# 1800. Maximum Ascending Subarray Sum

Easy Array

## Problem Description

The task is to find the highest sum of a contiguous subarray within an array of positive integers `nums`. Each subarray must be sorted in an ascending order without breaks, which means each element in the subarray must be strictly greater than its preceding element. Note that a subarray composed of a single element counts as ascending.

## Intuition

The intuition behind the solution is to iterate through the array and maintain a running total `t` whenever the ascending condition is met. As soon as we hit a number that is not greater than the previous number, we must begin a new subarray and reset our running total to the current number's value because an ascending subarray can no longer continue past this point.

## Solution Approach

The solution is implemented in Python and follows a simple iterative approach to solve the problem. The code utilizes no extra data structures, operating directly on the input list to keep track of two important values:

- `t`: This variable holds the current sum of the latest ascending subarray being considered.
- `ans`: This is used to maintain the maximum sum encountered so far as we iterate through the array.

The iteration starts from the beginning of the list `nums`. For each number `v` at index `i`, we check whether it maintains an ascending order with respect to the previous number `nums[i - 1]`. The initial condition `i == 0` accounts for the start of the array, and by default, we begin with an ascending subarray consisting of the first element.

If the current number `v` is greater than the previous number, we are still in an ascending subarray. We add `v` to our current sum `t` and then compare it with `ans` to potentially update the maximum sum found. This comparison and potential update take place using the `max()` function:

```
ans = max(ans, t)
```

When the ascending condition breaks (the current number is not greater than the previous one), we need to reset the running sum `t` to the current number's value, as we are now starting a new ascending subarray:

```
t = v
```

This process continues until we've examined each number in the array. By the end of the iteration, `ans` contains the maximum sum of an ascending subarray, which is then returned.

The overall algorithm exhibits O(n) time complexity, where n is the number of elements in input array, as it requires a single pass through the list. No additional space is used beyond the input and constant variables, which results in an O(1) space complexity.

### Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the following array of positive integers:

```
nums = [10, 20, 70, 40, 50, 60]
```

As we go through `nums`, we will aggregate the sums of ascending subarrays and keep track of the maximum sum encountered. We will utilize two variables, `t` and `ans`, throughout the process. Follow these steps:

1. Initialize `t` with the first element and `ans` with 0, since we have not encountered any subarrays yet. `t = 10`, `ans = 0`
2. Move to the next element (20) and compare it with the previous one (10). Since 20 is greater than 10, it continues an ascending subarray. Add 20 to `t` and update `ans`. `t = 10 + 20 = 30`, `ans = max(0, 30) = 30`
3. The next element is 70, which is greater than 20, so add 70 to `t` and update `ans`. `t = 30 + 70 = 100`, `ans = max(30, 100) = 100`
4. We now encounter 40. Since 40 is not greater than the preceding element 70, the ascending condition is broken. Here, we start a new subarray. Thus, reset `t` to the value of the current element, 40, and `ans` remains the same as it's still the maximum sum encountered. `t = 40`, `ans = 100`
5. The next element is 50, which is greater than 40. We add 50 to `t` and compare it with `ans`. `t = 40 + 50 = 90`, `ans` remains 100 as 90 is not greater than 100.
6. Lastly, 60 is greater than 50, so add it to `t` and check against `ans`. `t = 90 + 60 = 150`, `ans = max(100, 150) = 150`

At the end of this process, `ans` holds the value of the highest sum of an ascending subarray within the array `nums`, which in this case is `150`. This sum comes from the subarray [40, 50, 60].

By following this iterative approach, we find the maximum sum efficiently with just one pass through the input array.

