2461. Maximum Sum of Distinct Subarrays With Length K

Sliding Window

Problem Description

<u>Array</u>

Hash Table

Medium

You are provided with an array nums consisting of integers and an integer k. The task is to locate the subarray (a contiguous nonempty sequence of elements within the array) with a length of exactly k elements such that all elements within the subarray are distinct. Once you identify such subarrays, you need to calculate their sums and determine the maximum sum that can be achieved under these constraints. If there are no subarrays that satisfy the conditions, you should return 0.

Intuition

Counter) to keep track of the distinct elements within the current window. The sliding window technique is a common approach for problems involving contiguous subarray (or substring) operations. Instead of generating all possible subarrays which would be computationally expensive, we can use a sliding window to move across nums with a fixed length of k. At each step, update the sum of the current window as well as the Counter, which represents

The intuition behind the solution is based on a sliding window technique coupled with the use of a hash map (in Python, a

the frequency of each number within the window. The sum is updated by adding the new element that comes into the window and subtracting the one that gets removed. If at any point, the size of the Counter is equal to k, it means all the elements in the window are distinct. This condition is checked after every move of the window one step forward. When this condition is met, we consider the current sum as a candidate for the

maximum sum. By maintaining a running sum of the elements in the current window and updating the Counter accordingly, we can efficiently determine the moment a subarray consisting of k distinct elements is formed and adjust the maximum sum if needed. The

solution, therefore, arrives at finding the maximum subarray sum with the given constraints through the sliding window technique and a Counter to track the distinct elements within each window. Solution Approach The solution follows a sliding window approach to maintain a subarray of length k and a Counter from the collections module to

Initialize a Counter with the first k elements of nums. This data structure will help efficiently track distinct elements by storing

their frequency (number of occurrences). Calculate the initial sum s of the first k elements. This serves as the starting subarray sum that we will update as we traverse

the nums array. Check if the initial Counter length is k, meaning all elements in the current window are distinct. If they are, assign that sum to

satisfies the given conditions.

track the frequency of elements within this subarray.

Here's a step-by-step explanation of how the implementation works:

Let's walk through a small example to illustrate the solution approach.

Calculate the initial sum s of these k elements: 4 + 2 + 4 = 10.

Suppose we have the following nums array and k value:

Iterate through the nums array starting at index k and for every new element added to the window: a. Increase the count of the new element (nums[i]) in the Counter and add its value to the running sum s. b. Decrease the count of the element that is no

longer within the window (nums [i-k]) and subtract its value from the running sum s. c. Remove the element from the Counter

ans. If not, ans is assigned a value of 0 to denote no valid subarray has been found yet.

if its frequency drops to zero, ensuring only elements actually in the current window are considered.

- After updating the Counter and the sum for the new window position, check again if the size of the Counter is k. If it is, check if the current sum s is greater than the previously recorded ans. If so, update ans with the new sum. Continue this process until the end of the array is reached. The maintained ans will be the maximum sum of a subarray that
- After completing the iteration, return the final value of ans. The algorithm ensures that at any point, only subarrays of length k with all distinct elements are considered for updating the
- all distinct elements within the defined length k.

recomputing the sum for each subarray from scratch. Thus, the implementation effectively finds the maximum subarray sum with

maximum sum. This is achieved using the Counter to check for distinct elements and a running sum mechanism to avoid

nums = [4, 2, 4, 5, 6]k = 3We want to find the subarray of length k where all elements are distinct and has the maximum sum. Let's apply our solution approach step by step:

We initialize a Counter with the first k elements of nums, which are [4, 2, 4]. The Counter will look like: {4: 2, 2: 1}.

The initial Counter length is not k (because we have only two distinct elements, and k is 3), so we do not have all distinct

element 5).

elements. We set ans to 0.

Since 15 > 11, we update ans to 15.

elements are distinct for the given k.

