<https://leetcode.ca/2022-04-21-2219-Maximum-Sum-Score-of-Array/>

You are given a **0-indexed** integer array nums of length n.

The **sum score** of nums at an index i where 0 <= i < n is the **maximum** of:

The sum of the **first** i + 1 elements of nums.

The sum of the **last** n - i elements of nums.

Return **the maximum sum score of nums at any index.**

**Example 1:**

Input: nums = [4,3,-2,5]

Output: 10

**Explanation:**

The sum score at index 0 is max(4, 4 + 3 + -2 + 5) = max(4, 10) = 10.

The sum score at index 1 is max(4 + 3, 3 + -2 + 5) = max(7, 6) = 7.

The sum score at index 2 is max(4 + 3 + -2, -2 + 5) = max(5, 3) = 5.

The sum score at index 3 is max(4 + 3 + -2 + 5, 5) = max(10, 5) = 10.

The maximum sum score of nums is 10.

**Example 2:**

Input: nums = [-3,-5]

Output: -3

**Explanation:**

The sum score at index 0 is max(-3, -3 + -5) = max(-3, -8) = -3.

The sum score at index 1 is max(-3 + -5, -5) = max(-8, -5) = -5.

The maximum sum score of nums is -3.

**Constraints:**

n == nums.length

1 <= n <= 10^5

-10^5 <= nums[i] <= 10^5

**Attempt 1: 2024-12-15**

**Solution 1: Prefix Sum (10 min)**

**Refer to chatGPT**

class Solution {

public long maximumSumScore(int[] nums) {

int n = nums.length;

long totalSum = 0;

for (int num : nums) {

totalSum += num;

}

long leftSum = 0;

long maxScore = Long.MIN\_VALUE;

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

leftSum += nums[i];

// Calculate rightSum as totalSum - leftSum + nums[i]

long rightSum = totalSum - leftSum + nums[i];

// Update maxScore with the larger of leftSum or rightSum

maxScore = Math.max(maxScore, Math.max(leftSum, rightSum));

}

return maxScore;

}

}

Time Complexity: O(n)

Space Complexity: O(1)

**Explanation**

Total Sum Calculation:

First, compute the total sum of the array. This helps calculate the sum of the right subarray efficiently.

Iterate Through the Array:

Maintain a running leftSum as you iterate through the array.

Calculate the rightSum for each index as:
rightSum=totalSum−leftSum+nums[i]\text{rightSum} = \text{totalSum} - \text{leftSum} + \text{nums[i]}rightSum=totalSum−leftSum+nums[i]

This ensures that both the left subarray (up to index i) and the right subarray (from index i onward) are considered.

Update Maximum Score:

At each step, compare the leftSum and rightSum and update the maximum score (maxScore).

Return Result:

After the loop, maxScore contains the maximum sum score for any subarray.

**Refer to**

<https://algo.monster/liteproblems/2219>

In this problem, you're provided with an array of integers named nums indexed from 0 to n-1, where n is the length of the array. You need to calculate what is called the **sum score** for each index of the array. The **sum score** at a particular index i is defined as the maximum between two sums:

The sum of elements from the start of the array up through index i.

The sum of elements from index i through the end of the array.

Your task is to determine the maximum **sum score** that can be obtained at any index i in the provided array.

**Intuition**

The key to solving this problem lies in understanding prefix and suffix sums, which are cumulative sums of the elements of the array from the beginning up to a certain element, and from a certain element to the end, respectively.

Calculate the [prefix sum](https://algo.monster/problems/subarray_sum) for the array, which gives you the sum of elements from the start of the array up to each index i. This is stored in an array s where s[i] would represent the sum of elements from nums[0] to nums[i-1].

Iterate through the array to calculate the sum score at each index. This can be done in a single pass after the [prefix sum](https://algo.monster/problems/subarray_sum) array has been built:

For each index i, determine the sum from the start of the array to index i. This is given by the prefix sum at index i + 1 (since we initialized prefix sum with 0 at the beginning).

To determine the sum from index i to the end of the array, subtract the prefix sum up to index i from the total sum of the array, which is the last element of the prefix sum array.

