Maximum point you can obtain from cards

Problem Statement: Given N cards arranged in a row, each card has an associated score denoted by the cardScore array. Choose exactly k cards. In each step, a card can be chosen either from the beginning or the end of the row. The score is the sum of the scores of the chosen cards.

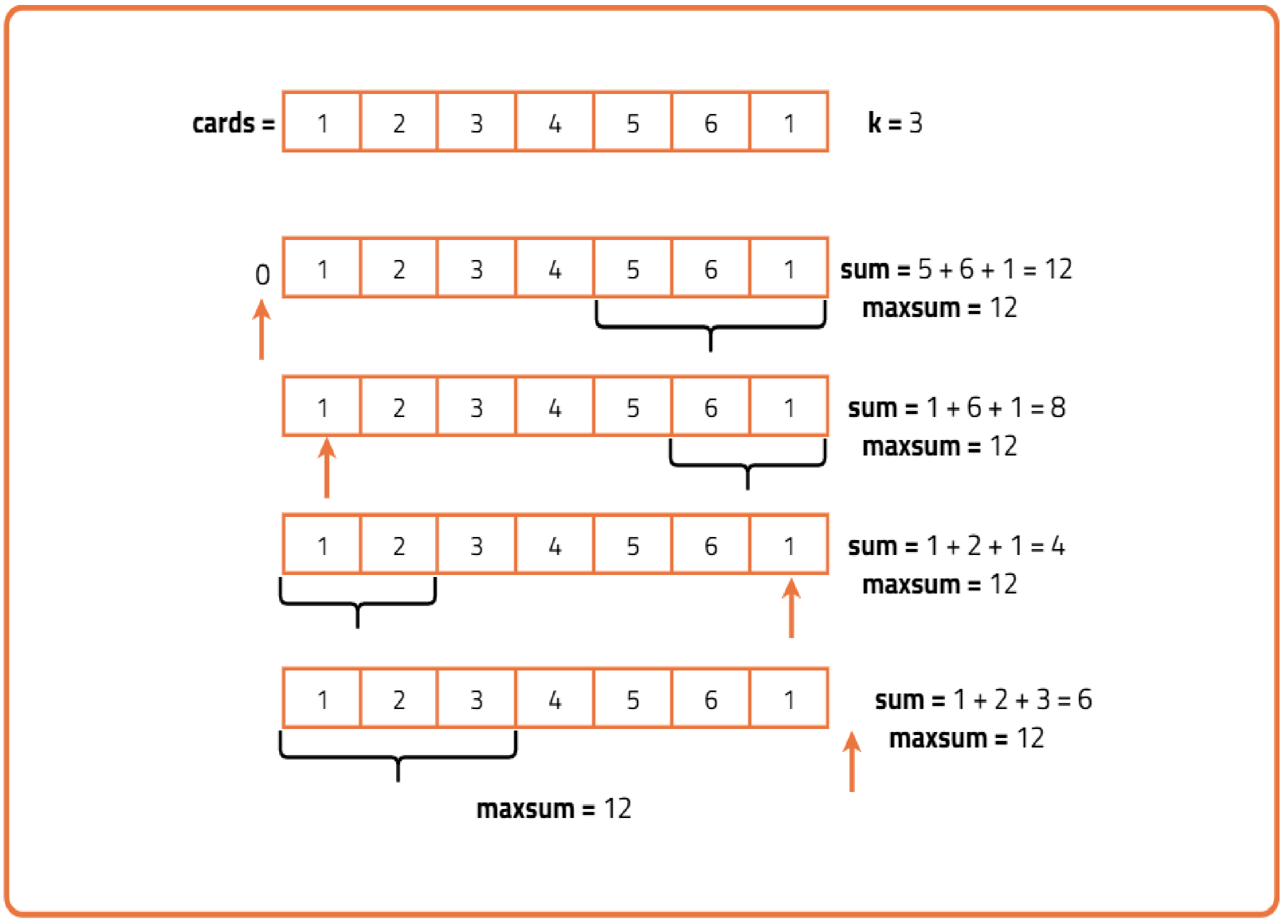
Examples

```
Input :cardScore = [1, 2, 3, 4, 5, 6] , k = 3
Output : 15
Explanation :Choosing the rightmost cards will maximize your total score.
So optimal cards chosen are the rightmost three cards 4 , 5 , 6.
Th score is 4 + 5 + 6 => 15.

Input :cardScore = [5, 4, 1, 8, 7, 1, 3 ] , k = 3
Output :12
Explanation : In first step we will choose card from beginning with score of 5.
In second step we will choose the card from beginning again with score of 4.
In third step we will choose the card from end with score of 3.
The total score is 5 + 4 + 3 => 12
```

