1176. Diet Plan Performance

Sliding Window

Problem Description

The caloric intake for each day is stored in an array where calories[i] represents the calories consumed on the i-th day. The person evaluates their diet every k consecutive days, assessing whether they have gained or lost points based on their total caloric intake over those days. If the total caloric intake is less than a lower limit, the person loses a point; if it's more than an upper limit, they gain a point; otherwise, their score does not change. The task is to calculate the total number of points the person has after they have finished their diet, which can be a positive or negative number.

In this problem, a person is tracking their diet over a number of days, where each day they consume a certain number of calories.

The solution is based on a <u>sliding window</u> technique where the window size is k. A sliding window is helpful in tracking a subset

sum of the first k days. Then, as the window moves forward by one day at a time, instead of recalculating the sum from scratch, the solution simply subtracts the calories from the day that is no longer in the window and adds the calories from the new day. This ensures that the sum is maintained for the current window efficiently.

After each shift of the window, the new sum is checked against the lower and upper bounds to determine if a point should be lost, gained, or if the score remains the same. This check is encapsulated in a function called check(s), which returns -1, 1, or 0,

of data in a larger dataset, in this case, the caloric intake over k consecutive days. The solution starts by calculating the initial

respectively. By summing these up, we can find the total score after all days have been accounted for.

Solution Approach

The provided Python solution uses an efficient <u>sliding window</u> approach to solve the diet plan performance problem. The sliding window concept is applicable when it is required to calculate something among all contiguous subarrays of a certain length in an

efficient manner. Here is a breakdown of the implementation steps:

1. **Initial Sum Calculation**: The initial sum s of the calories for the first k days is calculated using the built-in sum() function on the slice of the first k elements of the calories list.

Initial Points Calculation: The initial sum s is then passed to the check(s) function, which compares s with the lower and

count of the new day entering the window (calories[i]). This keeps the sum s up-to-date with the calorie count of the

Update Points: After updating the sum for the new window position, the check(s) function is used again to determine if

upper bounds. Depending on the comparison, the function returns -1, 1, or 0, representing losing a point, gaining a point, or no change in points, respectively. This initial points value is stored in the lans variable.

dieter has after completing the diet.

points are gained or lost, and ans = 0.

change in this example.

the diet are ans = 0.

Solution Implementation

def evaluate points(calories sum):

sliding window sum = sum(calories[:k])

score = evaluate points(sliding_window_sum)

if calories sum < lower:</pre>

return -1

Python

current window without having to re-calculate the sum from scratch.

(2000 and 3000, respectively), they neither gain nor lose a point: ans = 0.

Helper function to evaluate the points for the given sum of calories

If the sum is less than the lower bound, return -1 point

If the sum is greater than the upper bound, return 1 point

Calculate the initial sum of the first 'k' elements

Initialize score with the points from initial sum

- no change in points, respectively. This initial points value is stored in the ans variable.

 3. Sliding the Window: The code enters a loop that will iterate starting from the k-th day to the last day. For each iteration, it adjusts the sum by subtracting the calorie count of the day that's exiting the window (calories[i k]) and adding the
- points should be gained or lost based on the new sum. The result is added to the ans variable, which accumulates the total points.

 5. Returning the Result: After all windows have been processed, the function returns ans, which contains the total points the
- very efficient solution.

 The main tools used in the provided solution are the <u>sliding window</u> pattern and a helper function to calculate the point changes.

 The <u>calories</u> list serves as the main data structure, with simple arithmetic used to manage the sum and points.

This approach has a time complexity of O(n), where n is the number of days, and a space complexity of O(1), which makes it a

Let's consider a simple example where a person is tracking their calories over 5 days:

Calories = [1200, 1300, 1250, 1500, 1100], k = 2, lower = 2000, upper = 3000

1. Initial Sum Calculation: Start by summing the first k days (2 days in this case), so sum the first and second day's calories: s = 1200 + 1300 = 2500.

The person evaluates their diet every 2 consecutive days and loses or gains points if their caloric intake is below 2000 or above

2. Initial Points Calculation: Pass the initial sum s to the check(s) function. Since 2500 is between the lower and upper limits

3000, respectively.

Example Walkthrough

3. Sliding the Window: Now, for each day from the 3rd day to the 5th:

• Day 3 (New window: 1300, 1250): Adjust the sum to only include day 2 and 3 by subtracting day 1's calories and adding day 3's calories: s

= 2500 - 1200 + 1250 = 2550. After that, check the s value with check(s). Since 2550 is still between the lower and upper limits, no

check(s). Since 2750 is also between the limits, ans remains at 0.
 Day 5 (New window: 1500, 1100): Adjust the sum for day 4 and 5: s = 2750 - 1250 + 1100 = 2600. Check s with check(s). The sum is still between the limits, so the points stay the same, ans = 0.

Update Points: At each step of the sliding window, we updated the ans based on the check(s) function, which led to no

Returning the Result: The loop ends as there are no more windows to slide. The total points the dieter has after completing

○ Day 4 (New window: 1250, 1500): Adjust the sum to include the calories from day 3 and 4: s = 2550 - 1300 + 1500 = 2750. Pass s to

- The individual did not gain or lose any points during their diet according to the rules provided because their caloric intake stayed within the specified range for every assessment period.
- from typing import List

 class Solution:
 def dietPlanPerformance(self, calories: List[int], k: int, lower: int, upper: int) -> int:

elif calories_sum > upper:
 return 1
Otherwise, no points are awarded or deducted
else:
 return 0

Length of the calories list n = len(calories) # Iterate over the remaining elements, updating the sum and score

for i in range(k, n):

points--;

points++;

return points;

// Return the calculated points.

int totalDays = calories.size();

if (currentCalories < lower) {</pre>

} else if (currentCalories > upper) {

for (int i = k; i < totalDavs; ++i) {</pre>

if (currentCalories < lower) {</pre>

} else if (currentCalories > upper) {

--performanceScore;

++performanceScore;

} else if (windowSum > upper) {

int dietPlanPerformance(std::vector<int>& calories, int k, int lower, int upper) {

int currentCalories = std::accumulate(calories.begin(), calories.begin() + k, 0);

// Slide the 'k' day window through the remaining days and update the performance score.

