2).

∘ For i = 4 and i = 5: There are no elements to the right of i = 4 or i = 5 that are smaller than nums[i], so we do not find any valid

∘ For i = 3: left is 1, and the current element 4 is greater than both left and right[i + 1] which is 3. The sum is 1 + 4 + 3 = 8, but since

Checking and returning the answer: After this iteration, we find that ans holds the value 6, which is the minimum sum of a

∘ For i = 2: left remains 2, while right[i + 1] is 3. The current element is 1, so this cannot be a peak. We update left to 1.

mountain triplet that we have found. This sum corresponds to the triplet (2, 3, 1) at indices (0, 1, 2).

Therefore, our function would return 6 as the smallest possible sum of a mountain triplet from the given nums array.

Iterating over nums: We declare left as infinity (inf) and ans also as infinity. These will track the minimum left element and

Solution Implementation

ans is 6, we don't update ans. We update left to 1.

def minimum_sum(self, nums: List[int]) -> int:

from the current position to the end

Initialize the 'right_min' list with infinities

Populate the 'right_min' list with the minimum value

'left_min' will hold the minimum value from the start

if left_min < num and right_min[i + 1] < num:</pre>

return -1 if result == float('inf') else result

vector<int> minRight(size + 1, INF);

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

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

int minimumSum = INF;

let leftMin: number = Infinity;

for (let i = 0; i < length; i++) {</pre>

leftMin = Math.min(leftMin, nums[i]);

// Otherwise, return the minimum sum calculated

return minimumSum === Infinity ? -1 : minimumSum;

if (leftMin < nums[i] && rightMin[i + 1] < nums[i]) {</pre>

int minLeft = INF;

minRight[i] = min(minRight[i + 1], nums[i]);

If 'result' is still infinity, it means no valid sum was found

return -1 in this case, otherwise return the computed result

which will hold the minimum values from right

right_min = [float('inf')] * (num_elements + 1)

Find the length of the nums list

num_elements = len(nums)

to the current position

result = left_min = float('inf')

Iterate through the numbers

for i, num in enumerate(nums):

5th step: right[4] = min(nums[4], right[5]) = min(3, inf) = 3

4th step: right[3] = min(nums[3], right[4]) = min(4, 3) = 3

3rd step: right[2] = min(nums[2], right[3]) = min(1, 3) = 1

2nd step: right[1] = min(nums[1], right[2]) = min(3, 1) = 1

1st step: right[0] = min(nums[0], right[1]) = min(2, 1) = 1

Finding potential peaks: We iterate over the elements in the array:

- **Python**
- for i in range(num_elements -1, -1, -1): right_min[i] = min(right_min[i + 1], nums[i]) # Initialize 'result' (minimum sum) and 'left_min' with infinity

Check if both left and right numbers are less than the current number

if yes, then calculate the result as current number plus left and right minimums

result = min(result, left_min + num + right_min[i + 1]) # Update 'left_min' with the minimum value encountered so far left_min = min(left_min, num)

Java

class Solution {

```
public int minimumSum(int[] nums) {
        int n = nums.length;
       // Create an array to hold the minimum values from the right side.
        int[] minRight = new int[n + 1];
        final int INF = 1 << 30; // A representation of infinity (a very large number).
       minRight[n] = INF; // Initialize the last element as infinity.
       // Populate the minRight array with the minimum values from the right side.
        for (int i = n - 1; i \ge 0; --i) {
            minRight[i] = Math.min(minRight[i + 1], nums[i]);
        int answer = INF; // Initialize answer as infinity.
        int minLeft = INF; // Variable to keep track of the minimum value from the left side.
       // Iterate through the array to find the minimum sum that follows the given constraint.
        for (int i = 0; i < n; ++i) {
           // Check if the current number is greater than both the minimum values on its left and right.
            if (minLeft < nums[i] && minRight[i + 1] < nums[i]) {</pre>
                // Update the answer with the sum of the smallest elements on both sides of nums[i].
                answer = Math.min(answer, minLeft + nums[i] + minRight[i + 1]);
            // Update the minimum value from the left side.
           minLeft = Math.min(minLeft, nums[i]);
       // If the answer remains infinity, it means there were no valid sums found, so return -1.
       return answer == INF ? -1 : answer;
C++
class Solution {
public:
    int minimumSum(vector<int>& nums) {
       // Get the size of the vector
       int size = nums.size();
       // Define an infinite value for comparison purposes
       const int INF = 1 << 30;
```