## Solution Implementation

### Python

```python
# The Solution class encapsulates the algorithm to find the maximum ascending subarray sum.
class Solution:
    def maxAscendingSum(self, nums: List[int]) -> int:
        max_sum = temp_sum = nums[0]  # Initialize max_sum and temp_sum with the first element

        # Iterate through the list starting from the second element
        for i in range(1, len(nums)):
            # If the current element is greater than the previous element, add it to the temp_sum.
            if nums[i] > nums[i - 1]:
                temp_sum += nums[i]
            else:
                # Else, assign current element to temp_sum as the start of a new subarray.
                temp_sum = nums[i]

            # Update max_sum if temp_sum is greater.
            max_sum = max(max_sum, temp_sum)

        return max_sum  # Return the maximum sum found.
```

### Java

```java
class Solution {
    public int maxAscendingSum(int[] nums) {
        int maxSum = 0; // This variable will store the maximum ascending subarray sum
        int currentSum = 0; // This variable will keep the current subarray sum

        // Iterate over all the elements in the array
        for (int i = 0; i < nums.length; ++i) {
            // Check if the current element is greater than the previous element or it is the first element
            if (i == 0 || nums[i] > nums[i - 1]) {
                currentSum += nums[i]; // Add the current element to the current sum
                maxSum = Math.max(maxSum, currentSum); // Update the maxSum with the larger of the two sums
            } else {
                currentSum = nums[i]; // If the current element is not greater then start a new subarray from the current elemen
            }
        }
        return maxSum; // Return the maximum sum of ascending subarray found
    }
}
```

### C++

```cpp
class Solution {
public:
    // Function to find the maximum ascending subarray sum.
    int maxAscendingSum(vector<int>& nums) {
        int maxSum = 0; // Variable to store the maximum sum of ascending subarray
        int currentSum = 0; // Variable to store the current subarray sum

        // Iterate through all the elements of the input vector
        for (int i = 0; i < nums.size(); ++i) {
            // Start a new subarray if we are at the first element or if the current element
            // is greater than the previous one, thus obeying the ascending order
            if (i == 0 || nums[i] > nums[i - 1]) {
                currentSum += nums[i]; // Accumulate current subarray sum
                maxSum = max(maxSum, currentSum); // Update maxSum if currentSum is greater
            } else {
                // If the current element is not greater than the previous one,
                // start a new subarray sum from the current element
                currentSum = nums[i];
            }
        }

        // Return the maximum sum of ascending subarray found
        return maxSum;
    }
};
```

### TypeScript

```typescript
/**
 * Calculates the maximum ascending subarray sum in an array of numbers
 * @param nums - The given array of numbers
 * @returns The maximum sum of an ascending subarray
 */
function maxAscendingSum(nums: number[]): number {
    const length = nums.length; // Length of the input array
    let maxSum = nums[0]; // Initialize maxSum as the first element
    let currentSum = nums[0]; // Initialize currentSum as the first element

    // Iterate through the array starting from the second element
    for (let i = 1; i < length; i++) {
        // If the current element is not larger than the previous one,
        // compare and update the maxSum with the currentSum so far,
        // then reset the currentSum to the current element
        if (nums[i] <= nums[i - 1]) {
            maxSum = Math.max(maxSum, currentSum);
            currentSum = nums[i];
        } else {
            // If the current element is larger, add it to the currentSum
            currentSum += nums[i];
        }
    }

    // Return the maximum sum between maxSum and the currentSum
    // to cover the case where the last element was part of the ascending sequence
    return Math.max(maxSum, currentSum);
}
```

```python
# The Solution class encapsulates the algorithm to find the maximum ascending subarray sum.
class Solution:
    def maxAscendingSum(self, nums: List[int]) -> int:
        max_sum = temp_sum = nums[0]  # Initialize max_sum and temp_sum with the first element

        # Iterate through the list starting from the second element
        for i in range(1, len(nums)):
            # If the current element is greater than the previous element, add it to the temp_sum.
            if nums[i] > nums[i - 1]:
                temp_sum += nums[i]
            else:
                # Else, assign current element to temp_sum as the start of a new subarray.
                temp_sum = nums[i]

            # Update max_sum if temp_sum is greater.
            max_sum = max(max_sum, temp_sum)

        return max_sum  # Return the maximum sum found.
```

## Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` list. This is because the code iterates through the `nums` list once, with each operation within the loop having a constant time complexity.

The space complexity of the code is `O(1)` as it uses a fixed amount of extra space; only two integer variables `ans` and `t` are used, regardless of the input size.