// Add the calorie of the new day and subtract the calorie of the day exiting the 'k' day window.

// Calculate the total calories for the initial 'k' day period.

int performanceScore = 0; // Initialize the performance score.

currentCalories += calories[i] - calories[i - k];

return performanceScore; // Return the final performance score.

// Update the performance score based on the initial 'k' day period.

--performanceScore; // Decrease score when below lower limit.

++performanceScore; // Increase score when above upper limit.

// Update the score based on the new 'k' day period's calorie count.

```
# Update the sliding window sum by adding the current element and removing the oldest one
            sliding window sum += calories[i] - calories[i - k]
            # Update the score based on the updated sum
            score += evaluate_points(sliding_window_sum)
        # Return the total score after evaluating all sliding windows
        return score
Java
class Solution {
    public int dietPlanPerformance(int[] calories, int k, int lower, int upper) {
        // Initialize the sum of the first 'k' elements.
        int windowSum = 0;
        // Calculate the sum of the first 'k' calories.
        for (int i = 0; i < k; ++i) {
            windowSum += calories[i];
        // Initialize the performance points.
        int points = 0;
        // Check if the initial 'k' day period is below or above the threshold.
        if (windowSum < lower) {</pre>
            points--;
        } else if (windowSum > upper) {
            points++;
        // Iterate through the array starting from the 'k'th day.
        for (int i = k; i < calories.length; ++i) {</pre>
            // Slide the window by 1: remove the first element and add the new one.
            windowSum += calories[i] - calories[i - k];
            // Adjust points based on the new sum.
            if (windowSum < lower) {</pre>
```

};

C++

public:

#include <vector>

#include <numeric>

class Solution {

```
TypeScript
// Evaluates the diet plan performance by calculating points based on calorie intake over a sliding window.
// Parameters:
// calories: An array representing daily calorie intake.
// k: An integer representing the width of the sliding window of days.
// lower: An integer representing the lower threshold of calories.
// upper: An integer representing the upper threshold of calories.
// Returns:
// The total points calculated by comparing the total calories in each window against the lower and upper thresholds.
function dietPlanPerformance(calories: number[], k: number, lower: number, upper: number): number {
    const totalDays = calories.length; // Total number of days represented in the calories array.
    let windowSum = calories.slice(0, k).reduce((sum, current) => sum + current, 0); // Sum of first 'k' days.
    let points = 0; // Initialize points to judge the performance.
    // Award or deduct points based on the first 'k' days' total calories.
    if (windowSum < lower) {</pre>
        points--:
    } else if (windowSum > upper) {
        points++;
    // Slide the window one day at a time, updating the sum and points accordingly.
    for (let i = k; i < totalDays; ++i) {</pre>
        windowSum += calories[i] - calories[i - k]; // Update the window sum by adding the new day's calories and subtracting the fir
        // Award or deduct points based on the current window's calories.
        if (windowSum < lower) {</pre>
            points--;
        } else if (windowSum > upper) {
            points++;
    return points; // Return the calculated points.
```

```
n = len(calories)

# Iterate over the remaining elements, updating the sum and score
for i in range(k, n):
    # Update the sliding window sum by adding the current element and removing the oldest one
    sliding window sum += calories[i - calories[i - k]
    # Update the score based on the updated sum
    score += evaluate_points(sliding_window_sum)

# Return the total score after evaluating all sliding windows
return score

Time and Space Complexity

The provided code defines a function dietPlanPerformance that calculates a diet plan performance score based on calories
```

def dietPlanPerformance(self, calories: List[int], k: int, lower: int, upper: int) -> int:

Helper function to evaluate the points for the given sum of calories

If the sum is less than the lower bound, return -1 point

If the sum is greater than the upper bound, return 1 point

Otherwise, no points are awarded or deducted

Calculate the initial sum of the first 'k' elements

Initialize score with the points from initial sum

Summation of the first k elements is done in O(k) time. Following this initialization, the function iterates over the remaining elements, from k to n-1. For each iteration, it takes constant time to update the sum s (due to the subtraction of the oldest calorie value and the addition of the new calorie value) and to evaluate the check function. The

simplifies to O(n) for large n.

Space Complexity

Let's break down the operation:

Time Complexity

from typing import List

def evaluate points(calories sum):

elif calories_sum > upper:

sliding window sum = sum(calories[:k])

score = evaluate points(sliding_window_sum)

if calories sum < lower:</pre>

return -1

return 1

return 0

Length of the calories list

else:

class Solution:

the sum s (due to the subtraction of the oldest calorie value and the addition of the new calorie value) and to evaluate the check function. The loop, therefore, runs (n-k) times.

• Since k is a constant with respect to n, the time taken for initial summation is negligible for large n. Thus, the dominant term is (n-k), which

intake over a sliding window of size k, and compares each window's sum with given lower and upper bounds.

The time complexity of the code is O(n), where n is the total number of days (length of the calories array).

- 0(k) + 0(n-k) = 0(n)
 - Here's why:

The space complexity of the code is 0(1).

- The extra space used by the program is constant, as it only requires a fixed number of single-value variables (s, n, and ans) regardless of the input size.
 No additional data structures that scale with the input size are used; the input list itself is not modified.
- 0(1)

The check function is defined within the method and does not consume extra space that depends on the input size.