<https://leetcode.com/problems/k-radius-subarray-averages/description/>

You are given a **0-indexed** array nums of n integers, and an integer k.

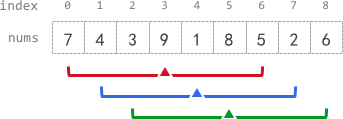
The **k-radius average** for a subarray of nums **centered** at some index i with the **radius** k is the average of **all** elements in nums between the indices i - k and i + k (**inclusive**). If there are less than k elements before or after the index i, then the **k-radius average** is -1.

Build and return an array avgs of length n where avgs[i] is the **k-radius average** for the subarray centered at index i.

The **average** of x elements is the sum of the x elements divided by x, using integer division. The **integer division** truncates toward zero, which means losing its fractional part.

For example, the average of four elements 2, 3, 1, and 5 is (2 + 3 + 1 + 5) / 4 = 11 / 4 = 2.75, which truncates to 2.

**Example 1:**



**Input:** nums = [7,4,3,9,1,8,5,2,6], k = 3

**Output:** [-1,-1,-1,5,4,4,-1,-1,-1]

**Explanation:**

- avg[0], avg[1], and avg[2] are -1 because there are less than k elements **before** each index.

- The sum of the subarray centered at index 3 with radius 3 is: 7 + 4 + 3 + 9 + 1 + 8 + 5 = 37.

Using **integer division**, avg[3] = 37 / 7 = 5.

- For the subarray centered at index 4, avg[4] = (4 + 3 + 9 + 1 + 8 + 5 + 2) / 7 = 4.

- For the subarray centered at index 5, avg[5] = (3 + 9 + 1 + 8 + 5 + 2 + 6) / 7 = 4.

- avg[6], avg[7], and avg[8] are -1 because there are less than k elements **after** each index.

**Example 2:**

**Input:** nums = [100000], k = 0

**Output:** [100000]

**Explanation:**

- The sum of the subarray centered at index 0 with radius 0 is: 100000. avg[0] = 100000 / 1 = 100000.

**Example 3:**

**Input:** nums = [8], k = 100000

**Output:** [-1]

**Explanation:**

- avg[0] is -1 because there are less than k elements before and after index 0.

**Constraints:**

n == nums.length

1 <= n <= 10^5

0 <= nums[i], k <= 10^5

**Attempt 1: 2024-12-25**

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

class Solution {

    public int[] getAverages(int[] nums, int k) {

        int windowSize = k \* 2 + 1;

        int n = nums.length;

        int[] result = new int[n];

        Arrays.fill(result, -1);

        // Must set as 'long' type in case maximum input

        long windowSum = 0;

        for(int i = 0; i < n; i++) {

            windowSum += nums[i];

            if(i >= windowSize - 1) {

                result[i - k] = (int) (windowSum / windowSize);

                windowSum -= nums[i - windowSize + 1];

            }

        }

        return result;

    }

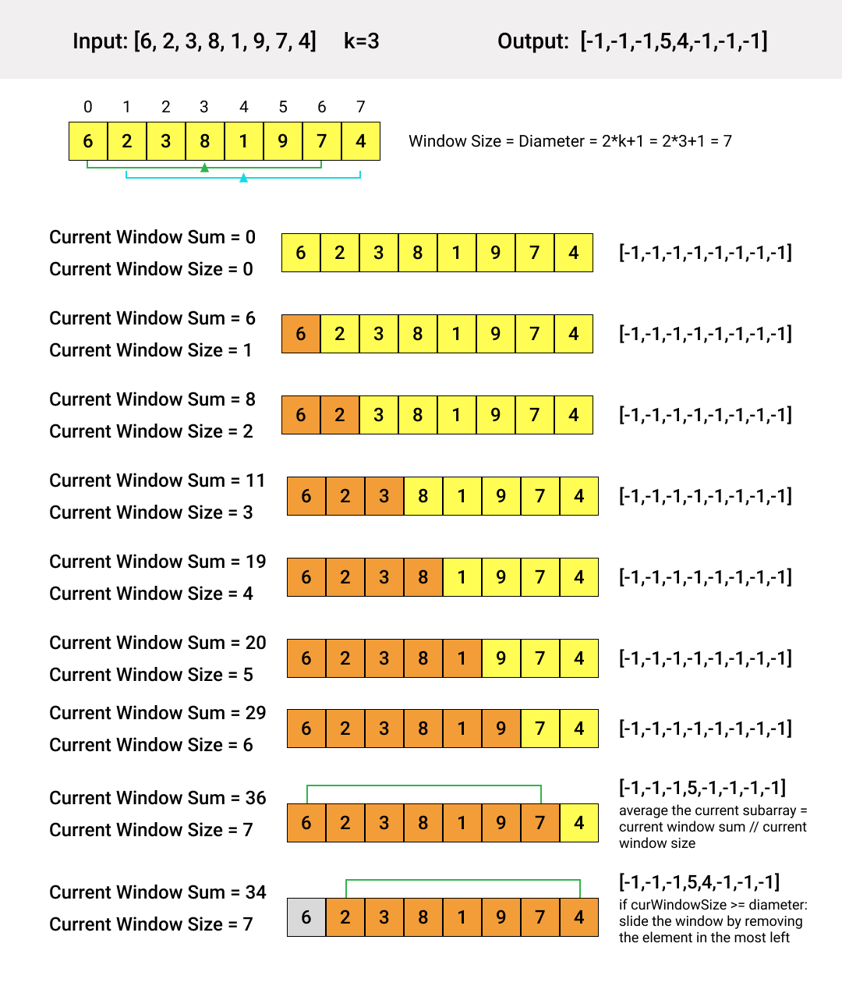
}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/k-radius-subarray-averages/solutions/1599973/python-3-sliding-window-illustration/>



class Solution:

def getAverages(self, nums: List[int], k: int) -> List[int]:

res = [-1]\*len(nums)

left, curWindowSum, diameter = 0, 0, 2\*k+1

for right in range(len(nums)):

curWindowSum += nums[right]

if (right-left+1 >= diameter):

res[left+k] = curWindowSum//diameter

curWindowSum -= nums[left]

left += 1

return res

Time complexity: O(n), where n is the length of the input array.