The sum score at index i is the maximum of these two sums.

After calculating the sum scores for each index, the maximum sum score is the answer, which represents the maximum value obtained from any index i.

By using a [prefix sum](https://algo.monster/problems/subarray_sum) array and iterating through the array only once, we can solve this problem efficiently with a time complexity of O(n), where n is the length of the input array.

**Solution Approach**

The given solution implements the intuition behind the problem in Python. It utilizes the accumulate function from the itertools module to generate prefix sums efficiently and a simple loop to compare sums at each index.

Here is a step-by-step explanation of the provided code:

**Step 1: Calculate Prefix Sums**

The line s = [0] + list(accumulate(nums)) is crucial. It creates a new list s that stores the prefix sums of the nums array. The accumulate function takes each element and adds it to the sum of all the previous elements. The [0] at the start of this list is to simplify calculations for the [prefix sum](https://algo.monster/problems/subarray_sum) at index 0. After this line, s[i] will contain the sum of nums from nums[0] up to nums[i-1].

**Step 2: Iterate to Find Maximum Sum Score**

The expression (max(s[i + 1], s[-1] - s[i]) for i in range(len(nums))) uses a generator to calculate sum scores without storing them. For each index i in nums, it calculates two sums:

s[i + 1] gives us the sum of elements from the beginning of the array to index i ([prefix sum](https://algo.monster/problems/subarray_sum)

s[-1] - s[i] gives us the sum of elements from index i to the end of the array (this works because s[-1] represents the total sum of the array, which is the last element in the prefix sums list).

**Step 3: Find the Maximum Value**

The max function is used again to find the maximum sum score from the generator. This is the maximum value that can be obtained from any of the sum scores calculated in the previous step.

The result of the max function call is returned as the final answer, representing the overall maximum sum score at any index i in the input nums array.

**Algorithm Pattern**

The algorithm follows a pattern often used for solving cumulative sum problems: it first preprocesses the input array to build a data structure (in this case, a [prefix sum](https://algo.monster/problems/subarray_sum) array), enabling efficient calculations of subarray sums. It then iterates through the array a single time to find the desired maximum value.

**Data Structure**

An array (or list in Python) is the primary data structure used here for storing the prefix sums.

**Time Complexity**

Since each of the steps above runs in O(n) time, where n is the length of the nums array, and there are no nested loops, the overall time complexity of the function is O(n).

**Solution Implementation**

class Solution {

public long maximumSumScore(int[] nums) {

int length = nums.length;

// Create an array to store the prefix sums

long[] prefixSums = new long[length + 1];

// Calculate the prefix sums

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

prefixSums[i + 1] = prefixSums[i] + nums[i];

}

// Initialize the maximum sum as the smallest possible value

long maxSum = Long.MIN\_VALUE;

// Find the maximum sum score by choosing the larger between

// the sum from the start to the current element or

// the sum from the current element to the end

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

// As design, prefixSum[i] mapping to sum up between nums[0, i - 1],

// so prefixSum[i + 1] will includes nums[i] as mapping to [0, i],

// so prefixSums[length] - prefixSums[i] will be nums[i, length - 1]

long sumFromStart = prefixSums[i + 1];

long sumFromEnd = prefixSums[length] - prefixSums[i];

maxSum = Math.max(maxSum, Math.max(sumFromStart, sumFromEnd));

}

// Return the maximum score found

return maxSum;

}

}

**Time and Space Complexity**

The time complexity of the given code is O(n), where n is the number of elements in the input list nums. This is due to the following reasons:

We first precompute the prefix sums with accumulate(nums), which takes O(n) time.

Then, we perform a single pass over the nums list to calculate the maximum sum score. During each iteration, we calculate the maximum between s[i + 1] and s[-1] - s[i]. There are n such calculations, each taking constant time.

The space complexity of the code is O(n). This is because we are creating a new list called s that contains the prefix sums of the nums array, which is of size n + 1.

**Refer to**

[L560.Subarray Sum Equals K](note://5CE29DE019904808986311AFAC74C85C)