1)Naive Solution (Brute Force)

- **Idea**
- The naive approach is based on **enumerating all possible contiguous subarrays** and computing their sums, then selecting the maximum one.
- **Input**
- An array A of n integers (may include positive and negative values).
- **Output**
- The maximum sum of any contiguous subarray.

Pseudocode

```
• MaxSum = -∞
• for i = 0 to n-1:
•    for j = i to n-1:
•       sum = 0
•       for k = i to j:
•          sum = sum + A[k]
•       if sum > maxSum:
•          maxSum = sum
• return maxSum
```

Time Complexity Analysis

- Three nested loops, Total time complexity: **$O(n^3)$,** Space complexity: **O(1)**

Improved Naive Version (Prefix / Accumulated Sum)

Instead of recomputing the sum from scratch, we accumulate the sum while extending the subarray

Pseudocode

```
maxSum = -∞
for i = 0 to n-1:
   sum = 0
   for j = i to n-1:
      sum = sum + A[j]
      if sum > maxSum:
         maxSum = sum
return maxSum
```

Complexity

- Time complexity: $O(n^2)$
- Space complexity: $O(1)$

2) Identification (Why Naive is Inefficient)

- The naive solution **checks all possible subarrays**, even when it is clear that extending a subarray with a negative sum cannot produce an optimal result.
- There is **overlapping computation** of sums.
- This motivates the need for an optimized approach.

---

3) Transition to Optimized Solution (Kadane's Algorithm)

Observation:

- If the current subarray sum becomes negative, it is better to **start a new subarray**.
- We only need to track:
    - currentSum: best sum ending at current index
    - bestSum: maximum sum found so far

This observation leads directly to **Kadane's Algorithm**, which runs in **O(n)** time