Maximum sum of a subarray of size k - GeeksforGeeks

Source: https://www.geeksforgeeks.org/find-maximum-minimum-sum-subarray-size-k/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Maximum sum of a subarray of size k Last Updated : 25 Aug, 2025 Given an array of integers arr[] and an integer k , find the maximum possible sum among all contiguous subarrays of size exactly k . A subarray is a sequence of consecutive elements from the original array. Return the maximum sum that can be obtained from any such subarray of length k. Examples: Input : arr[] = [100, 200, 300, 400], k = 2 Output : 700 Explanation: We get maximum sum by adding subarray [300,400] of size 2 Input : arr[] = [1, 4, 2, 10, 23, 3, 1, 0, 20], k = 4 Output : 39 Explanation: We get maximum sum by adding subarray [4, 2, 10, 23] of size 4. Input : arr[] = [2, 3], k = 1 Output : 3 Explanation: The subarrays of size 1 are [2] and [3]. The maximum sum is 3. Try it on GfG Practice Table of Content [Naive Approach] Fixed-Size Window Brute Force - O(n × k) time and O(1) space [Better Approach - 1] Using Prefix Sum - O(n) Time and O(n) space [Better Approach - 2] Sliding Window using Queue - O(n) Time and O(k) Space [Expected Approach] Optimized Sliding Window - O(n) Time and O(1) Space [Naive Approach] Fixed-Size Window Brute Force - O(n × k) time and O(1) space The idea is to iterate over all possible subarrays of size k and calculate their sums one by one. For each subarray, compare its sum with the current maximum and update accordingly. C++ #include <iostream> #include <vector> using namespace std ; int maxSubarraySum (vector < int >& arr , int k) { int n = arr . size (); int maxSum = 0 ; // check all subarrays of size k for (int i = 0 ; i <= n - k ; i ++) { int currSum = 0 ; // compute sum of current subarray for (int j = 0 ; j < k ; j ++) { currSum += arr [i + j]; } // update maximum sum maxSum = max (maxSum , currSum); } return maxSum ; } int main () { vector < int > arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; cout << maxSubarraySum (arr , k) << endl ; return 0 ; } Java class GfG { // check all subarrays of size k static int maxSubarraySum (int [] arr , int k) { int n = arr . length ; int maxSum = 0 ; for (int i = 0 ; i <= n - k ; i ++) { int currSum = 0 ; // compute sum of current subarray for (int j = 0 ; j < k ; j ++) { currSum += arr [i + j] ; } // update maximum sum maxSum = Math . max (maxSum , currSum); } return maxSum ; } public static void main (String [] args) { int [] arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; System . out . println (maxSubarraySum (arr , k)); } } Python def maxSubarraySum (arr , k): n = len (arr) maxSum = 0 # check all subarrays of size k for i in range (n - k + 1): currSum = 0 # compute sum of current subarray for j in range (k): currSum += arr [i + j] # update maximum sum maxSum = max (maxSum , currSum) return maxSum if __name__ == "__main__" : arr = [2 , 1 , 5 , 1 , 3 , 2] k = 3 print (maxSubarraySum (arr , k)) C# using System ; class GfG { // check all subarrays of size k static int maxSubarraySum (int [] arr , int k) { int n = arr . Length ; int maxSum = 0 ; for (int i = 0 ; i <= n - k ; i ++) { int currSum = 0 ; // compute sum of current subarray for (int j = 0 ; j < k ; j ++) { currSum += arr [i + j]; } // update maximum sum maxSum = Math . Max (maxSum , currSum); } return maxSum ; } static void Main () { int [] arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; Console . WriteLine (maxSubarraySum (arr , k)); } } JavaScript function maxSubarraySum (arr , k) { let n = arr . length ; let maxSum = 0 ; // check all subarrays of size k for (let i = 0 ; i <= n - k ; i ++) { let currSum = 0 ; // compute sum of current subarray for (let j = 0 ; j < k ; j ++) { currSum += arr [i + j]; } // update maximum sum maxSum = Math . max (maxSum , currSum); } return maxSum ; } // Driver Code let arr = [2 , 1 , 5 , 1 , 3 , 2]; let k = 3 ; console . log (maxSubarraySum (arr , k)); Output 9 [Better Approach - 1] Using Prefix Sum - O(n) Time and O(n) Space The idea is to precompute the prefix sum array where each element at index i stores the sum of elements from index 0 to i-1. Using this, we can compute the sum of any subarray in constant time O(1) using the difference of two prefix values. This eliminates the need to iterate over each subarray element repeatedly. C++ #include <iostream> #include <vector> using namespace std ; int maxSubarraySum (vector < int >& arr , int k) { int n = arr . size (); vector < int > prefix (n + 1 , 0); // build prefix sum array for (int i = 0 ; i < n ; i ++) { prefix [i + 1] = prefix [i] + arr [i]; } int maxSum = 0 ; // compute sum of each subarray of size k // using prefix