<https://leetcode.com/problems/maximum-ascending-subarray-sum/description/>

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 [numsl, numsl+1, ..., numsr-1, numsr] is ascending if for all i where l <= i < r, numsi < numsi+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

**Attempt 1: 2024-08-26**

**Solution 1: Intuitive scan array (10 min)**

**Wrong Solution**

**Error out on input:**

**nums = [100,10,1]**

**Output = 10**

**Expected = 100**

class Solution {

    public int maxAscendingSum(int[] nums) {

        int maxSum = 0;

        int curMax = nums[0];

        for(int i = 1; i < nums.length; i++) {

            if(nums[i] > nums[i - 1]) {

                curMax += nums[i];

            } else {

                curMax = nums[i];

            }

            maxSum = Math.max(maxSum, curMax);

        }

        return maxSum;

    }

}

**Correct solution**

**Just one line change:**

**'int maxSum = 0;' change to 'int maxSum = curMax;' and move after 'int curMax = nums[0];'**

class Solution {

    public int maxAscendingSum(int[] nums) {

        int curMax = nums[0];

        int maxSum = curMax;

        for(int i = 1; i < nums.length; i++) {

            if(nums[i] > nums[i - 1]) {

                curMax += nums[i];

            } else {

                curMax = nums[i];

            }

            maxSum = Math.max(maxSum, curMax);

        }

        return maxSum;

    }

}

**Refer to**

<https://leetcode.com/problems/maximum-ascending-subarray-sum/solutions/1168768/java-one-pas-solution/>

class Solution {

public int maxAscendingSum(int[] nums) {

int res = nums[0],temp = nums[0];

for(int i = 1;i<nums.length;i++){

if(nums[i] > nums[i-1])

temp+=nums[i];

else

temp = nums[i];

res = Math.max(res,temp);

}

return res;

}

}

**Refer to chatGPT**

LeetCode problem 1800 is titled "Maximum Ascending Subarray Sum." In this problem, you are given an array of positive integers, and you need to find the maximum possible sum of an ascending subarray in the array. An ascending subarray is defined as a contiguous subarray in which each element is strictly greater than the previous one.

**And why not using prefix sum tech ?**

**Using the prefix sum technique to solve this problem isn't straightforward because the problem specifically involves finding the maximum sum of a contiguous ascending subarray, which requires you to restart the sum when the sequence is no longer ascending. However, we can still leverage the idea of prefix sums by using them conditionally within the loop.**

Here’s a Java solution to solve this problem:

class Solution {

public int maxAscendingSum(int[] nums) {

int maxSum = 0;

int currentSum = nums[0];

// Loop through the array starting from the second element

for (int i = 1; i < nums.length; i++) {

if (nums[i] > nums[i - 1]) {

// If the current number is greater than the previous one,

// add it to the current sum

currentSum += nums[i];

} else {

// If not, update maxSum and reset currentSum to the current number

maxSum = Math.max(maxSum, currentSum);

currentSum = nums[i];

}

}

// After the loop, check one last time to update maxSum

maxSum = Math.max(maxSum, currentSum);

return maxSum;

}

}

**Explanation:**

Initialization:

maxSum is initialized to 0 to keep track of the maximum ascending subarray sum found so far.

currentSum is initialized to the first element of the array because every subarray must start with at least one element.

Iteration:

The loop starts from the second element (i = 1) since we are checking each element against its previous one.

If the current element nums[i] is greater than the previous one nums[i-1], it means we are still in an ascending subarray, so we add nums[i] to currentSum.

If nums[i] is not greater than nums[i-1], it means the ascending order is broken. We update maxSum with the maximum value between maxSum and currentSum and reset currentSum to the current element nums[i].

Final Check:

After the loop ends, we perform one last check to ensure that the last currentSum is considered.

Return:

Finally, return maxSum as the maximum sum of an ascending subarray found in the array.

**Example**

For example, if the input array is:

int[] nums = {10, 20, 30, 5, 10, 50};

Solution solution = new Solution();

int result = solution.maxAscendingSum(nums); // Output will be 65

The output will be 65 because the subarray [5, 10, 50] has the maximum ascending sum.