2909. Minimum Sum of Mountain Triplets II

Medium Array Leetcode Link

Problem Description

You are tasked with finding a specific type of triplet in an integer array `nums`, where the array indices start at 0. This triplet needs to follow the characteristics of a mountain, which means for indices `i`, `j`, `k` satisfying `i < j < k`, the elements at these indices must adhere to the pattern `nums[i] < nums[j]` and `nums[k] < nums[j]`. The goal is to determine the smallest possible sum of such a mountain triplet. If there are no triplets that form a mountain, you should return `-1`.

Intuition

The problem calls for an efficient way to find the smallest sum of a "mountain" within an array. To approach this, you would need to check each possible middle element (the peak of the mountain) to see if there's a smaller number before it (uphill) and a smaller number after it (downhill). A brute-force solution would be to try every possible triplet to find the minimum sum, but that would result in a high time complexity.

Instead, we can use pre-processing to optimize this procedure. By iterating through the array once, we can store the minimum value found to the right of each element in a new array called `right`. This pre-processing helps us quickly find the smallest element that can be the downhill part of the mountain for any middle element we're considering.

As we traverse the array to find the middle element of potential mountain triplets, we keep track of the smallest value encountered so far to serve as the uphill part of the mountain. This value is stored in a variable called `left`. At any point, if our current element is greater than both `left` and the corresponding value in `right`, we have found a valid mountain. We then check if the sum of `left`, the current element, and the corresponding `right` is less than the best answer we have found so far, updating our answer accordingly.

If after traversing the array no valid mountain is found, the answer remains at its initial infinite value and we return `-1`. Otherwise, we return the smallest sum found.

Solution Approach

The implementation of the solution is based on the Reference Solution Approach provided, which cleverly reduces the problem's complexity by pre-processing the array and using two pointers to track the minimum values required to identify a potential mountain.

Here's how the solution approach is implemented step by step:

1. **Pre-processing:** We create an array `right` which holds the minimum value to the right of each index in the `nums` array. We initialize every element in `right` to be `inf` (`inf` is a placeholder for infinity), which ensures that any number compared to it will be smaller. Starting from the second last element in `nums` and moving backwards, we populate `right[i]` with the minimum between `right[i + 1]` and `nums[i]`.

2. **One pass enumeration with two pointers:** As we iterate through the `nums` array using the variable `i`, we maintain two pointers/values:

   - `left`: The minimum value in the subarray to the left of the current index `i` (including `nums[i]`).
   - `ans`: The result variable which keeps track of the minimum sum of a valid mountain that has been found so far.
3. **Evaluating potential mountains:** For each element `nums[i]`, which we are considering to be the peak of the mountain (`j`), we check if we have a smaller number on its left (uphill) and on its right (downhill):

   - The check `left < nums[i]` ensures that the current peak is higher than the minimum value on the left.
   - The check `right[i + 1] < nums[i]` ensures that we have a valid downhill part for the mountain.
   - If both conditions are met, `nums[i]` can be the peak of a mountain, and we update `ans` with the sum of `left`, `nums[i]`, and `right[i + 1]` if it's smaller than the current `ans`.
4. **Updating the minimum left value:** After each iteration, we update `left` to be the minimum value between the current `left` and `nums[i]`.

5. **Returning the result:** Once we've iterated through the entire array, we check whether `ans` is still infinity. If it is, no valid mountain was found, and we return `-1`. Otherwise, we return the minimum sum stored in `ans`.

By following this approach, we avoid the need for a triple nested loop to check every possible triplet, which significantly optimizes the solution. The algorithm achieves a time complexity of O(n) since it only requires two passes through the `nums` array (one for pre-processing and one for the main iteration).

