<https://leetcode.com/problems/maximum-points-you-can-obtain-from-cards/description/>

There are several cards **arranged in a row**, and each card has an associated number of points. The points are given in the integer array cardPoints.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly k cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array cardPoints and the integer k, return the **maximum score** you can obtain.

**Example 1:**

**Input:** cardPoints = [1,2,3,4,5,6,1], k = 3

**Output:** 12

**Explanation:** After the first step, your score will always be 1. However, choosing the rightmost card first will maximize your total score. The optimal strategy is to take the three cards on the right, giving a final score of 1 + 6 + 5 = 12.

**Example 2:**

**Input:** cardPoints = [2,2,2], k = 2

**Output:** 4

**Explanation:** Regardless of which two cards you take, your score will always be 4.

**Example 3:**

**Input:** cardPoints = [9,7,7,9,7,7,9], k = 7

**Output:** 55

**Explanation:** You have to take all the cards. Your score is the sum of points of all cards.

**Constraints:**

1 <= cardPoints.length <= 10^5

1 <= cardPoints[i] <= 10^4

1 <= k <= cardPoints.length

**Attempt 1: 2024-05-04**

**Solution 1: Fixed length Sliding Window (10 min)**

class Solution {

    public int maxScore(int[] cardPoints, int k) {

        int n = cardPoints.length;

        int total = 0;

        for(int cardPoint : cardPoints) {

            total += cardPoint;

        }

        int sum = 0;

        int i;

        for(i = 0; i < n - k; i++) {

            sum += cardPoints[i];

        }

        int curSum = sum;

        for(i = n - k; i < n; i++) {

            curSum += cardPoints[i] - cardPoints[i - n + k];

            sum = Math.min(sum, curSum);

        }

        return total - sum;

    }

}

**Refer to**

<https://leetcode.com/problems/maximum-points-you-can-obtain-from-cards/solutions/2198002/c-solution-using-sliding-window/>

**Intuition**

The idea here is pretty tricky to guess...

What we do is we initialize a window of size k...

Now, since we can select either from the start or from the end, we only have access to either the first k items or last k items, and hence we are trying to limit our access using this window...

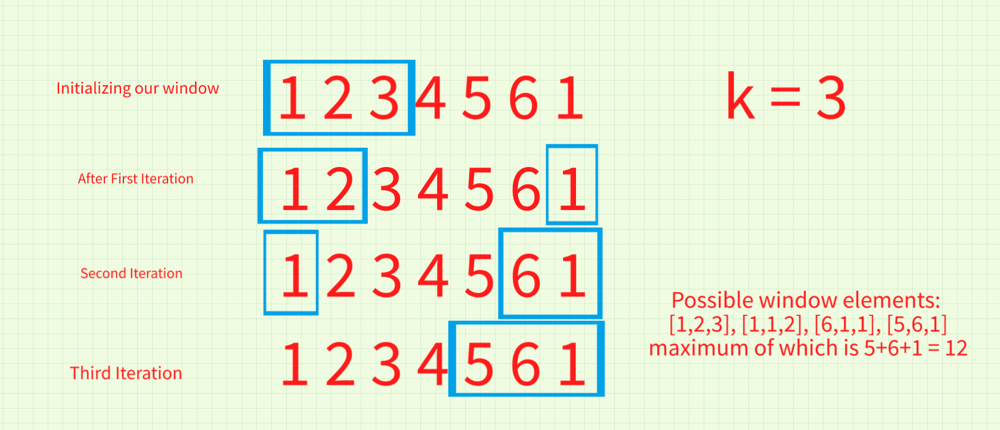
So, we include all the elements from start in our window, till its full...

The sum of elements at each instance in our window will be kept track of using another variable that will store our result.

Now, we remove the last element from our window, and add the last unvisited element of our cardPoints array... Similarly we keep on removing 1 element from our window and start adding the last unvisited element of our cardPoints array...

We keep doing this until we reach the start of our array, in this way we have covered all our possible picks...

**Example**



In our possible windows, we can see that we are covering all the possible picks in our given array using 3 cards...

class Solution {

public:

int maxScore(vector<int>& cardPoints, int k) {

int res = 0;

//First k elements in our window

for(int i=0;i<k;i++) res+=cardPoints[i];

int curr=res;

for(int i=k-1;i>=0;i--) {

//We remove the last visited element and add the non-visited element from the last

curr-=cardPoints[i];

curr+=cardPoints[cardPoints.size()-k+i];

//We check the maximum value any possible combination can give

res = max(res, curr);

}

return res;

}

};

**Refer to**

<https://leetcode.com/problems/maximum-points-you-can-obtain-from-cards/solutions/597763/python3-easy-sliding-window-o-n-find-minimum-subarray/>

**Problem Translation: Find the smallest subarray sum of length len(cardPoints) - k**

class Solution {

public int maxScore(int[] cardPoints, int k) {

int n = cardPoints.length - k;

int total = 0;

int min = Integer.MAX\_VALUE;

int cur = 0;

int l = 0;

int r = 0;

while (r < cardPoints.length) {

total += cardPoints[r];

cur += cardPoints[r];

if (r - l + 1 == n) {

min =Math.min(min, cur);

cur -= cardPoints[l++];

}

r++;

}

return total - (min == Integer.MAX\_VALUE ? 0 : min);

}

}

**Refer to**

[L1658.Minimum Operations to Reduce X to Zero (Ref.L1423)](note://WEB2d3951b594e0ba1abab46b9e91d0033c)