<https://leetcode.com/problems/maximum-subarray-sum-with-length-divisible-by-k/description/>

You are given an array of integers nums and an integer k.

Return the maximum sum of a subarray of nums, such that the size of the subarray is divisible by k.

**Example 1:**

**Input:** nums = [1,2], k = 1

**Output:** 3

**Explanation:**

The subarray [1, 2] with sum 3 has length equal to 2 which is divisible by 1.

**Example 2:**

**Input:** nums = [-1,-2,-3,-4,-5], k = 4

**Output:** -10

**Explanation:**

The maximum sum subarray is [-1, -2, -3, -4] which has length equal to 4 which is divisible by 4.

**Example 3:**

**Input:** nums = [-5,1,2,-3,4], k = 2

**Output:** 4

**Explanation:**

The maximum sum subarray is [1, 2, -3, 4] which has length equal to 4 which is divisible by 2.

**Constraints:**

1 <= k <= nums.length <= 2 \* 10^5

-10^9 <= nums[i] <= 10^9

**Attempt 1: 2025-01-18**

**Solution 1: Prefix Sum + Kadane's Algorithm (180 min)**

class Solution {

public long maxSubarraySum(int[] nums, int k) {

int len = nums.length; // Length of the input array `nums`.

long[] presum = new long[len + 1]; // Prefix sum array to store cumulative sums.

// Build the prefix sum array.

// `presum[i]` stores the sum of elements from `nums[0]` to `nums[i-1]`.

for (int i = 1; i <= len; i++) {

presum[i] = presum[i - 1] + nums[i - 1];

}

long result = Long.MIN\_VALUE; // Initialize the result to the smallest possible value.

// Iterate over all possible remainders (0 to k-1).

// This splits the array into disjoint groups based on indices.

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

int size = (len - i) / k; // Calculate how many subarrays of length `k` exist starting from index `i`.

long[] tmp = new long[size]; // Temporary array to store the sums of these subarrays.

// Construct the `tmp` array, which stores subarray sums of length `k`.

for (int j = 0; j < size; j++) {

int start = i + j \* k; // Start index of the subarray.

int end = start + k; // End index of the subarray.

tmp[j] = presum[end] - presum[start]; // Subarray sum using prefix sum difference.

}

// Use Kadane's algorithm to find the maximum subarray sum in `tmp`.

result = Math.max(result, kadaneHelper(tmp));

}

return result; // Return the maximum sum of a subarray whose size is divisible by `k`.

}

/\*\*

\* Kadane's Algorithm to find the maximum sum of a contiguous subarray.

\* @param tmp: Array of subarray sums to process.

\* @return Maximum sum of any contiguous subarray.

\*/

private long kadaneHelper(long[] tmp) {

long maxSum = Long.MIN\_VALUE; // Initialize the maximum sum to the smallest possible value.

long curSum = 0; // Tracks the sum of the current subarray.

// Iterate through the array and calculate the maximum subarray sum.

for (int i = 0; i < tmp.length; i++) {

// Update the current sum: either extend the current subarray or start a new one.

curSum = Math.max(tmp[i], curSum + tmp[i]);

// Update the maximum sum found so far.

maxSum = Math.max(maxSum, curSum);

}

return maxSum; // Return the maximum subarray sum.

}

/\*\*

\* Utility method to convert an `int[]` array to a `long[]` array.

\* Useful for cases where elements might exceed the range of `int`.

\* @param nums: Input array of integers.

\* @return A new array with the same values as `nums` but in `long` type.

\*/

private long[] convertToLong(int[] nums) {

long[] result = new long[nums.length]; // Allocate a new array of type `long` with the same size.

for (int i = 0; i < nums.length; i++) {

result[i] = nums[i]; // Copy each element from `nums` to the new `long` array.

}

return result; // Return the converted array.

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/maximum-subarray-sum-with-length-divisible-by-k/solutions/6124512/7ms-o-n-kadane-with-benefits/>

**Intuition**

Maximum subarray sum always immediately gives Kadane's algo vibes.

Let's see how to apply it here.

**Approach**

We can use Kadane's when we want to find max subarray sum among all of them. But in this question we want to process only limited number of arrays.

Let's decompose this task into K other tasks which we can solve with Kadane's.

First, if *k* = 1 - we simply use Kadane's algo.

What if *k* > 1?

**Let's build an array where each element will be valid (i.e. size is divisible by K). Let's start with *i* == 0. Then the next valid subarray is [*i*, *i* + *k*]. Next after that? [*i* + *k*, *i* + 2*k*]! This way we have an array where each subarray is of length *x* ∗ *k*.**

But we will lose some subarrays won't we? Yes, if we only consider *i* == 0 as a starting point. Let's just apply the same logic for all *i* within [0, *k*). This way we have covered all the subarrays matching the criteria.

Last thing is - to speed up we use prefix sums to calculate the helper array faster.

