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++

```
using namespace std;

#include <iostream>
#include <vector>

class MaxSumSubArrayOfSizeK {
 public:
  static int findMaxSumSubArray(int k, const vector<int>& arr) {
    int maxSum = 0;
    // TODO: Write your code here
    return maxSum;
  }
};

```

Show Results Show Console

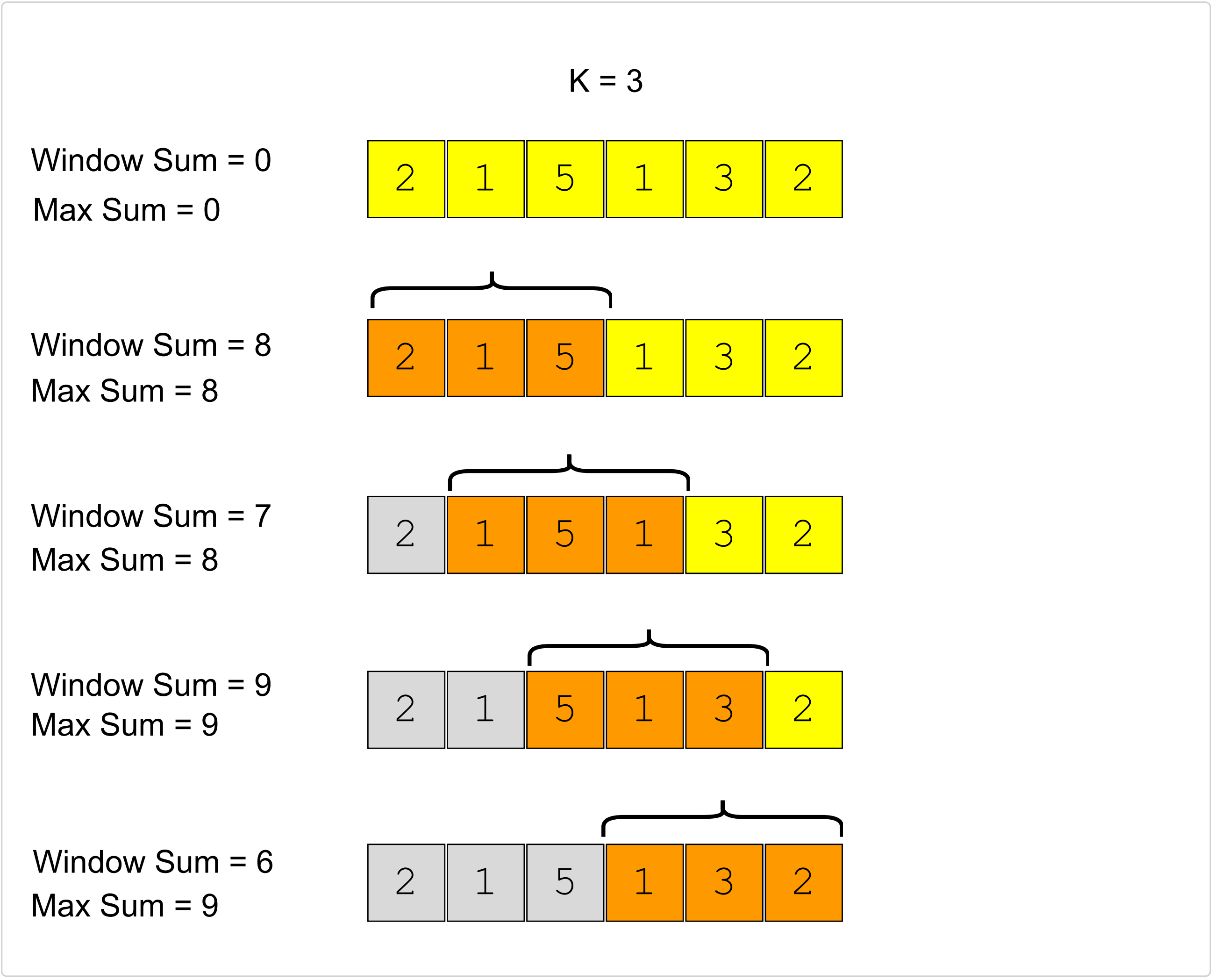
0 of 2 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✗	max_sub_array_of_size_k(3, [2, 1, 5, 1,	9	-1	Incorrect Output
✗	max_sub_array_of_size_k(2, [2, 3, 4, 1,	7	-1	Incorrect Output

0.496s

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

```cpp
using namespace std;

#include <iostream>
#include <vector>

class MaxSumSubArrayOfSizeK {
 public:
  static int findMaxSumSubArray(int k, const vector<int>& arr) {
    int maxSum = 0, windowSum;
    for (int i = 0; i <= arr.size() - k; i++) {
      windowSum = 0;
      for (int j = i; j < i + k; j++) {
        windowSum += arr[j];
      }
      maxSum = max(maxSum, windowSum);
    }

    return maxSum;
  }
};

int main(int argc, char* argv[]) {
  cout << "Maximum sum of a subarray of size K: "
       << MaxSumSubArrayOfSizeK::findMaxSumSubArray(3, vector<int>{2, 1, 5, 1, 3, 2}) << endl;
  cout << "Maximum sum of a subarray of size K: "
       << MaxSumSubArrayOfSizeK::findMaxSumSubArray(2, vector<int>{2, 3, 4, 1, 5}) << endl;
}
```

Output 0.857s

```
Maximum sum of a subarray of size K: 9
Maximum sum of a subarray of size K: 7
```

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

```cpp
using namespace std;

#include <iostream>
#include <vector>

class MaxSumSubArrayOfSizeK {
 public:
  static int findMaxSumSubArray(int k, const vector<int>& arr) {
    int windowSum = 0, maxSum = 0;
    int windowStart = 0;
    for (int windowEnd = 0; windowEnd < arr.size(); windowEnd++) {
      windowSum += arr[windowEnd];  // 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 (windowEnd >= k - 1) {
        maxSum = max(maxSum, windowSum);
        windowSum -= arr[windowStart];  // subtract the element going out
        windowStart++;                  // slide the window ahead
      }
    }

    return maxSum;
  }
};

int main(int argc, char* argv[]) {
  cout << "Maximum sum of a subarray of size K: "
       << MaxSumSubArrayOfSizeK::findMaxSumSubArray(3, vector<int>{2, 1, 5, 1, 3, 2}) << endl;
  cout << "Maximum sum of a subarray of size K: "
       << MaxSumSubArrayOfSizeK::findMaxSumSubArray(2, vector<int>{2, 3, 4, 1, 5}) << endl;
}
```

Output 1.338s

```
Maximum sum of a subarray of size K: 9
Maximum sum of a subarray of size K: 7
```

Time Complexity

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

Space Complexity

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