<https://leetcode.com/problems/maximum-sum-of-distinct-subarrays-with-length-k/description/>

You are given an integer array nums and an integer k. Find the maximum subarray sum of all the subarrays of nums that meet the following conditions:

The length of the subarray is k, and

All the elements of the subarray are **distinct**.

Return the maximum subarray sum of all the subarrays that meet the conditions. If no subarray meets the conditions, return 0.

A **subarray** is a contiguous non-empty sequence of elements within an array.

**Example 1:**

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

**Output:** 15

**Explanation:** The subarrays of nums with length 3 are:

- [1,5,4] which meets the requirements and has a sum of 10.

- [5,4,2] which meets the requirements and has a sum of 11.

- [4,2,9] which meets the requirements and has a sum of 15.

- [2,9,9] which does not meet the requirements because the element 9 is repeated.

- [9,9,9] which does not meet the requirements because the element 9 is repeated.

We return 15 because it is the maximum subarray sum of all the subarrays that meet the conditions

**Example 2:**

**Input:** nums = [4,4,4], k = 3

**Output:** 0

**Explanation:** The subarrays of nums with length 3 are:

- [4,4,4] which does not meet the requirements because the element 4 is repeated.

We return 0 because no subarrays meet the conditions.

**Constraints:**

1 <= k <= nums.length <= 10^5

1 <= nums[i] <= 10^5

**Attempt 1: 2024-12-30**

**Solution 1: Fixed length Sliding Window using HashSet (120 min)**

**Wrong Solution 1: set.remove() more than required (82/93)**

**Test out by:**

**Input: nums = [9,9,9,1,2,3], k = 3, Output = 6, Expected = 12**

class Solution {

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

        long maxSum = 0;

        long curSum = 0;

        Set<Integer> set = new HashSet<>();

        int i = 0;

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

            curSum += nums[j];

            set.add(nums[j]);

            if(j - i + 1 > k) {

                set.remove(nums[i]);

                curSum -= nums[i];

                i++;

            }

            if(set.size() == k) {

                maxSum = Math.max(maxSum, curSum);

            }

        }

        return maxSum;

    }

}

**Wrong Solution 2: set.remove() less than required (73/93)**

**Test out by:**

**Input: nums = [1,5,4,2,9,9,9], k = 3, Output = 27, Expected = 15**

class Solution {

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

long maxSum = 0;

long curSum = 0;

Set<Integer> set = new HashSet<>();

int i = 0;

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

curSum += nums[j];

set.add(nums[j]);

if(j - i + 1 > k) {

if(set.size() > k) {

set.remove(nums[i]);

}

curSum -= nums[i];

i++;

}

if(set.size() == k) {

maxSum = Math.max(maxSum, curSum);

}

}

return maxSum;

}

}

**Correct Solution**

class Solution {

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

        long maxSum = 0;

        long curSum = 0;

        Set<Integer> set = new HashSet<>();

        int i = 0;

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

            curSum += nums[j];

            // We have to handle duplicates before normal window shrink

            // in other words, don't add duplicates initially into the

            // potential window

            // Test out by: nums = [9,9,9,1,2,3], k = 3

            while(set.contains(nums[j])) {

                curSum -= nums[i];

                set.remove(nums[i]);

                i++;

            }

            set.add(nums[j]);

            if(j - i + 1 > k) {

                set.remove(nums[i]);

                curSum -= nums[i];

                i++;

            }

            // We still need to check the set size

            // Test out by: nums = [4,4,4], k = 3

            if(set.size() == k) {

                maxSum = Math.max(maxSum, curSum);

            }

        }

        return maxSum;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/maximum-sum-of-distinct-subarrays-with-length-k/solutions/2783175/sliding-window-using-hashmap-c-java/comments/2023215>

class Solution {

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

Set<Integer> set = new HashSet<>();

long max = 0, sum = 0;

int windowStart = 0;

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

while (set.contains(nums[i]) || set.size() == k) {

set.remove(nums[windowStart]);

sum -= nums[windowStart++];

}

sum += nums[i];

set.add(nums[i]);

if (set.size() == k) {

max = Math.max(max, sum);

}

}

return max;

}

}

**Refer to chatGPT**

class Solution {

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

long maxSum = 0;

long currentSum = 0;

HashSet<Integer> uniqueElements = new HashSet<>();

int start = 0;

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

// Add the current number to the window

currentSum += nums[end];

// Handle duplicate elements by shrinking the window

while (uniqueElements.contains(nums[end])) {

uniqueElements.remove(nums[start]);

currentSum -= nums[start];

start++;

}

// Add the current element to the set

uniqueElements.add(nums[end]);

// If the window size equals `k`, evaluate the sum

if (end - start + 1 == k) {

maxSum = Math.max(maxSum, currentSum);

// Shrink the window from the left to prepare for the next iteration

uniqueElements.remove(nums[start]);

currentSum -= nums[start];

start++;

}

}

return maxSum;

}

}

**Explanation:**

Sliding Window:

Use a sliding window of size k to traverse the array while maintaining distinct elements.