**Complexity**

Time complexity: *O*((*n*/*k*)∗*k*)=*O*(*n*)

Space complexity: *O*(*n*∗*k*)

class Solution {

public long maxSubarraySum(int[] nums, int k) {

if (k == 1) {

return kadane(nums);

}

long res = Long.MIN\_VALUE;

long[] prefix = new long[nums.length + 1];

for (int i = 1; i <= nums.length; i++) {

prefix[i] = prefix[i - 1] + nums[i - 1];

}

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

int len = (nums.length - i) / k;

long[] tmp = new long[len];

for (int j = 0; j < len; j++) {

int start = i + j \* k, end = start + k;

tmp[j] = prefix[end] - prefix[start];

}

res = Math.max(res, kadane(tmp));

}

return res;

}

static long kadane(long[] nums) {

long res = Long.MIN\_VALUE;

long curr = 0;

for (int i = 0; i < nums.length; i++) {

if (curr + nums[i] > nums[i]) {

curr += nums[i];

} else {

curr = nums[i];

}

if (curr > res) {

res = curr;

}

}

return res;

}

static long kadane(int[] nums) {

long res = Long.MIN\_VALUE;

long curr = 0;

for (int i = 0; i < nums.length; i++) {

if (curr + nums[i] > nums[i]) {

curr += nums[i];

} else {

curr = nums[i];

}

if (curr > res) {

res = curr;

}

}

return res;

}

}

**The above solution optimized by chatGPT**

class Solution {

    public long maxSubarraySum(int[] nums, int k) {

        // Special case: if k is 1, we can simply find the maximum subarray sum

        // of the entire array using Kadane's algorithm.

        if (k == 1) {

            return kadane(convertToLong(nums)); // Convert nums to long[] and call kadane

        }

        long res = Long.MIN\_VALUE; // Initialize the result to the smallest possible value

        long[] prefix = new long[nums.length + 1]; // Prefix sum array to calculate subarray sums

        // Compute the prefix sum for nums. Prefix[i] stores the sum of nums[0] to nums[i-1].

        for (int i = 1; i <= nums.length; i++) {

            prefix[i] = prefix[i - 1] + nums[i - 1];

        }

        // Iterate over all possible remainders (0 to k-1) to find subarrays with lengths divisible by k.

        for (int i = 0; i < k; i++) {

            // Calculate the number of subarrays of length divisible by k starting from remainder i.

            int len = (nums.length - i) / k;

            long[] tmp = new long[len]; // Temporary array to store subarray sums

            // Fill the tmp array with subarray sums of length k that start at index i, i+k, i+2k, ...

            for (int j = 0; j < len; j++) {

                int start = i + j \* k; // Start index of the subarray

                int end = start + k;  // End index of the subarray (exclusive)

                tmp[j] = prefix[end] - prefix[start]; // Subarray sum is the difference in prefix sums

            }

            // Use Kadane's algorithm to find the maximum sum of a contiguous subarray in tmp.

            res = Math.max(res, kadane(tmp));

        }

        return res; // Return the maximum subarray sum for subarrays with lengths divisible by k.

    }

    // Kadane's algorithm to find the maximum sum of a contiguous subarray in a long array.

    static long kadane(long[] nums) {

        long res = Long.MIN\_VALUE; // Initialize result to the smallest possible value

        long curr = 0;             // Current sum of the subarray being processed

        for (int i = 0; i < nums.length; i++) {

            // Either extend the current subarray or start a new subarray at nums[i].

            curr = Math.max(curr + nums[i], nums[i]);

            // Update the result with the maximum sum seen so far.

            res = Math.max(res, curr);

        }

        return res; // Return the maximum subarray sum.

    }

    // Helper function to convert an int array to a long array

    // This is necessary because Kadane's algorithm works on long arrays.

    static long[] convertToLong(int[] nums) {

        long[] result = new long[nums.length];

        for (int i = 0; i < nums.length; i++) {

            result[i] = nums[i];

        }

        return result;

    }

}

**Process Explanation**

**Problem Understanding:**

We are tasked with finding the **maximum sum of a subarray** of nums such that the **length of the subarray is divisible by k**.

A subarray's length is the count of elements in it, and its sum is the total of those elements.

**Solution Steps:**

Prefix Sum Array:

We compute a prefix sum array to enable efficient calculation of subarray sums.

prefix[i] contains the sum of the elements from nums[0] to nums[i-1].

Divide and Conquer:

For each remainder i in the range [0, k-1], consider subarrays of length k starting from index i.

Use a temporary array tmp to store the sums of these subarrays.

Kadane's Algorithm:

Once we have the subarray sums for the current remainder i in tmp, we use Kadane's algorithm to find the maximum sum of a contiguous subarray in tmp.

Result Calculation:

The result is the maximum sum obtained across all remainders.

**Why Kadane's Algorithm is Used:**

Kadane’s algorithm efficiently finds the maximum sum of a contiguous subarray in O(n), making it ideal for this problem.

**Time Complexity**

Prefix Sum Calculation:

O(n), where n is the length of nums.

Remainder Iteration:

O(k), as there are k remainders.

Kadane's Algorithm for Each Remainder:

O(n), as the total number of elements across all tmp arrays is nnn.

**Total Time Complexity:** O(n+kn)=O(n) (as k≤n).

**Refer to**

[L53.Maximum Subarray (Ref.L821,L918)](note://872ABD256AD34054B5D1F2E9992E2CAA)

[L918.Maximum Sum Circular Subarray (Ref.L53,L1658)](note://WEB1495ae8db0086f5ab30de99c6528274c)