package com.iimtiaz.day\_03;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class MaximumSubarray {  
 public static void main(String[] args) {  
 int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};  
 System.*out*.println(new Solution\_1().maxSubArray(nums));  
 System.*out*.println(new Solution\_2().maxSubArray(nums));  
 System.*out*.println(new Solution\_3().*maxSubArray*(nums));  
 }  
}  
  
// A brute force solution that generates all possible subarrays and calculates their sums. This approach has a time  
// complexity of O(n^3), which is not efficient for large inputs due to the constraints (1 <= nums.length <= 10^5).  
// This is likely why solution isn’t passing all the test cases, as it may be exceeding the time limit for large inputs.  
  
*/\*\*  
 Time complexity: O(n^3)  
 O(n^2): Outer and inner loops iterating over all possible subarray sizes and start positions.  
 O(n): Nested loop within the inner loop iterates over each element within the current subarray size and start position.  
 O(1): Adding elements to the subarray list and finding the max are constant time operations.  
  
 Space complexity: O(n^2)  
 O(n^2): The result list stores all generated subarrays, leading to quadratic space complexity.  
 O(n): Temporary subarray list within the inner loop contributes additional linear space complexity.  
 \*/*class Solution\_1 {  
 public int maxSubArray(int[] nums) {  
 List<List<Integer>> result = new ArrayList<>();  
 //The outer loop variable size starts at 0 (empty subarray) and goes up to the length of the nums array  
 // (largest possible subarray).  
 for (int size = 0; size <= nums.length; size++) {  
 //The variable start starts at 0 and goes up to the length minus the current size, ensuring the subarray  
 // ends within the nums array.  
 for (int start = 0; start + size <= nums.length; start++) {  
 //Creates a new empty List<Integer> for each iteration of the inner loop.  
 //It then iterates from the current start position to start plus the current size.  
 //Within the inner loop, each element at index i from the nums array is added to the subarray list.  
 List<Integer> subarray = new ArrayList<>();  
 for (int i = start; i < start + size; i++) {  
 subarray.add(nums[i]);  
 }  
 result.add(subarray);  
 }  
 }  
  
 Integer max = Integer.*MIN\_VALUE*;  
 for (List<Integer> subarray : result) {  
 //System.out.println("Subarray: " + subarray);  
 Integer current = 0;  
 for (Integer s : subarray) {  
 current += s;  
 }  
 //System.out.println("Sum: " + current);  
 //System.out.println();  
 max = Math.*max*(max, current);  
 }  
 return max;  
 }  
}  
  
*/\*\*  
 Time complexity: O(n)  
 O(1): Initializing variables like n, max\_so\_far, and max\_ending\_here.  
 O(n): Single loop iterating through each element of the array once.  
 O(1): Operations like addition, comparison, and assignment are considered constant time on average.  
  
 Space complexity: O(1)  
 Only a few fixed-size variables are used (e.g., n, max\_so\_far, max\_ending\_here). No additional data structures  
 are created during the calculation.  
 \*/*class Solution\_2 {  
 public int maxSubArray(int[] a) {  
 int n = a.length;  
 int max\_so\_far = Integer.*MIN\_VALUE*;  
 int max\_ending\_here = 0;  
 for (int i = 0; i < n; i++) {  
 max\_ending\_here += a[i];  
 max\_so\_far = Math.*max*(max\_ending\_here, max\_so\_far);  
 if (max\_ending\_here < 0) {  
 max\_ending\_here = 0;  
 }  
 }  
 return max\_so\_far;  
 }  
}  
  
*/\*\*  
 Time complexity: O(n)  
 O(1): Initializing variables like currentSum and maxSum.  
 O(n): Single loop iterating through each element of the array once.  
 O(1): Operations like addition, comparison, and assignment are considered constant time on average.  
  
 Space complexity: O(1)  
 Only a few fixed-size variables are used (e.g., currentSum, maxSum). No additional data structures are  
 created during the calculation.  
\*/*class Solution\_3 {  
 public static 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;  
 }  
}