Sliding Window Maximum (Maximum of all subarrays of size K) - GeeksforGeeks

Source: https://www.geeksforgeeks.org/sliding-window-maximum-maximum-of-all-subarrays-of-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 Sliding Window Maximum (Maximum of all subarrays of size K) Last Updated : 13 Aug, 2025 Given an array arr[] of integers and an integer k , your task is to find the maximum value for each contiguous subarray of size k . The output should be an array of maximum values corresponding to each contiguous subarray. Examples : Input: arr[] = [1, 2, 3, 1, 4, 5, 2, 3, 6], k = 3 Output: [3, 3, 4, 5, 5, 5, 6] Explanation: 1st contiguous subarray = [1 2 3] max = 3 2nd contiguous subarray = [2 3 1] max = 3 3rd contiguous subarray = [3 1 4] max = 4 4th contiguous subarray = [1 4 5] max = 5 5th contiguous subarray = [4 5 2] max = 5 6th contiguous subarray = [5 2 3] max = 5 7th contiguous subarray = [2 3 6] max = 6 Input: arr[] = [5, 1, 3, 4, 2, 6], k = 1 Output: [5, 1, 3, 4, 2, 6] Explanation: When k = 1, each element in the array is its own subarray, so the output is simply the same array. Input: arr[] = [1, 3, 2, 1, 7, 3], k = 3 Output: [3, 3, 7, 7] Try it on GfG Practice Table of Content [Naive Approach] - Using Nested Loops - O(n * k) Time and O(1) Space [Better Approach] - Using Max-Heap - (n * log n) Time and O(n) Space [Expected Approach] - Using Deque - O(n) Time and O(k) Space [Naive Approach] - Using Nested Loops - O(n * k) Time and O(1) Space The idea is to run the nested loops, the outer loop will mark the starting point of the subarray of length k , and the inner loop will run from the starting index to index + k , and print the maximum element among these k elements. Below is the implementation of the above approach: C++ #include <iostream> #include <vector> using namespace std ; // Method to find the maximum for each // and every contiguous subarray of size k. vector < int > maxOfSubarrays (vector < int >& arr , int k) { int n = arr . size (); // to store the results vector < int > res ; for (int i = 0 ; i <= n - k ; i ++) { // Find maximum of subarray beginning // with arr[i] int max = arr [i]; for (int j = 1 ; j < k ; j ++) { if (arr [i + j] > max) max = arr [i + j]; } res . push_back (max); } return res ; } int main () { vector < int > arr = { 1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6 }; int k = 3 ; vector < int > res = maxOfSubarrays (arr , k); for (int maxVal : res) { cout << maxVal << " " ; } return 0 ; } Java import java.util.ArrayList ; import java.util.* ; class GfG { // Method to find the maximum for each // and every contiguous subarray of size k. static ArrayList < Integer > maxOfSubarrays (int [] arr , int k) { int n = arr . length ; // to store the results ArrayList < Integer > res = new ArrayList < Integer > (); for (int i = 0 ; i <= n - k ; i ++) { // Find maximum of subarray beginning // with arr[i] int max = arr [i] ; for (int j = 1 ; j < k ; j ++) { if (arr [i + j] > max) max = arr [i + j] ; } res . add (max); } return res ; } public static void main (String [] args) { int [] arr = { 1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6 }; int k = 3 ; ArrayList < Integer > res = maxOfSubarrays (arr , k); for (int maxVal : res) { System . out . print (maxVal + " "); } } } Python # Method to find the maximum for each # and every contiguous subarray of size k. def maxOfSubarrays (arr , k): n = len (arr) # to store the results res = [] for i in range (0 , n - k + 1): # Find maximum of subarray beginning # with arr[i] max = arr [i] for j in range (1 , k): if arr [i + j] > max : max = arr [i + j] res . append (max) return res if __name__ == "__main__" : arr = [1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6] k = 3 res = maxOfSubarrays (arr , k) for maxVal in res : print (maxVal , end = " ") C# using System ; using System.Collections.Generic ; class GfG { // Method to find the maximum for each // and every contiguous subarray of size k. static List < int > maxOfSubarrays (int [] arr , int k) { int n = arr . Length ; // to store the results List < int > res = new List < int > (); for (int i = 0 ; i <= n - k ; i ++) { // Find maximum of subarray beginning // with arr[i] int max = arr [i]; for (int j = 1 ; j < k ; j ++) { if (arr [i + j] > max) max = arr [i + j]; } res . Add (max); } return res ; } static void Main () { int [] arr = { 1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6 }; int k = 3 ; List < int > res =

