Dash
All
Articles
Videos
Problems
Quiz
Contest
Prev
Next

Maximums of all subarrays of size K

Given an array and an integer **K**, find the maximum for each and every contiguous subarray of size K.

Examples :

> **Input:** arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}, K = 3
> **Output:** 3 3 4 5 5 5 6
> **Explanation:** Maximum of 1, 2, 3 is 3
> Maximum of 2, 3, 1 is 3
> Maximum of 3, 1, 4 is 4
> Maximum of 1, 4, 5 is 5
> Maximum of 4, 5, 2 is 5
> Maximum of 5, 2, 3 is 5
> Maximum of 2, 3, 6 is 6
>
> **Input**: arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}, K = 4
> **Output**: 10 10 10 15 15 90 90
> **Explanation:** Maximum of first 4 elements is 10, similarly for next 4 elements (i.e from index 1 to 4) is 10, So the sequence generated is 10 10 10 15 15 90 90

Naive Approach: To solve the problem using this approach follow the below idea:

> The idea is very basic run a nested loop, the outer loop which will mark the starting point of the subarray of length K, the inner loop will run from the starting index to index+K, and print the maximum element among these K elements.

Follow the given steps to solve the problem:

- Create a nested loop, the outer loop from starting index to N - Kth elements. The inner loop will run for K iterations.
- Create a variable to store the maximum of K elements traversed by the inner loop.
- Find the maximum of K elements traversed by the inner loop.
- Print the maximum element in every iteration of the outer loop

Below is the implementation of the above approach:

C++ Java

```
// Java program for the above approach

public class GFG {

    // Method to find the maximum for
    // each and every contiguous
    // subarray of size K.
    static void printKMax(int arr[], int N, int K)
    {
        int j, max;
```

```
        for (int i = 0; i <= N - K; i++) {

            max = arr[i];

            for (j = 1; j < K; j++) {
                if (arr[i + j] > max)
                    max = arr[i + j];
            }
            System.out.print(max + " ");
        }
    }

    // Driver's code
    public static void main(String args[])
    {
        int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int K = 3;

          // Function call
        printKMax(arr, arr.length, K);
    }
}
```

Output

```
3 4 5 6 7 8 9 10
```

Time Complexity: O(N * K), The outer loop runs N-K+1 times and the inner loop runs K times for every iteration of the outer loop. So time complexity is O((n-k+1)*k) which can also be written as O(N * K)

Auxiliary Space: O(1)

Maximum of all subarrays of size K using Deque:

> Create a Deque, *Qi* 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 *Qi* to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the *Qi* is the largest and element at rear/back of *Qi* is the smallest of current window.

Below is the dry run of the above approach:

Dash
All
Articles
Videos
Problems
Quiz
Contest

Initially :

	0	1	2	3	4	5	6	7	8
arr	1	2	3	1	4	5	2	3	6

, K = 3

Dequeue stores index of the maximum element at front and stores index of minimum element at back. At any time, it store indexes which belongs to current window

Step 1 :

Take the first window and keep necessary index in dequeue

	0	1	2	3	4	5	6	7	8
arr	1	2	3	1	4	5	2	3	6

dequeue : { 2 }
Maximum element is arr[2] = 3

Step 2 :

	0	1	2	3	4	5	6	7	8
arr	1	2	3	1	4	5	2	3	6

dequeue : { 2,3 }
Maximum element is arr[2] = 3

Step 3 :

	0	1	2	3	4	5	6	7	8
arr	1	2	3	1	4	5	2	3	6

dequeue : { 4 }
Maximum element is arr[4] = 4

Step 4 :

	0	1	2	3	4	5	6	7	8
arr	1	2	3	1	4	5	2	3	6

dequeue : { 5 }

Maximum element is arr[5] = 5

Step 5 :

dequeue : { 5,6 }
Maximum element is arr[5] = 5

Step 6 :

dequeue : { 5,7 }
Maximum element is arr[5] = 5

Step 7 :

dequeue : { 8 }
Maximum element is arr[8] = 6

Follow the given steps to solve the problem:

- Create a deque to store K elements.
- 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++ Java

```
// Java Program to find the maximum for
// each and every contiguous subarray of size K.
import java.util.Deque;
import java.util.LinkedList;

public class SlidingWindow {

    // A Dequeue (Double ended queue)
    // based method for printing
    // maximum element of
    // all subarrays of size K
    static void printMax(int arr[], int N, int K)
    {

        // Create a Double Ended Queue, Qi
        // that will store indexes of array elements
        // The queue will store indexes of
        // useful elements in every window and it will
        // maintain decreasing order of values
```

```
// from front to rear in Qi, i.e.,
// arr[Qi.front[]] to arr[Qi.rear()]
// are sorted in decreasing order
Deque<Integer> Qi = new LinkedList<Integer>();

/* Process first k (or first window)
elements of array */
int i;
for (i = 0; i < K; ++i) {

    // For every element, the previous
    // smaller elements are useless so
    // remove them from Qi
    while (!Qi.isEmpty()
           && arr[i] >= arr[Qi.peekLast()])

        // Remove from rear
        Qi.removeLast();

    // Add new element at rear of queue
    Qi.addLast(i);
}

// Process rest of the elements,
// i.e., from arr[k] to arr[n-1]
for (; i < N; ++i) {

    // The element at the front of the
    // queue is the largest element of
```

```
            // previous window, so print it
            System.out.print(arr[Qi.peek()] + " ");

            // Remove the elements which
            // are out of this window
            while ((!Qi.isEmpty()) && Qi.peek() <= i - K)
                Qi.removeFirst();

            // Remove all elements smaller
            // than the currently
            while ((!Qi.isEmpty())
                   && arr[i] >= arr[Qi.peekLast()])
                Qi.removeLast();

            // Add current element at the rear of Qi
            Qi.addLast(i);
        }

        // Print the maximum element of last window
        System.out.print(arr[Qi.peek()]);
    }

    // Driver's code
    public static void main(String[] args)
    {
        int arr[] = { 12, 1, 78, 90, 57, 89, 56 };
        int K = 3;
```

```
        // Function call
        printMax(arr, arr.length, K);
    }
}
```

Output

```
78 90 90 90 89
```

Time Complexity: O(N). It seems more than O(N) at first look. It can be observed that every element of the array is added and removed at most once. So there are total of 2n operations.
Auxiliary Space: O(K). Elements stored in the dequeue take O(K) space.

Dash
All
Articles
Videos
Problems
Quiz
Contest

Mark as Read

Report An Issue

If you are facing any issue on this page. Please let us know.