Brute Force Approach
Algorithm
We can only take `k` cards, but from either end of the array. To find the maximum sum, we try all combinations of taking `i` cards from the start and `(k- i)` from the end, for all `i` from 0 to `k`. The best combination gives the answer.
Imagine a row of chocolates. You can only grab exactly `k`, and only from the left or right ends. To get the tastiest set, you try different combos maybe 2 from the left and 3 from the right and pick the one with the highest total sweetness.

- Calculate the total number of cards available.
- Initialize a variable to store the maximum score obtained so far.
- For each possible way to take cards from both ends (from 0 to k cards from the start):
    - Pick a certain number of cards from the beginning.
    - Pick the remaining number of cards from the end.
    - Calculate the sum of selected cards from both ends.
    - Update the maximum score if the current sum is greater.
- After checking all possible combinations, return the maximum score obtained.



Code

```
import java.util.*;

class Solution {
    // Function to calculate the maximum score by trying all front/back combos
    public int maxScore(int[] cardPoints, int k) {
        // Total number of cards
        int n = cardPoints.length;

        // Variable to store the max score found
        int maxSum = 0;

        // Try all combinations: i from front, k-i from back
        for (int i = 0; i <= k; i++) {
            // Store current sum for this combo
            int tempSum = 0;

            // Add first i cards from front
            for (int j = 0; j < i; j++) {
                tempSum += cardPoints[j];
            }

            // Add remaining cards from back
            for (int j = 0; j < k - i; j++) {
                tempSum += cardPoints[n - 1 - j];
            }

            // Update max score
            maxSum = Math.max(maxSum, tempSum);
        }
        // Return the highest score possible
        return maxSum;
    }
}

// Driver Code
public class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] cards = {1, 2, 3, 4, 5, 6, 1};
        int k = 3;
        System.out.println(sol.maxScore(cards, k));
    }
}
```