Example Walkthrough

To illustrate the solution approach, let's consider a small example of the `nums` array:

```
nums = [2, 1, 4, 7, 3, 2, 5]
```

Step 1: Pre-processing

We begin by creating a `right` array that will contain the minimum value to the right of each index, initializing all elements to `infinity`:

```
right = [inf, inf, inf, inf, inf, inf, inf]
```

We then update the `right` array by traversing from right to left:

```
Starting with the second to last index, right[5] is min(inf, 2) = 2
right = [inf, inf, inf, inf, inf, 2, inf]

right[4] is min(2, 3) = 2
right = [inf, inf, inf, inf, 2, 2, inf]

right[3] is min(2, 7) = 2
right = [inf, inf, inf, 2, 2, 2, inf]

right[2] is min(2, 4) = 2
right = [inf, inf, 2, 2, 2, 2, inf]

right[1] is min(2, 1) = 1
right = [inf, 1, 2, 2, 2, 2, inf]

right[0] will not be used since it must be the leftmost element of a triplet.
```

Final `right` array after pre-processing:

```
right = [inf, 1, 2, 2, 2, 2, inf]
```

Step 2: One pass enumeration with two pointers

We initialize two variables - `left` starting with `infinity` and `ans` also with `infinity`.

As we start iterating over the array, our pointers will update as follows:

```
left = inf
ans = inf
```

Step 3: Evaluating potential mountains

- When `i` is 0 (`nums[i]` is 2), `i` is at the leftmost edge, so `left` does not update because no valid mountain can start here.
- When `i` is 1 (`nums[i]` is 1), the left array so far [2] has a minimum of 2, and our `right` is 1. No valid mountain can have a peak at index 1 since `left >= nums[i]`.
- When `i` is 2 (`nums[i]` is 4), we have `left = 1` and `right[3]` is 2. This satisfies the mountain property (`left < nums[i] > right[3]`). We sum `1 + 4 + 2` to get the total sum of 7. Since `ans` was `infinity`, it is now updated to 7.
- When `i` is 3 (`nums[i]` is 7), a sum is possible here, but it won't be minimum because `right[4] = 2` and the minimum sum would be `1 + 7 + 2` which is 10, larger than our current `ans`.
- For `i` 4, 5, and 6, while a valley exists to the right (`right[i + 1] < nums[i]`), no smaller value exists to the left (`left >= nums[i]`), so no valid mountain can exist at these points.

Step 4: Updating the minimum left value

After each iteration, `left` is updated to the minimum value it has seen so far (if smaller than `nums[i]`), which helps efficiently find the uphill part for the next iterations.

Step 5: Returning the result

After iterating through the entire `nums` array, we find that `ans` is 7. Since it's no longer `infinity`, we have found at least one valid mountain, and `7` is the smallest sum of such a mountain triplet.

Therefore, the smallest possible sum of a mountain triplet in the `nums` array is `7`.

Python Solution

```python
from typing import List

class Solution:
    def minimumSum(self, nums: List[int]) -> int:
        # get the length of the nums list
        num_length = len(nums)
        # initialize a list to keep track of the minimum value to the right
        # including the current position, filled with infinity to start
        min_right = [float('inf')] * (num_length + 1)

        # populate the min_right list with the minimum values from right to left
        for i in range(num_length - 1, -1, -1):
            min_right[i] = min(min_right[i + 1], nums[i])

        # initialize the answer and the minimum value from the left with infinity
        answer = min_left = float('inf')

        # iterate over nums to find the minimum sum
        for i, x in enumerate(nums):
            # if the current number is greater than the smallest number found
            # to its left and to its right, consider it as a candidate for the answer
            if min_left < x and min_right[i + 1] < x:
                answer = min(answer, min_left + x + min_right[i + 1])

            # update min_left to the smallest number found so far from the left
            min_left = min(min_left, x)

        # return -1 if answer has not been updated, otherwise return the answer
        return -1 if answer == float('inf') else answer