maxOfSubarrays (arr , k); foreach (int maxVal in res) { Console . Write (maxVal + " "); } } } Javascript // Method to find the maximum for each // and every contiguous subarray of size k. function maxOfSubarrays (arr , k) { let n = arr . length ; // to store the results let res = []; for (let i = 0 ; i <= n - k ; i ++) { // Find maximum of subarray beginning // with arr[i] let max = arr [i]; for (let j = 1 ; j < k ; j ++) { if (arr [i + j] > max) max = arr [i + j]; } res . push (max); } return res ; } // Driver Code let arr = [1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6]; let k = 3 ; let res = maxOfSubarrays (arr , k); console . log (res . join (" ")); Output 3 3 4 5 5 5 6 Time Complexity: O(n * k), as we are using nested loops, where the outer loop runs n times, and for each iteration of outer loop, inner loop runs k times. Auxiliary Space: O(1) [Better Approach] - Using Max-Heap - (n * log n) Time and O(n) Space The idea is to use priority queue or heap data structure to make sure that heap has largest item of the current window. Create a max heap of the first k items Now iterate one by one. While the next item to be added is greater than the heap top, remove the top. We mainly make sure that the greater items of the previous window are not there in a heap. Below is the implementation of the above approach: C++ #include <bits/stdc++.h> using namespace std ; // Method to find the maximum for each // and every contiguous subarray of size k. vector < int > maxOfSubarrays (const vector < int >& arr , int k) { int n = arr . size (); // to store the results vector < int > res ; // to store the max value priority_queue < pair < int , int > > heap ; // Initialize the heap with the first k elements for (int i = 0 ; i < k ; i ++) heap . push ({ arr [i], i }); // The maximum element in the first window res . push_back (heap . top (). first); // Process the remaining elements for (int i = k ; i < arr . size (); i ++) { // Add the current element to the heap heap . push ({ arr [i], i }); // Remove elements that are outside the current // window while (heap . top (). second <= i - k) heap . pop (); // The maximum element in the current window res . push_back (heap . top (). first); } return res ; } int main () { vector < int > arr = { 1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6 }; int k = 3 ; vector < int > res = maxOfSubarrays (arr , k); for (int maxVal : res) { cout << maxVal << " " ; } return 0 ; } Java import java.util.ArrayList ; import java.util.Comparator ; import java.util.PriorityQueue ; class GfG { // Method to find the maximum for each // and every contiguous subarray of size k. static ArrayList < Integer > maxOfSubarrays (int [] arr , int k) { int n = arr . length ; // to store the results ArrayList < Integer > res = new ArrayList < Integer > (); // to store the max value PriorityQueue < Pair > heap = new PriorityQueue < Pair > (new Comparator < Pair > () { public int compare (Pair a , Pair b) { return b . first - a . first ; } }); // Initialize the heap with the first k elements for (int i = 0 ; i < k ; i ++) heap . add (new Pair (arr [i] , i)); // The maximum element in the first window res . add (heap . peek (). first); // Process the remaining elements for (int i = k ; i < arr . length ; i ++) { // Add the current element to the heap heap . add (new Pair (arr [i] , i)); // Remove elements that are outside the current // window while (heap . peek (). second <= i - k) heap . poll (); // The maximum element in the current window res . add (heap . peek (). first); } return res ; } static class Pair { int first ; int second ; Pair (int first , int second) { this . first = first ; this . second = second ; } } public static void main (String [] args) { int [] arr = { 1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6 }; int k = 3 ; ArrayList < Integer > res = maxOfSubarrays (arr , k); for (int maxVal : res) { System . out . print (maxVal + " "); } } } Python import heapq # Method to find the maximum for each # and every contiguous subarray of size k. def maxOfSubarrays (arr , k): n = len (arr) # to store the results res = [] # to store the max value heap = [] # Initialize the heap with the first k elements for i in range (0 , k): heapq . heappush (heap , (- arr [i], i)) # The maximum element in the first window res . append (- heap [0][0]) # Process the