Space complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/2090>

**Problem Description**

In this problem, you are given an array nums containing n integers, indexed from 0 to n-1. You are also given an integer k, which defines a radius. The task is to calculate the *k-radius average* for each element in the array. The *k-radius average* for an element at index i includes all the elements from index i - k to i + k, inclusive. To qualify for calculating the average, there must be at least k elements before and after index i. If an element doesn't have enough neighbors to satisfy the radius k, the average for that element will be -1.

It is important to note that the average is computed using integer division, meaning you add all the elements within the range and then divide by the total number of elements, truncating the result to remove the decimal part.

For example, for the array [1,3,5,7,9] and k = 1, the result would be [-1, 3, 5, 7, -1] because:

elements at indices 0 and 4 don't have enough neighbors for radius 1;

element at index 1 has an average of (1+3+5)/3, which truncates to 3;

and so on for the elements at indices 2 and 3.

**Intuition**

The straightforward approach would be to calculate the average for each element individually, which would involve summing elements within the radius for every index, resulting in a time-consuming process with a time complexity of O(n\*k).

However, a more efficient method is to use a moving sum (or sliding window). By maintaining a running sum of the last 2k+1 elements, we can compute the k-radius average for the current index by simply dividing this sum by 2k+1. After computing the average, move the window by increasing the index i, add the next number in the array to the sum, and subtract the number that just left the window (which would be at index i - 2k). This way, we only need one pass through the array, resulting in a time complexity of O(n).

The provided Python function getAverages implements this efficient sliding window approach. An array ans of the same length as nums is initially filled with -1. This will store our results. The variable s is the sliding sum, which is updated as we iterate over nums with index i. If i is large enough (i >= k \* 2), it means we can compute an average for the element at index i - k. We add the current value v to s, and if the window is valid, we compute the average and update ans[i - k]. One important detail is the use of integer division s // (k \* 2 + 1) as required by the problem statement. Finally, we return the completed array ans.

**Solution Approach**

The solution provided uses a sliding window technique to efficiently compute each k-radius average in a single pass through the input array nums.

Here's an in-depth explanation of the algorithm's steps:

Initialize a running sum, s, which will keep track of the sum of elements in the current window. We will also initialize an array ans with the same length as nums filled with -1s, to store our results.

Iterate through the array nums with both index and value (i and v). Increment the running sum s by the value v.

Check if the current index i allows us to have a complete window of 2k+1 elements. This is determined by the condition i >= k \* 2. If i is less than k \* 2, we cannot calculate the average for i - k since we do not have enough elements before index i.

If we do have enough elements, calculate the average for the element at index i - k as the running sum s divided by 2k+1. We use integer division // here as specified by the problem constraints. The result is stored in ans[i - k].

Then, to maintain the size of the window, subtract the element at index i - 2k from the running sum s. This is the element that's falling out of our window as we move forward.

Continue this process until all elements have been visited, and the ans array is fully populated with the k-radius averages where possible or -1 where the average cannot be calculated.

The algorithm makes use of simple data structures which are a running sum variable s and an array ans to hold the results. The sliding window pattern here avoids redundantly recalculating the sum for overlapping parts of the window, thus optimizing the process to a time complexity of O(n) where n is the length of the input array.

Here's the core code snippet that illustrates the algorithm:

s = 0

ans = [-1] \* len(nums)

for i, v in enumerate(nums):

s += v

if i >= k \* 2:

ans[i - k] = s // (k \* 2 + 1)

s -= nums[i - k \* 2]

By maintaining the sliding window and updating the running sum in this manner, the algorithm efficiently computes the required averages without redundant calculations.

**Solution Implementation**

class Solution {

public int[] getAverages(int[] nums, int k) {

// Get the length of the input array

int n = nums.length;

// Create a new array for storing prefix sums with length of n + 1

long[] prefixSums = new long[n + 1];

// Compute the prefix sums

for (int i = 0; i < n; ++i) {

prefixSums[i + 1] = prefixSums[i] + nums[i];

}

// Initialize the answer array with -1, which signifies positions where

// the k-range average cannot be computed

int[] averages = new int[n];

Arrays.fill(averages, -1);

// Determine the averages for the k-range for each valid position

for (int i = 0; i < n; ++i) {

// Check if current index i allows a full k-range on both sides

if (i - k >= 0 && i + k < n) {

// Calculate the sum for this k-range

long sumForRange = prefixSums[i + k + 1] - prefixSums[i - k];

// Calculate the average and cast it to int before storing it in the result array

// (k << 1 | 1) calculates the size of the range, which is (2 \* k + 1)

averages[i] = (int) (sumForRange / (2 \* k + 1));

}

}

// Return the completed array with averages and -1 for non-computable positions

return averages;

}

}

**Time and Space Complexity**

The time complexity of the given code is O(n), where n is the length of the input list nums. This is because there is a single loop that iterates over all elements of nums once. Within the loop, operations are done in constant time including calculating the sum for the average and updating the sum by subtracting the element that is falling out of the sliding window.

The space complexity of the code is O(n), where n is the length of the input list nums. This is due to the ans list which is initialized to the same length as nums. There are no other data structures that depend on the size of the input that would increase the space complexity.

**Refer to**

[L560.Subarray Sum Equals K](note://5CE29DE019904808986311AFAC74C85C)