Calculate the sum of the first 'k' elements

max_sum = current_sum if len(num_counter) == k else 0

Add the new element to the counter and sum

Remove the (i-k)'th element from the counter and sum

If the number of unique elements equals 'k', assign sum to 'max_sum', else 0

Update 'max_sum' if the number of unique elements in the window equals 'k'

Iterate over the rest of the elements, starting from the 'k'th element

Solution Implementation

from collections import Counter

current_sum = sum(nums[:k])

for i in range(k, len(nums)):

current_sum += nums[i]

num_counter[nums[i]] += 1

if len(num_counter) == k:

// Return the maximum sum found

// Function to compute the maximum subarray sum with exactly k unique elements

std::unordered_map<int, int> count; // Map to store the frequency of elements

count[nums[i]]++; // Increment the frequency of the current element

// Initialize answer with the sum of the first window if it contains k unique elements

count[nums[i - k]]--; // Decrement frequency of the oldest element going out of the window

count[nums[i]]++; // Increment frequency of the new element in the window

// If the oldest element frequency reaches 0, remove it from the count map

using ll = long long; // Alias for long long to simplify the code

sum += nums[i]; // Add the current element to the sum

sum += nums[i]; // Add new element to current sum

sum -= nums[i - k]; // Subtract this element from current sum

// Update maxSum if current window contains k unique elements

long long maximumSubarraySum(std::vector<int>& nums, int k) {

ll sum = 0; // Initialize sum of the current subarray

int n = nums.size(); // Size of the input array

// Initialize the window of size k

ll maxSum = count.size() == k ? sum : 0;

if (count[nums[i - k]] == 0) {

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

count.erase(nums[i - k]);

// Slide the window across the array

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

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

return maxSum;

if num_counter[nums[i - k]] == 0:

del num_counter[nums[i - k]]

max_sum = max(max_sum, current_sum)

Python

Example Walkthrough

the Counter. The Counter now looks like: $\{2: 1, 5: 1, 6: 1\}$ with distinct counts and we have a new sum s = 11. As the size of the

○ We add the new element's value to the sum: s = 10 - 4 + 5 = 11 (we subtract the first element of the current window 4 and add the new

We decrease the count of the element that slid out of the window (4) in the Counter. Since its frequency drops to zero, it is removed from

• We increase the count of the new element (nums[3] = 5) in the Counter. So the Counter becomes {4: 1, 2: 1, 5: 1}.

 \circ Add nums [4] = 6 to the Counter, updating it to {2: 1, 5: 1, 6: 1} and sum s = 11 - 2 + 6 = 15. Now, each element's count is 1 and the Counter size is equal to k, which means all elements are distinct. We compare the sum s to ans.

Counter is equal to k, we compare s to ans. Since 11 > 0, we update ans = 11.

Next, we slide the window by one more element and repeat steps 4 and 5:

Now we start iterating from the index k in the nums array, which is nums [k] = nums[3] = 5.

Thus, the subarray [4, 5, 6] of length k = 3 gives us the maximum sum of 15 under the constraint that all elements within it are distinct.

After finishing the iteration, we end up with the final value of ans = 15, which is the maximum sum of a subarray where all

class Solution: def maximumSubarraySum(self, nums: List[int], k: int) -> int: # Initialize a counter for the first 'k' elements num counter = Counter(nums[:k])

num_counter[nums[i - k]] -= 1 current_sum -= nums[i - k] # If there's no more instances of the (i-k)'th element, remove it from the counter

```
# Return the maximum sum found that matches the unique count condition
       return max_sum
Java
class Solution {
   public long maximumSubarraySum(int[] nums, int k) {
       int n = nums.length; // Store the length of input array nums
       // Create a HashMap to count the occurrences of each number in a subarray of size k
       Map<Integer, Integer> countMap = new HashMap<>(k);
        long sum = 0; // Initialize sum of elements in the current subarray
       // Traverse the first subarray of size k and initialize the countMap and sum
       for (int i = 0; i < k; ++i) {
           countMap.merge(nums[i], 1, Integer::sum);
           sum += nums[i];
       // Initialize the answer with the sum of the first subarray if all elements are unique
        long maxSum = countMap.size() == k ? sum : 0;
       // Loop over the rest of the array, updating subarrays and checking for maximum sum
       for (int i = k; i < n; ++i) {
           // Add current element to the countMap and update the sum
           countMap.merge(nums[i], 1, Integer::sum);
           sum += nums[i];
           // Remove the element that's k positions behind the current one from countMap and update the sum
           int count = countMap.merge(nums[i - k], -1, Integer::sum);
           if (count == 0) {
               countMap.remove(nums[i - k]);
           sum = nums[i - k];
           // Update maxSum if the countMap indicates that we have a subarray with k unique elements
           if (countMap.size() == k) {
               maxSum = Math.max(maxSum, sum);
```