remaining elements for i in range (k , len (arr)): # Add the current element to the heap heapq . heappush (heap , (- arr [i], i)) # Remove elements that are outside the current # window while heap [0][1] <= i - k : heapq . heappop (heap) # The maximum element in the current window res . append (- heap [0][0]) return res if __name__ == "__main__" : arr = [1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6] k = 3 res = maxOfSubarrays (arr , k) for maxVal in res : print (maxVal , end = " ") C# using System ; using System.Collections.Generic ; class GfG { // Method to find the maximum for each // and every contiguous subarray of size k. static List < int > maxOfSubarrays (int [] arr , int k) { int n = arr . Length ; // to store the results List < int > res = new List < int > (); // to store the max value // Using SortedSet to simulate a max-heap SortedSet < Pair > heap = new SortedSet < Pair > (new PairComparer ()); // Initialize the heap with the first k elements for (int i = 0 ; i < k ; i ++) heap . Add (new Pair (arr [i], i)); // The maximum element in the first window res . Add (heap . Min . first); // Process the remaining elements for (int i = k ; i < arr . Length ; i ++) { // Add the current element to the heap heap . Add (new Pair (arr [i], i)); // Remove elements that are outside the current // window while (heap . Min . second <= i - k) heap . Remove (heap . Min); // The maximum element in the current window res . Add (heap . Min . first); } return res ; } class Pair { public int first ; public int second ; public Pair (int first , int second) { this . first = first ; this . second = second ; } } class PairComparer : IComparer < Pair > { public int Compare (Pair a , Pair b) { if (a . first != b . first) return

b . first . CompareTo (a . first); return a . second . CompareTo (b . second); } } static void Main () { int [] arr = { 1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6 }; int k = 3 ; List < int > res = maxOfSubarrays (arr , k); foreach (int maxVal in res) { Console . Write (maxVal + " "); } } } Javascript // Method to find the maximum for each // and every contiguous subarray of size k. function maxOfSubarrays (arr , k) { let n = arr . length ; // to store the results let res = []; // to store the max value let heap = []; // Function to sort the heap in descending order based on 'first' function sortHeap () { heap . sort (function (a , b) { if (a . first !== b . first) return b . first - a . first ; return a . second - b . second ; }); } // Initialize the heap with the first k elements for (let i = 0 ; i < k ; i ++) heap . push ({ first : arr [i], second : i }); sortHeap (); // The maximum element in the first window res . push (heap [0]. first); // Process the remaining elements for (let i = k ; i < arr . length ; i ++) { // Add the current element to the heap heap . push ({ first : arr [i], second : i }); sortHeap (); // Remove elements that are outside the current // window while (heap [0]. second <= i - k) { heap . shift (); sortHeap (); } // The maximum element in the current window res . push (heap [0]. first); } return res ; } // Driver Code let arr = [1 , 2 , 3 , 1 , 4 , 5 , 2 , 3 , 6]; let k = 3 ; let res = maxOfSubarrays (arr , k); console . log (res . join (" ")); Output 3 3 4 5 5 5 6 Time Complexity: O(n log n), where n is the size of the array. Inserting an element in heap takes (log n) time and we are inserting all n elements, thus the time complexity will be O(n * log n). Auxiliary Space: O(n), where n is the size of the array, this method requires O(n) space in the worst case when the input array is an increasing array [Expected Approach] - Using Deque - O(n) Time and O(k) Space Create a Deque , dq of capacity k , that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on right side of it in current window. Process all array elements one by one and maintain dq to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the dq is the largest and element at rear/back of dq is the smallest of current window. Illustration: Step-by-Step Algorithm : Create a deque to store only useful elements of current window. Run a loop and insert the first k elements in the deque. Before inserting the element, check if the element at the back of the queue is smaller than the current element, if it is so remove the element from the back of the deque until all elements left in the deque are greater than the current