HashSet for Uniqueness:

A HashSet is used to keep track of distinct elements in the current window.

If a duplicate is encountered, shrink the window from the left until the duplicate is removed.

Evaluate the Sum:

Once the window size becomes exactly k, calculate the sum and update maxSum if the current sum is greater.

Adjust the Window:

After calculating the sum for the current window of size k, shrink the window from the left to ensure the size remains consistent for the next iteration.

**Solution 2: Fixed length Sliding Window using HashMap (10 min)**

class Solution {

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

long maxSum = 0; // Variable to store the maximum subarray sum

long curSum = 0; // Variable to store the current subarray sum

Map<Integer, Integer> map = new HashMap<>(); // HashMap to track the frequency of elements in the current window

int i = 0; // Left pointer of the sliding window

// Iterate through the array using the right pointer 'j'

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

// Add the current element to the running sum

curSum += nums[j];

// Increment the frequency of the current element in the map

map.put(nums[j], map.getOrDefault(nums[j], 0) + 1);

// If the window size exceeds 'k', shrink the window from the left

if(j - i + 1 > k) {

// Decrease the frequency of the leftmost element in the map

map.put(nums[i], map.get(nums[i]) - 1);

// If the frequency becomes 0, remove the element from the map

if(map.get(nums[i]) == 0) {

map.remove(nums[i]);

}

// Subtract the leftmost element's value from the current sum

curSum -= nums[i];

// Move the left pointer to shrink the window

i++;

}

// If the window size is exactly 'k' at this moment (after window

// size confirmed as k), and all elements are distinct (map's size

// equal to k means k unique keys), check if the current sum is

// the maximum so far

if(map.size() == k) {

maxSum = Math.max(maxSum, curSum);

}

}

// Return the maximum subarray sum found

return maxSum;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Detailed Explanation**

Initialization:

maxSum keeps track of the maximum valid subarray sum found so far.

curSum maintains the sum of the current sliding window.

map is used to count the frequency of elements in the current window, allowing us to ensure all elements are distinct.

Sliding Window Logic:

Adding Elements (**j pointer):**

Add the current element to curSum.

Update the frequency of the current element in the map.

Shrinking the Window:

If the window size exceeds k, adjust the left boundary (i pointer):

Decrease the frequency of the leftmost element in the map.

If its frequency becomes 0, remove it from the map.

Subtract the leftmost element's value from curSum.

Increment the i pointer to shrink the window.

Validation and Maximization:

After ensuring the window size is exactly k, check if all elements are distinct (map.size() == k).

If valid, update maxSum with the maximum of its current value and curSum.

Time Complexity:

The for loop runs O(n), and each map operation (insert, remove, get) is O(1) on average.

Total time complexity: O(n).

Space Complexity:

The space complexity is O(k) for the map, as at most k distinct elements are stored in the map.

**Advantages of Using a HashMap:**

It provides an efficient way to count frequencies and ensure all elements in the window are distinct.

The logic is robust and handles duplicate elements correctly, ensuring only valid windows are considered.

This implementation is concise yet effective, leveraging a sliding window and HashMap to solve the problem efficiently.

**Refer to**

<https://leetcode.com/problems/maximum-sum-of-distinct-subarrays-with-length-k/solutions/2783099/map-sliding-window-w-dry-run/>

class Solution {

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

long sum = 0, max = 0;

int n = nums.length;

Map<Integer,Integer> map = new HashMap();

if(n == 1)

return nums[0];

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

map.put(nums[i], map.getOrDefault(nums[i],0) + 1);

sum += nums[i];

if(i>=k){

map.put(nums[i-k],map.get(nums[i-k]) - 1);

sum -= nums[i-k];

if(map.get(nums[i-k]) == 0)

map.remove(nums[i-k]);

}

if(map.size() == k)

max = Math.max(sum,max);

}

return max;

}

}

**Refer to**

[L1004.P2.7.Max Consecutive Ones III](note://B779B0B4DF164F39BF6D73423112A6BB)

[L2401.Longest Nice Subarray (Ref.L424,L2024)](note://WEB7538a2bbd5f8e4e2b3e803ba68408e0d)