// Create a vector to store the minimum from the right of each position, initialize with INF at the end

// Populate minRight with the minimum value from the right side in reverse order

// Initialize variables to store the minimum sum and the minimum on the left

// Iterate through numbers, updating minLeft and checking for a potential minimum sum

// Initialize 'leftMin' as Infinity to track the minimum value to the left of current index

// Check current element with minimum elements from both sides (excluding adjacent elements)

// Update 'minimumSum' if the current triplet sum is smaller than the previously found

// Iterate over the 'nums' array to find the minimum sum of non-adjacent array elements

minimumSum = Math.min(minimumSum, leftMin + nums[i] + rightMin[i + 1]);

// Update 'leftMin' with the minimum value found so far from the left

// If 'minimumSum' is still Infinity, it means no triplet was found, return -1

```
// If the current element is greater than the minimum on the left and right,
           // attempt to update the minimum sum
            if (minLeft < nums[i] && minRight[i + 1] < nums[i]) {</pre>
                minimumSum = min(minimumSum, minLeft + nums[i] + minRight[i + 1]);
           // Update the minimum on the left with the current element
           minLeft = min(minLeft, nums[i]);
       // If minimumSum remains INF, no valid sum is found; thus, return -1
       // Otherwise, return the calculated minimum sum
       return minimumSum == INF ? -1 : minimumSum;
};
TypeScript
function minimumSum(nums: number[]): number {
    // Calculate the length of nums array
    const length = nums.length;
    // Create an array 'rightMin' to store minimum values to the right of each element
    const rightMin: number[] = Array(length + 1).fill(Infinity);
    // Populate 'rightMin' with the minimum values observed from the end of the array
    for (let i = length - 1; i >= 0; i--) {
        rightMin[i] = Math.min(rightMin[i + 1], nums[i]);
    // Initialize 'minimumSum' as Infinity to track the minimum sum of non—adjacent array elements
    let minimumSum: number = Infinity;
```

```
from typing import List
class Solution:
   def minimum_sum(self, nums: List[int]) -> int:
       # Find the length of the nums list
        num_elements = len(nums)
       # Initialize the 'right_min' list with infinities
       # which will hold the minimum values from right
        right_min = [float('inf')] * (num_elements + 1)
       # Populate the 'right_min' list with the minimum value
       # from the current position to the end
        for i in range(num_elements - 1, -1, -1):
            right_min[i] = min(right_min[i + 1], nums[i])
       # Initialize 'result' (minimum sum) and 'left_min' with infinity
        # 'left_min' will hold the minimum value from the start
       # to the current position
        result = left_min = float('inf')
       # Iterate through the numbers
        for i, num in enumerate(nums):
            # Check if both left and right numbers are less than the current number
            # if yes, then calculate the result as current number plus left and right minimums
            if left_min < num and right_min[i + 1] < num:</pre>
                result = min(result, left_min + num + right_min[i + 1])
            # Update 'left_min' with the minimum value encountered so far
```

The time complexity of the given code is O(n). 1. The first for loop iterates from n-1 to 0, which results in n iterations.

Time and Space Complexity

left_min = min(left_min, num)

return -1 if result == float('inf') else result

If 'result' is still infinity, it means no valid sum was found

return -1 in this case, otherwise return the computed result

2. The second for loop iterates from 0 to n-1, which is also n iterations. Each of these loops runs in linear time relative to the number of elements n in the list nums. There are no nested loops, and each

Time Complexity

operation inside the loops runs in constant time 0(1). Hence, adding the time cost gives us a time complexity that is still linear: O(n) + O(n) = O(2n) which simplifies to O(n).

complexity is determined by the right list, giving 0(n + 1); this simplifies to 0(n).

Space Complexity The space complexity of the given code is O(n). The code uses an additional list right of size n+1 elements to keep track of the minimum elements to the right. Aside from the

right list and trivial variables (ans, left, i, and x), no additional space that scales with input size n is used. Therefore, the space