element. Then insert the current element, at the back of the deque. Now, run a loop from k to the end of the array. Print the front element of the deque. Remove the element from the front of the queue if they are out of the current window. Insert the next element in the deque. Before inserting the element, check if the element at the back of the queue is smaller than the current element, if it is so remove the element from the back of the deque until all elements left in the deque are greater than the current element. Then insert the current element, at the back of the deque. Print the maximum element of the last window. Below is the implementation of the above approach: C++ #include <bits/stdc++.h> using namespace std ; // Method to find the maximum for each // and every contiguous subarray of size k. vector < int > maxOfSubarrays (vector < int >& arr , int k) { // to store the results vector < int > res ; // create deque to store max values deque < int > dq (k); // Process first k (or first window) elements of array for (int i = 0 ; i < k ; ++ i) { // For every element, the previous smaller elements // are useless so remove them from dq while (! dq . empty () && arr [i] >= arr [dq . back ()]) { // Remove from rear dq . pop_back (); } // Add new element at rear of queue dq . push_back (i); } // Process rest of the elements, i.e., from arr[k] to arr[n-1] for (int i = k ; i < arr . size (); ++ i) { // The element at the front of the queue is the largest // element of previous window, so store it res . push_back (arr [dq . front ()]); // Remove the elements which are out of this window while (! dq . empty () && dq . front () <= i - k) { // Remove from front of queue dq . pop_front (); } // Remove all elements smaller than the currently being // added element (remove useless elements) while (! dq . empty () && arr [i] >= arr [dq . back ()]) { dq . pop_back (); } // Add current element at the rear of dq dq . push_back (i); } // store the maximum element of last window res . push_back (arr [dq . front ()]); return res ; } int main () { vector < int > arr = { 1 , 3 , 2 , 1 , 7 , 3 }; int k = 3 ; vector < int > res = maxOfSubarrays (arr , k); for (int maxVal : res) { cout << maxVal << " " ; } return 0 ; } Java import java.util.ArrayList ; import java.util.Deque ; import java.util.ArrayDeque ; class GfG { // Method to find the maximum for each // and every contiguous subarray of size k. static ArrayList < Integer > maxOfSubarrays (int [] arr , int k) { int n = arr . length ; // to store the results ArrayList < Integer > res = new ArrayList < Integer > (); // create deque to store max values Deque < Integer > dq = new ArrayDeque < Integer > (); // Process first k (or first window) elements of array for (int i = 0 ; i < k ; ++ i) { // For every element, the previous smaller elements // are useless so remove them from dq while (! dq . isEmpty () && arr [i] >= arr [dq . peekLast ()]) { // Remove from rear dq . pollLast (); } // Add new element at rear of queue dq . addLast (i); } // Process rest of the elements, i.e., from arr[k] to arr[n-1] for (int i = k ; i < arr . length ; ++ i) { // The element at the front of the queue is the largest // element of

previous window, so store it res . add (arr [dq . peekFirst ()]); // Remove the elements which are out of this window while (! dq . isEmpty () && dq . peekFirst () <= i - k) { // Remove from front of queue dq . pollFirst (); } // Remove all elements smaller than the currently being // added element (remove useless elements) while (! dq . isEmpty () && arr [i] >= arr [dq . peekLast ()]) { dq . pollLast (); } // Add current element at the rear of dq dq . addLast (i); } // store the maximum element of last window res . add (arr [dq . peekFirst ()]); return res ; } public static void main (String [] args) { int [] arr = { 1 , 3 , 2 , 1 , 7 , 3 }; int k = 3 ; ArrayList < Integer > res = maxOfSubarrays (arr , k); for (int maxVal : res) { System . out . print (maxVal + " "); } } } Python from collections import deque # Method to find the maximum for each # and every contiguous subarray of size k. def maxOfSubarrays (arr , k): n = len (arr) # to store the results res = [] # create deque to store max values dq = deque () # Process first k (or first window) elements of array for i in range (0 , k): # For every element, the previous smaller elements # are useless so remove them from dq while dq