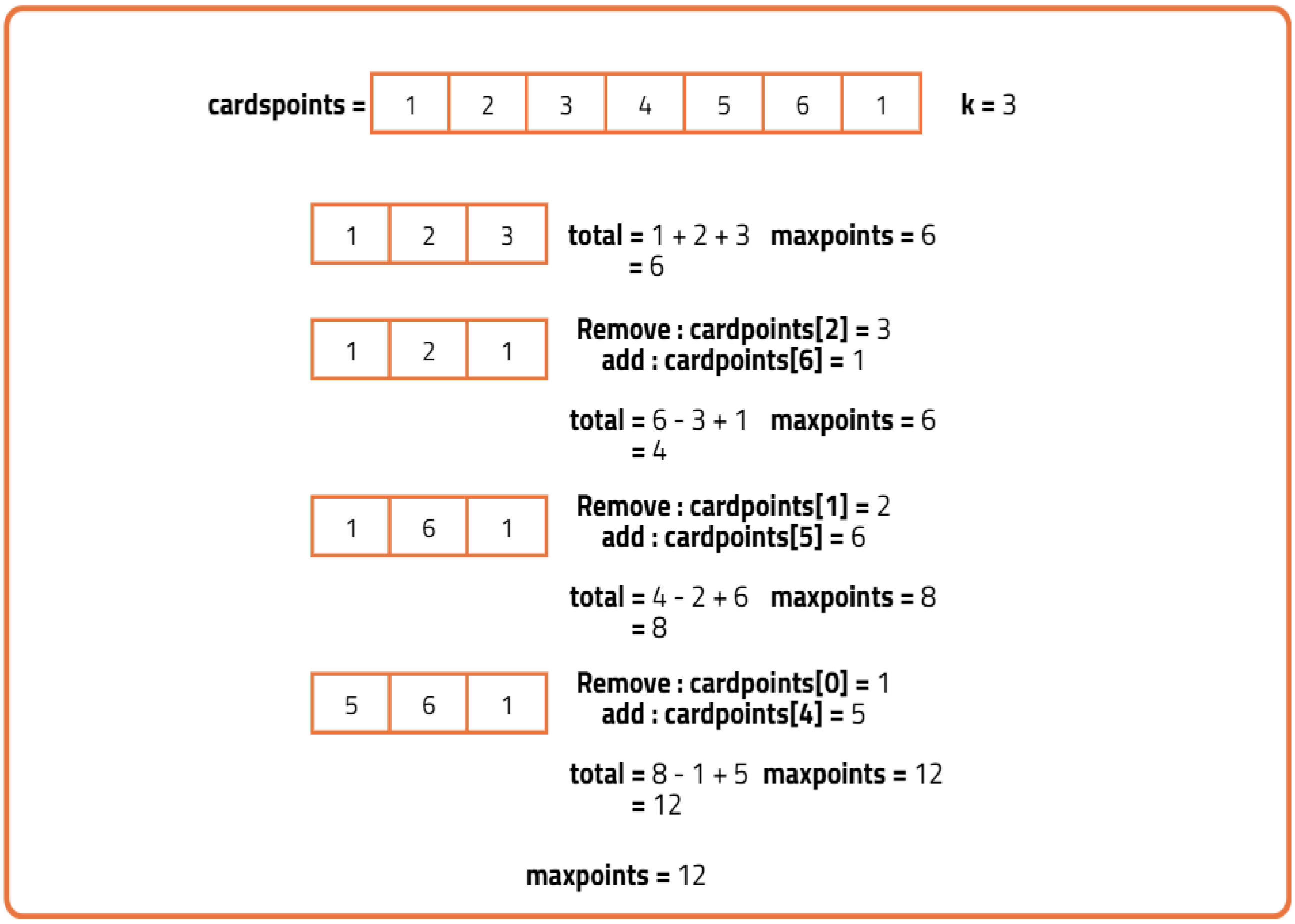
Complexity Analysis

Time Complexity: O(k),We try all combinations of taking cards from the front and back such that the total is exactly k cards. For each combination, we perform constant-time calculations, leading to a total of O(k) iterations.

Space Complexity: O(1),Only a fixed number of variables are used to store temporary sums and results, regardless of input size.

Optimal Approach
Algorithm
Instead of trying all combinations by recalculating front and back sums each time, we use a modified sliding window technique. We start by taking all `k` cards from the front, then gradually shift the window by removing one card from the front and adding one from the back. This keeps the number of selected cards fixed, but changes the selection balance between front and back. At each step, we update the score in constant time, which makes the solution much faster than brute force.

- Calculate the sum of the first few elements from the start of the array, equal to the total number of cards to be selected.
- Store this sum as the initial maximum possible score.
- Iterate from the end of this initial window, gradually removing one element from the end of the current front window and adding one new element from the back of the array.
- This maintains the total number of selected cards but shifts the balance between front and back.
- After each shift, compare the new total score with the previously stored maximum and update the maximum if the new score is higher.
- Repeat this process for as many shifts as there are cards to be picked.
- Return the highest score obtained after evaluating all possible combinations of selections from the front and back.



Code

```java
import java.util.*;
class Solution {
    // Function to return maximum score by picking k cards from either end
    public int maxScore(int[] cardPoints, int k) {
        // Get the total number of cards
        int n = cardPoints.length;

        // Calculate the sum of first k cards from the front
        int total = 0;
        for (int i = 0; i < k; i++) {
            total += cardPoints[i];
        }

        // Store the maximum score
        int maxPoints = total;

        // Slide the window: remove from front and add from back
        for (int i = 0; i < k; i++) {
            // Subtract card from front
            total -= cardPoints[k - 1 - i];

            // Add card from back
            total += cardPoints[n - 1 - i];

            // Update the max score
            maxPoints = Math.max(maxPoints, total);
        }

        // Return the best possible score
        return maxPoints;
    }
}

// Main class for testing
public class Main {
    // Driver code
    public static void main(String[] args) {
        // Define card points and k
        int[] cards = {1, 2, 3, 4, 5, 6, 1};
        int k = 3;

        // Create object of Solution
        Solution sol = new Solution();

        // Print the result
        System.out.println(sol.maxScore(cards, k));
    }
}
```

Complexity Analysis

Time Complexity: O(k) ,We calculate the initial sum of the first k cards , O(k) Then we slide the window k times,O(k) So overall: O(k + k) = O(k)

Space Complexity: O(1) , We only use a few variables (total, maxPoints, loop counters), no extra space used.