Maximum Sum Subarray of Size K (easy)

Problem Statement

Given an array of positive numbers and a positive number 'k', find the **maximum sum of any contiguous subarray of size 'k'**.

Example 1:

```
Input: [2, 1, 5, 1, 3, 2], k=3
Output: 9
Explanation: Subarray with maximum sum is [5, 1, 3].
```

Example 2:

```
Input: [2, 3, 4, 1, 5], k=2
Output: 7
Explanation: Subarray with maximum sum is [3, 4].
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

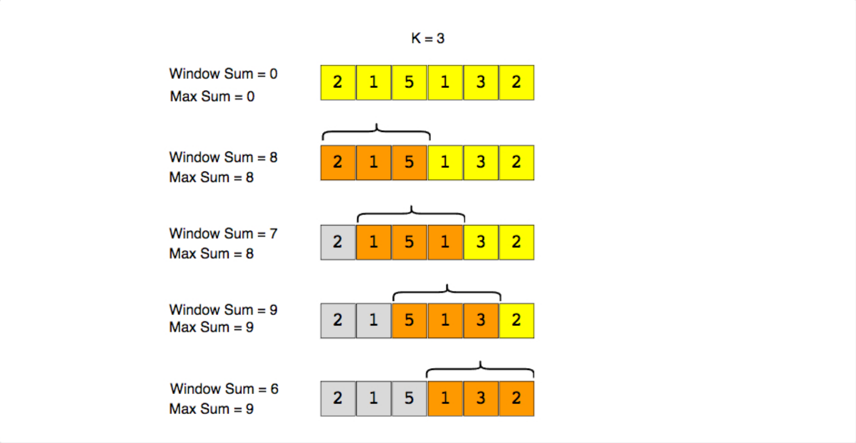
```
const max_sub_array_of_size_k = function(k, arr) {
  // TODO: Write your code here
  return -1;
};

```

TEST SAVE RESET

Solution

A basic brute force solution will be to calculate the sum of all 'k' sized subarrays of the given array, to find the subarray with the highest sum. We can start from every index of the given array and add the next 'k' elements to find the sum of the subarray. Following is the visual representation of this algorithm for Example-1:



Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
function max_sub_array_of_size_k(k, arr) {
  let maxSum = 0,
    windowSum = 0;

```

Happy Number (medium)
Middle of the LinkedList (easy)
Problem Challenge 1
Solution Review: Problem Challenge 1
Problem Challenge 2
Solution Review: Problem Challenge 2
Problem Challenge 3
Solution Review: Problem Challenge 3

Pattern: Merge Intervals

Introduction
Merge Intervals (medium)
Insert Interval (medium)
Intervals Intersection (medium)
Conflicting Appointments (medium)
Problem Challenge 1
Solution Review: Problem Challenge 1
Problem Challenge 2
Solution Review: Problem Challenge 2
Problem Challenge 3
Solution Review: Problem Challenge 3

Pattern: Cyclic Sort

Introduction
Cyclic Sort (easy)
Find the Missing Number (easy)
Find all Missing Numbers (easy)
Find the Duplicate Number (easy)
Find all Duplicate Numbers (easy)
Problem Challenge 1
Solution Review: Problem Challenge 1
Problem Challenge 2
Solution Review: Problem Challenge 2
Problem Challenge 3
Solution Review: Problem Challenge 3

Pattern: In-place Reversal of a LinkedList

Introduction
Reverse a LinkedList (easy)
Reverse a Sub-list (medium)
Reverse every K-element Sub-list (medium)
Problem Challenge 1
Solution Review: Problem Challenge 1
Problem Challenge 2
Solution Review: Problem Challenge 2

Pattern: Tree Breadth First Search

Introduction
Binary Tree Level Order Traversal

MW
Explore
Tracks
My Courses
Edpresso
Refer a Friend
Create

```
  for (i = 0; i < arr.length - k + 1; i++) {
    windowSum = 0;
    for (j = i; j < i + k; j++) {
      windowSum += arr[j];
    }
    maxSum = Math.max(maxSum, windowSum);
  }
  return maxSum;
}


console.log(`Maximum sum of a subarray of size K: ${max_sub_array_of_size_k(3, [2, 1, 5, 1, 3, 2])}`);
console.log(`Maximum sum of a subarray of size K: ${max_sub_array_of_size_k(2, [2, 3, 4, 1, 5])}`);
```

RUN SAVE RESET

The time complexity of the above algorithm will be $O(N * K)$, where 'N' is the total number of elements in the given array. Is it possible to find a better algorithm than this?

A better approach

If you observe closely, you will realize that to calculate the sum of a contiguous subarray we can utilize the sum of the previous subarray. For this, consider each subarray as a **Sliding Window** of size 'k'. To calculate the sum of the next subarray, we need to slide the window ahead by one element. So to slide the window forward and calculate the sum of the new position of the sliding window, we need to do two things:

1. Subtract the element going out of the sliding window i.e., subtract the first element of the window.
2. Add the new element getting included in the sliding window i.e., the element coming right after the end of the window.

This approach will save us from re-calculating the sum of the overlapping part of the sliding window. Here is what our algorithm will look like:

Java Python3 C++ JS

```
function max_sub_array_of_size_k(k, arr) {
  let maxSum = 0,
    windowSum = 0,
    windowStart = 0;

  for (window_end = 0; window_end < arr.length; window_end++) {
    windowSum += arr[window_end]; // add the next element
    // slide the window, we don't need to slide if we've not hit the required window size of 'k'
    if (window_end >= k - 1) {
      maxSum = Math.max(maxSum, windowSum);
      windowSum -= arr[windowStart]; // subtract the element going out
      windowStart += 1; // slide the window ahead
    }
  }
  return maxSum;
}


console.log(`Maximum sum of a subarray of size K: ${max_sub_array_of_size_k(3, [2, 1, 5, 1, 3, 2])}`);
console.log(`Maximum sum of a subarray of size K: ${max_sub_array_of_size_k(2, [2, 3, 4, 1, 5])}`);
```

RUN SAVE RESET

Time Complexity

The time complexity of the above algorithm will be $O(N)$.

Space Complexity

The algorithm runs in constant space $O(1)$.

✓ COMPLETED

← Back
Introduction

Next →
Smallest Subarray with a given sum (e...

Report an Issue