and arr [i] >= arr [dq [- 1]]: # Remove from rear dq . pop () # Add new element at rear of queue dq . append (i) # Process rest of the elements, i.e., from arr[k] to arr[n-1] for i in range (k , len (arr)): # The element at the front of the queue is the largest # element of previous window, so store it res . append (arr [dq [0]]) # Remove the elements which are out of this window while dq and dq [0] <= i - k : # Remove from front of queue dq . popleft () # Remove all elements smaller than the currently being # added element (remove useless elements) while dq and arr [i] >= arr [dq [- 1]]: dq . pop () # Add current element at the rear of dq dq . append (i) # store the maximum element of last window res . append (arr [dq [0]]) return res if __name__ == "__main__" : arr = [1 , 3 , 2 , 1 , 7 , 3] k = 3 res = maxOfSubarrays (arr , k) for maxVal in res : print (maxVal , end = " ") C# using System ; using System.Collections.Generic ; class GfG { // Method to find the maximum for each // and every contiguous subarray of size k. static List < int > maxOfSubarrays (int [] arr , int k) { int n = arr . Length ; // to store the results List < int > res = new List < int > (); // create deque to store max values LinkedList < int > dq = new LinkedList < int > (); // Process first k (or first window) elements of array for (int i = 0 ; i < k ; ++ i) { // For every element, the previous smaller elements // are useless so remove them from dq while (dq . Count > 0 && arr [i] >= arr [dq . Last . Value]) { // Remove from rear dq . RemoveLast (); } // Add new element at rear of queue dq . AddLast (i); } // Process rest of the elements, i.e., from arr[k] to arr[n-1] for (int i = k ; i < arr . Length ; ++ i) { // The element at the front of the queue is the largest // element of previous window, so store it res . Add (arr [dq . First . Value]); // Remove the elements which are out of this window while (dq . Count > 0 && dq . First . Value <= i - k) { // Remove from front of queue dq . RemoveFirst (); } // Remove all elements smaller than the currently being // added element (remove useless elements) while (dq . Count > 0 && arr [i] >= arr [dq . Last . Value]) { dq . RemoveLast (); } // Add current element at the rear of dq dq . AddLast (i); } // store the maximum element of last window res . Add (arr [dq . First . Value]); return res ; } static void Main () { int [] arr = { 1 , 3 , 2 , 1 , 7 , 3 }; int k = 3 ; List < int > res = maxOfSubarrays (arr , k); foreach (int maxVal in res) { Console . Write (maxVal + " "); } } } Javascript // Method to find the maximum for each // and every contiguous subarray of size k. function maxOfSubarrays (arr , k) { let n = arr . length ; // to store the results let res = []; // create deque to store max values let dq = []; // Process first k (or first window) elements of array for (let i = 0 ; i < k ; ++ i) { // For every element, the previous smaller elements // are useless so remove them from dq while (dq . length > 0 && arr [i] >= arr [dq [dq . length - 1]]) { // Remove from rear dq . pop (); } // Add new element at rear of queue dq . push (i); } // Process rest of the elements, i.e., from arr[k] to arr[n-1] for (let i = k ; i < arr . length ; ++ i) { // The element at the front of the queue is the largest // element of previous window, so store it res . push (arr [dq [0]]); // Remove the elements which are out of this window while (dq . length > 0 && dq [0] <= i - k) { // Remove from front of queue dq . shift (); } // Remove all elements smaller than the currently being // added element (remove useless elements) while (dq . length > 0 && arr [i] >= arr [dq [dq . length - 1]]) { dq . pop (); } // Add current element at the rear of dq dq . push (i); } // store the maximum element of last window res . push (arr [dq [0]]); return res ; } // Driver Code let arr = [1 , 3 , 2 , 1 , 7 , 3]; let k = 3 ; let res = maxOfSubarrays (arr , k); console . log (res . join (" ")); Output 3 3 7 7 Time Complexity: O(n), It can be observed that every element of the array is added and removed at most once. So there are a total of 2n operations. Auxiliary Space: O(k), Elements stored in the dequeue take O(k) space. Below is an extension of this problem: Sum of minimum and maximum elements of all subarrays of size k. Comment Article Tags: Article Tags: Misc Queue Heap DSA Arrays Amazon Flipkart Directi Zoho SAP Labs sliding-window + 7 More