1800. Maximum Ascending Subarray Sum

Given an array of positive integers `nums`, return the *maximum possible sum of an* ***ascending*** *subarray in* `nums`.

A subarray is defined as a contiguous sequence of numbers in an array.

A subarray `[nums l, nums l+1, ..., nums r-1, nums r]` is **ascending** if for all `i` where `l <= i < r`, `nums i < nums i+1`. Note that a subarray of size `1` is **ascending**.

Example 1:

```
Input: nums = [10,20,30,5,10,50]
Output: 65
Explanation: [5,10,50] is the ascending subarray with the maximum sum of 65.
```

Example 2:

```
Input: nums = [10,20,30,40,50]
Output: 150
Explanation: [10,20,30,40,50] is the ascending subarray with the maximum sum of
150.
```

Example 3:

```
Input: nums = [12,17,15,13,10,11,12]
Output: 33
Explanation: [10,11,12] is the ascending subarray with the maximum sum of 33.
```

Constraints:

- `1 <= nums.length <= 100`
- `1 <= nums[i] <= 100`

Overview

We need to find the highest possible sum of an ascending subarray in a given array of positive integers. An ascending subarray is a contiguous sequence where each element is strictly smaller than the next (i.e., `nums[i] < nums[i+1]` for all valid indices). A subarray of size 1 is always considered ascending because there are no adjacent elements to compare.

> **Note:** There is a difference between "ascending" and "non-decreasing." "Ascending" means strictly increasing, where each value is greater than the previous one. On the other hand, "non-decreasing" allows the values to stay the same or increase, so equality is permitted.
> For example:
>
> - `1 2 3 4 5` is **ascending** (strictly increasing).
> - `1 2 2 3 3 4 5` is **non-decreasing** (values can remain the same or increase).

1800. Maximum Ascending Subarray Sum

Intuition

Instead of using the brute-force method of checking every possible subarray, which is inefficient, we can solve the problem in a single pass through the array.

The key idea is to keep extending the current subarray as long as it stays ascending. If we encounter an element that isn't greater than the previous one, we stop and compare the current subarray's sum with the largest sum we've found so far and update it if the current element is not greater than the previous one. Then, we reset the current sum to the current element and start a new subarray from there.

This strategy works because, with all numbers being positive, extending a subarray will always increase its sum. Thus, we should never start a new subarray when we can extend the current one. For an added challenge, try solving the similar problem 53. Maximum Subarray (Kadane's algorithm), where the numbers are not restricted to be positive.

By following this idea, we only need to go through the array once. At the end of the loop, we perform a final check to ensure we account for the last subarray, just in case it had the largest sum.

The algorithm is visualized below:

1800. Maximum Ascending Subarray Sum

```
/*
   Single pass, single pointer
   Time complexity: O(n)
   Space complexity: O(1)
*/
typedef std::vector<int> vi;

class Solution {
   public:
      int maxAscendingSum(vi& nums) {
         int n=nums.size();
         int sum=nums[0]; // Prefix sum initial value
         int ans=nums[0]; // Array could be of size 1
         // Iterate over all elements
         for(int i=0;i<n-1;++i){
            // If previous element is greater than the next one
            if(nums[i]<nums[i+1]){
               // Cumulate the sum
               sum+=nums[i+1];
            }
            // Otherwise, reset the sum the the value that breaks the condition
            else sum=nums[i+1];

            // Maximize the sum
            ans=std::max(ans,sum);
         }
         return ans;
      }
};
```

1800. Maximum Ascending Subarray Sum

```
/*
    Single pass, single pointer
    Time complexity: O(n)
    Space complexity: O(1)
*/
typedef std::vector<int> vi;

class Solution {
    public:
        int maxAscendingSum(vi& nums) {
            int n=nums.size();
            int sum=nums[0],ans=0;
            for(int i=0;i<n-1;++i){
                if(nums[i]>=nums[i+1]){
                    ans=std::max(ans,sum);
                    sum=0;
                }
                sum+=nums[i+1];
            }
            return std::max(ans,sum);
        }
};
```