```

Java Solution

```java
class Solution {
    public int minimumSum(int[] nums) {
        int n = nums.length; // Length of the input array
        int[] rightMin = new int[n + 1]; // Array to hold the minimum values from the right
        final int INF = Integer.MAX_VALUE; // Use Java's max value to represent infinity
        rightMin[n] = INF; // Set the rightmost value to infinity as a sentinel value

        // Populate rightMin with the minimum values scanned from right to left
        for (int i = n - 1; i >= 0; --i) {
            rightMin[i] = Math.min(rightMin[i + 1], nums[i]);
        }

        int answer = INF; // Initialize answer with the maximum possible value (infinity)
        int leftMin = INF; // Variable to keep track of the minimum value scanned from left

        // Iterate over the array to find the minimum sum with the specified conditions
        for (int i = 0; i < n; ++i) {
            // Check if both leftMin and rightMin[i+1] are less than nums[i]
            if (leftMin < nums[i] && rightMin[i + 1] < nums[i]) {
                // Update the answer with the minimum of previous answer
                // and the sum of leftMin, nums[i], and rightMin[i+1]
                answer = Math.min(answer, leftMin + nums[i] + rightMin[i + 1]);
            }
            // Update leftMin with the smallest value encountered so far
            leftMin = Math.min(leftMin, nums[i]);
        }

        // Return the answer if it's not infinity, otherwise return -1
        return answer != INF ? answer : -1;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Including algorithm for std::min
using std::vector;
using std::min;

class Solution {
public:
    int minimumSum(vector<int>& nums) {
        int nums_size = nums.size(); // Size of the nums vector.
        const int INFINITY = 1 << 30; // Representing infinity as a very large number.
        vector<int> right_smallest(nums_size + 1, INFINITY); // Vector to keep track of the right smallest elements.

        // Populate the right_smallest vector with the smallest value found from the right.
        right_smallest[nums_size] = INFINITY; // Initialization to infinity at the end of the vector.
        for (int i = nums_size - 1; i >= 0; --i) {
            right_smallest[i] = min(right_smallest[i + 1], nums[i]);
        }

        int result = INFINITY; // This will hold the final result, initialized to infinity.
        int left_smallest = INFINITY; // Keeping track of the smallest value from the left.

        // Iterate through the vector to find the minimum sum.
        for (int i = 0; i < nums_size; ++i) {
            // Check if the current element is greater than both the left and right smallest elements.
            if (left_smallest < nums[i] && right_smallest[i + 1] < nums[i]) {
                // Update the result with the minimum sum found so far.
                result = min(result, left_smallest + nums[i] + right_smallest[i + 1]);
            }
            // Update left_smallest to be the smallest value from the left up to the current position.
            left_smallest = min(left_smallest, nums[i]);
        }

        // If the result is still infinity, then a sum cannot be formed as per the problem's requirement.
        // Return -1 in that case. Otherwise, return the result.
        return result == INFINITY ? -1 : result;
    }
};

```

Typescript Solution

```typescript
function minimumSum(nums: number[]): number {
    // Initialize the length of the given array
    const numsLength = nums.length;

    // Create an array 'right' to keep track of the smallest number to the right
    // for each position, initialized to Infinity.
    const rightMinValues: number[] = Array(numsLength + 1).fill(Infinity);

    // Populate the 'rightMinValues' array from right to left
    for (let i = numsLength - 1; i >= 0; --i) {
        rightMinValues[i] = Math.min(rightMinValues[i + 1], nums[i]);
    }

    // Declare variables to keep track of the minimum answer and left minimum value
    let minAnswer = Infinity;
    let leftMinValue = Infinity;

    // Iterate over the array to compute the minimum sum of a valid triplet
    for (let i = 0; i < numsLength; ++i) {
        // Check if the current number is larger than the minimum on both sides
        if (leftMinValue < nums[i] && rightMinValues[i + 1] < nums[i]) {
            // Update the minimum answer with the sum of the triplet
            // if it is smaller than the current minimum
            minAnswer = Math.min(minAnswer, leftMinValue + nums[i] + rightMinValues[i + 1]);
        }
        // Store the minimum on the left up to the current element
        leftMinValue = Math.min(leftMinValue, nums[i]);
    }

    // Return the minimum answer or -1 if the answer remains Infinity (no valid triplet found)
    return minAnswer === Infinity ? -1 : minAnswer;
}

```

Time and Space Complexity

The time complexity of the provided code is indeed `O(n)`. This is because the code consists of two loops over the `nums` list, each running for `n` iterations, where `n` is the length of the input list `nums`. The first loop is a reverse iteration used to build the `right` list, and the second loop goes through the `nums` list to compute the `ans` variable. Since both loops are independent and execute sequentially, their combined time complexity remains linear with the size of the input list.

The space complexity of the code is `O(n)` as well due to the allocation of the `right` list, which stores the minimum value encountered from the right side of the `nums` list. This `right` list has a length equal to `n + 1`, where `n` is the length of the `nums` list. The other variables used (`ans`, `left`, and a few others) do not depend on `n` and thus contribute a constant factor, which is ignorable when assessing space complexity.