LeetCode 53. Maximum Subarray

1. Problem Title & Link

- **53. Maximum Subarray**
- https://leetcode.com/problems/maximum-subarray/

2. Problem Statement (Short Summary)

Given an integer array nums, find the **contiguous subarray** (containing at least one number) which has the **largest sum**, and return its sum.

3. Examples (Input → Output)

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.

Input: nums = [1]
Output: 1

Input: nums = [5,4,-1,7,8]
Output: 23

4. Constraints

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4

5. Thought Process (Step by Step)

Two main views:

Kadane's Greedy (preferred) — scan once, keep track of:

current_sum = maximum subarray sum ending at current index
max_sum = global maximum found so far
Update:

current_sum = max(nums[i], current_sum + nums[i])
max_sum = max(max_sum, current_sum)

Intuition: either extend previous subarray or start fresh at current index.

DP view — same as Kadane: dp[i] = max(nums[i], dp[i-1] + nums[i]).
But we can do it in O(1) space using rolling variables.
Kadane is O(n) time, O(1) space and is ideal for interviews.

6. Pseudocode

```
max_sum = nums[0]
current_sum = nums[0]

for i from 1 to n-1:
   current_sum = max(nums[i], current_sum + nums[i])
   max_sum = max(max_sum, current_sum)

return max_sum
```

7. Code Implementation

✅ Python

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        max_sum = nums[0]
        current_sum = nums[0]
        for x in nums[1:]:
            current_sum = max(x, current_sum + x)
            max_sum = max(max_sum, current_sum)
        return max_sum
```

✅ Java

```
class Solution {
    public int maxSubArray(int[] nums) {
        int currentSum = nums[0];
        int maxSum = nums[0];
        for (int i = 1; i < nums.length; i++) {
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }
        return maxSum;
    }
}
```

8. Time & Space Complexity

- **Time:** O(n) — single pass
- **Space:** O(1) — constant extra space

9. Dry Run (Step-by-Step Execution)

👉 Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Initialize: current_sum = -2, max_sum = -2

i	x	current_sum = max(x, prev + x)	max_sum
0	-2	-2	-2
1	1	max(1, -2+1 = -1) → 1	max(-2,1) → 1
2	-3	max(-3, 1-3 = -2) → -2	1
3	4	max(4, -2+4 = 2) → 4	max(1,4) → 4
4	-1	max(-1, 4-1 = 3) → 3	4
5	2	max(2, 3+2 = 5) → 5	max(4,5) → 5
6	1	max(1, 5+1 = 6) → 6	max(5,6) → 6
7	-5	max(-5, 6-5 = 1) → 1	6
8	4	max(4, 1+4 = 5) → 5	6

Final max_sum = 6 (from subarray [4,-1,2,1]).

10. Concept Insight Table

Core Concept	Common Use Cases	Common Traps	Builds / Next Steps
Kadane's Algorithm (Greedy / DP) — maintain best subarray ending	- Max subarray / subsegment problems - Variants: max product subarray, circular subarray -	- Trying to use sliding window incorrectly for negative numbers - Forgetting all-negative arrays (initialize with nums[0]) - Using	🔷 Builds to **LeetCode 918 (Max Sum Circular Subarray)** 🔷 Related: **Maximum Product Subarray (152)**, **subarray sum**

11. Common Mistakes / Edge Cases

- Initializing max_sum or current_sum to 0 (fails when all numbers negative). Always initialize with nums[0].
- Trying two-pointer sliding-window — that requires non-negative constraints.
- Using O(n^2) brute force for large arrays causes TLE.

12. Variations / Follow-Ups

- **Maximum Product Subarray (LC 152)** — similar idea but needs care with negative numbers.
- **Max Subarray Sum Circular (LC 918)** — combine Kadane twice to handle wraparound.
- **Find subarray (indices) with max sum** — keep track of start/end pointers when updating max_sum.