array for (int i = 0 ; i <= n - k ; i ++) { int j = i + k - 1 ; int currSum = prefix [j + 1] - prefix [i]; // update maximum sum maxSum = max (maxSum , currSum); } return maxSum ; } int main () { vector < int > arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; cout << maxSubarraySum (arr , k) << endl ; return 0 ; } Java class GfG { static int maxSubarraySum (int [] arr , int k) { int n = arr . length ; int [] prefix = new int [n + 1] ; // build prefix sum array for (int i = 0 ; i < n ; i ++) { prefix [i + 1] = prefix [i] + arr [i] ; } int maxSum = 0 ; // compute sum of each subarray of size k // using prefix array for (int i = 0 ; i <= n - k ; i ++) { int j = i + k - 1 ; int currSum = prefix [j + 1] - prefix [i] ; // update maximum sum maxSum = Math . max (maxSum , currSum); } return maxSum ; } public static void main (String [] args) { int [] arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; System . out . println (maxSubarraySum (arr , k)); } } Python def maxSubarraySum (arr , k): n = len (arr) prefix = [0] * (n + 1) # build prefix sum array for i in range (n): prefix [i + 1] = prefix [i] + arr [i] maxSum = 0 # compute sum of each subarray of size k # using prefix array for i in range (n - k + 1): j = i + k - 1 currSum = prefix [j + 1] - prefix [i] # update maximum sum maxSum = max (maxSum , currSum) return maxSum if __name__ == "__main__" : arr = [2 , 1 , 5 , 1 , 3 , 2] k = 3 print (maxSubarraySum (arr , k)) C# using System ; class GfG { static int maxSubarraySum (int [] arr , int k) { int n = arr . Length ; int [] prefix = new int [n + 1]; // build prefix sum array for (int i = 0 ; i < n ; i ++) { prefix [i + 1] = prefix [i] + arr [i]; } int maxSum = 0 ; // compute sum of each subarray of size k // using prefix array for (int i = 0 ; i <= n - k ; i ++) { int j = i + k - 1 ; int currSum = prefix [j + 1] - prefix [i]; // update maximum sum maxSum = Math . Max (maxSum , currSum); } return maxSum ; } static void Main () { int [] arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; Console . WriteLine (maxSubarraySum (arr , k)); } } JavaScript function maxSubarraySum (arr , k) { const n = arr . length ; const prefix = new Array (n + 1). fill (0); // build prefix sum array for (let i = 0 ; i < n ; i ++) { prefix [i + 1] = prefix [i] + arr [i]; } let maxSum = 0 ; // compute sum of each subarray of size k // using prefix array for (let i = 0 ; i <= n - k ; i ++) { const j = i + k - 1 ; const currSum = prefix [j + 1] - prefix [i]; // update maximum sum maxSum = Math . max (maxSum , currSum); } return maxSum ; } // Driver Code const arr = [2 , 1 , 5 , 1 , 3 , 2]; const k = 3 ; console . log (maxSubarraySum (arr , k)); Output 9 [Better Approach - 2] Sliding Window using Queue - O(n) Time and O(k) Space The idea is to use a queue to maintain a window of the last k elements as we iterate through the array. We keep track of the current window sum by adding the new element to the sum and removing the oldest element (from the front of the queue) once the size exceeds k. At each step where the window size is exactly k, we update the maximum sum encountered so far. This ensures that we process each element exactly once and maintain the sliding window efficiently. C++ #include <iostream> #include <vector> #include <queue> using namespace std ; int maxSubarraySum (vector < int >& arr , int k) { int n = arr . size (); queue < int > q ; int sum = 0 , maxSum = 0 ; for (int i = 0 ; i < n ; i ++) { sum += arr [i]; q . push (arr [i]); // maintain window of size k if (q . size () > k) { sum -= q . front (); q . pop (); } // update maximum when window size becomes k if (q . size () == k) { maxSum = max (maxSum , sum); } } return maxSum ; } int main () { vector < int > arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; cout << maxSubarraySum (arr , k) << endl ; return 0 ; } Java import java.util.LinkedList ; import java.util.Queue ; class GfG { static int maxSubarraySum (int [] arr , int k) { int n = arr . length ; Queue < Integer > q = new LinkedList <> (); int sum = 0 , maxSum = 0 ; for (int i = 0 ; i < n ; i ++) { sum += arr [i] ; q . add (arr [i]); // maintain window of size k if (q . size () > k) { sum -= q . poll (); } // update maximum when window size becomes k if (q . size () == k) { maxSum = Math . max (maxSum , sum); } } return maxSum ; } public static void main (String [] args) { int [] arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; System . out . println (maxSubarraySum (arr , k)); } } Python from collections import deque def maxSubarraySum (arr , k): q = deque () sum = 0 maxSum = 0 for num in arr : sum += num q . append (num) # maintain window of size k if len (q) > k : sum -= q . popleft () # update maximum when window size becomes k if len (q) == k : maxSum = max (maxSum , sum) return maxSum if __name__ == "__main__" : arr = [2 , 1 , 5 , 1 , 3 , 2] k = 3 print (maxSubarraySum (arr , k)) C# using System ; using System.Collections.Generic ; class GfG { static int maxSubarraySum (int [] arr , int k) { Queue < int > q = new Queue < int > (); int sum = 0 , maxSum = 0 ; for (int i = 0 ; i < arr . Length ; i ++) { sum += arr [i]; q . Enqueue (arr [i]); // maintain window of size k if (q . Count > k) { sum -= q . Dequeue (); } // update maximum when window size becomes k if (q . Count == k) { maxSum = Math . Max (maxSum , sum); } } return maxSum ; } static void Main () { int [] arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; Console . WriteLine (maxSubarraySum (arr , k)); } } JavaScript function maxSubarraySum (arr , k) { let q = []; let sum = 0 ; let maxSum = 0 ; for (let i = 0 ; i < arr . length ; i ++) { sum += arr [i]; q . push (arr [i]); // maintain window of size k if (q . length > k) { sum -= q . shift (); } // update maximum when window size becomes k if (q . length === k) { maxSum = Math . max (maxSum , sum); } } return maxSum ; } // Driver Code let arr = [2 , 1 , 5 , 1 , 3 , 2]; let k = 3 ; console . log (maxSubarraySum (arr , k)); Output 9 [Expected Approach] Optimized Sliding Window - O(n) Time and O(1) Space The idea is to use a sliding window of size k to efficiently compute the maximum sum of any subarray of size k. First, calculate the sum of the initial window of size k. Then, slide the window by one element at a time: subtract the element that goes out of the window and add the new element that comes in. Step by Step Implementation: Calculate the sum of the first k elements and store it as currSum. Initialize maxSum = currSum. For each index i from k to n - 1, update the window by doing currSum = currSum + arr[i] - arr[i - k]. This adds the new element arr[i] and removes the element that slid out arr[i - k]. Update maxSum = max(maxSum, currSum) after each step. After the loop, return maxSum. C++ #include <iostream> #include <vector> using namespace std ; int maxSubarraySum (vector < int >& arr , int k) { int n = arr . size (); if (n < k) return -1 ; // compute sum of first window int windowSum = 0 ; for (int i = 0 ; i < k ; i ++) { windowSum += arr [i]; } int maxSum = windowSum ; // slide the window for (int i = k ; i < n ; i ++) { windowSum += arr [i] - arr [i - k]; maxSum = max (maxSum , windowSum); } return maxSum ; } int main () { vector < int > arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; cout << maxSubarraySum (arr , k) << endl ; return 0 ; } Java class GfG { static int maxSubarraySum (int [] arr , int k) { int n = arr . length ; if (n < k) return - 1 ; // compute sum of first window int windowSum = 0 ; for (int i = 0 ; i < k ; i ++) { windowSum += arr [i] ; } int maxSum = windowSum ; // slide the window for (int i = k ; i < n ; i ++) { windowSum += arr [i] - arr [i - k] ; maxSum = Math . max (maxSum , windowSum); } return maxSum ; } public static void main (String [] args) { int [] arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; System . out . println (maxSubarraySum (arr , k)); } } Python def maxSubarraySum (arr , k): n = len (arr) if n < k : return - 1 # compute sum of first window windowSum = sum (arr [: k]) maxSum = windowSum # slide the window for i in range (k , n): windowSum += arr [i] - arr [i - k] maxSum = max (maxSum , windowSum) return maxSum if __name__ == "__main__" : arr = [2 , 1 , 5 , 1 , 3 , 2] k = 3 print (maxSubarraySum (arr , k)) C# using System ; class GfG { static int maxSubarraySum (int [] arr , int k) { int n = arr . Length ; if (n < k) return - 1 ; // compute sum of first window int windowSum = 0 ; for (int i = 0 ; i < k ; i ++) { windowSum += arr [i]; } int maxSum = windowSum ; // slide the window for (int i = k ; i < n ; i ++) { windowSum += arr [i] - arr [i - k]; maxSum = Math . Max (maxSum , windowSum); } return maxSum ; } static void Main (string [] args) { int [] arr = { 2 , 1 , 5 , 1 , 3 , 2 }; int k = 3 ; Console . WriteLine (maxSubarraySum (arr , k)); } } JavaScript function maxSubarraySum (arr , k) { let n = arr . length ; if (n < k) return - 1 ; // compute sum of first window let windowSum = 0 ; for (let i = 0 ; i < k ; i ++) { windowSum += arr [i]; } let maxSum = windowSum ; // slide the window for (let i = k ; i < n ; i ++) { windowSum += arr [i] - arr [i - k]; maxSum = Math . max (maxSum , windowSum); } return maxSum ; } // Driver Code let arr = [2 , 1 , 5 , 1 , 3 , 2]; let k = 3 ; console . log (maxSubarraySum (arr , k)); Output 9 Comment Article Tags: Article Tags: DSA Arrays subarray sliding-window subarray-sum + 1 More