C++

public:

#include <vector>

class Solution {

#include <algorithm>

#include <unordered map>

```
maxSum = std::max(maxSum, sum);
          // Return the maximum subarray sum with exactly k unique elements
          return maxSum;
  };
  TypeScript
  function maximumSubarraySum(nums: number[], k: number): number {
      const n: number = nums.length;
      const countMap: Map<number, number> = new Map();
      let currentSum: number = 0;
      // Initialize the count map and current sum with the first 'k' elements
      for (let i: number = 0; i < k; ++i) {
          countMap.set(nums[i], (countMap.get(nums[i]) ?? 0) + 1);
          currentSum += nums[i];
      // Check if the first subarray of length 'k' has 'k' distinct numbers
      let maxSum: number = countMap.size === k ? currentSum : 0;
      // Traverse the array starting from the 'k'th element
      for (let i: number = k; i < n; ++i) {</pre>
          // Add the next number to the count map and current sum
          countMap.set(nums[i], (countMap.get(nums[i]) ?? 0) + 1);
          currentSum += nums[i];
          // Update the count map and current sum by removing the (i-k)'th number
          const prevCount: number = countMap.get(nums[i - k])! - 1;
          countMap.set(nums[i - k], prevCount);
          currentSum -= nums[i - k];
          // If after decrementing, the count is zero, remove it from the map
          if (prevCount === 0) {
              countMap.delete(nums[i - k]);
          // If the current subarray has 'k' distinct elements, update maxSum
          if (countMap.size === k) {
              maxSum = Math.max(maxSum, currentSum);
      // Return the maximum sum of a subarray with 'k' distinct numbers
      return maxSum;
from collections import Counter
class Solution:
   def maximumSubarraySum(self, nums: List[int], k: int) -> int:
       # Initialize a counter for the first 'k' elements
```

max_sum = max(max_sum, current_sum) # Return the maximum sum found that matches the unique count condition return max_sum

Time and Space Complexity

num_counter = Counter(nums[:k])

current_sum = sum(nums[:k])

for i in range(k, len(nums)):

current_sum += nums[i]

num_counter[nums[i]] += 1

num_counter[nums[i - k]] -= 1

if num_counter[nums[i - k]] == 0:

del num_counter[nums[i - k]]

current_sum -= nums[i - k]

if len(num counter) == k:

Calculate the sum of the first 'k' elements

max_sum = current_sum if len(num_counter) == k else 0

Add the new element to the counter and sum

Remove the (i-k)'th element from the counter and sum

amount of extra space used is proportional to the size of the window k.

If the number of unique elements equals 'k', assign sum to 'max_sum', else 0

If there's no more instances of the (i-k)'th element, remove it from the counter

Update 'max_sum' if the number of unique elements in the window equals 'k'

Iterate over the rest of the elements, starting from the 'k'th element

subarray of size k with unique elements. The code uses a sliding window approach by keeping a counter for the number of occurrences of each element within the current window of size k and computes the sum of elements in the window. **Time Complexity:**

The time complexity of the algorithm is <code>0(n)</code>, where n is the total number of elements in the input list nums. This is because the

code iterates through all the elements of nums once. For each element in the iteration, the time to update the cnt counter (Counter

The given Python code defines a method maximumSubarraySum within a class Solution to calculate the maximum sum of a

from the collections module) for the current window is constant on average due to the hash map data structure used internally. The operations inside the loop including incrementing, decrementing, deleting from the counter, and computing the sum are done in constant time. Since these operations are repeated for every element just once, it amounts to O(n).

Space Complexity: The space complexity of the algorithm is O(k). The cnt counter maintains the count of elements within a sliding window of size k. In the worst-case scenario, if all elements within the window are unique, the counter will hold k key-value pairs. Hence, the