2831. Find the Longest Equal Subarray

Binary Search Sliding Window

Problem Description

<u>Array</u>

Hash Table

remove elements when necessary.

In this problem, you're given an array of integers nums and an integer k. The task is to find the maximum length of a subarray where all elements are equal, after optionally deleting up to k elements from the array.

A subarray is defined as a contiguous part of the array which could be as small as an empty array or as large as the entire array. The concept of 'equality' here means that every item in the subarray is the same.

The challenge, therefore, is to figure out the strategy to remove up to k elements to maximize the length of this uniform subarray.

Intuition

Medium

The intuition behind the solution is to use a sliding window approach. The key insight is that the maximum length of equal subarray can be found by maintaining the count of elements within a given window and adjusting its size (using two pointers) to

To implement this, iterate over the array while keeping track of the frequency of each element in the current window using a counter. The variable mx is used to keep track of the maximum frequency of any element that we've seen in our current window. This represents the largest potential equal subarray if we were to remove other elements.

We can have a window that exceeds this maximum frequency by k elements, as we're allowed to delete at most k elements. Whenever the size of our current window exceeds mx + k, this implies that we need to remove some elements to bring it back

within the allowed size. We do this by shrinking the window from the left. Iteration continues until we've processed all elements, and the length of the largest possible equal subarray is returned.

This approach works because it continually adjusts the window size to accommodate the highest frequency element while

keeping count of the deletions within the limit of k. Whenever the limit is exceeded, the size of the window is reduced to restore the balance.

The provided solution code employs a sliding window technique along with a counter to efficiently keep track of the number of

Here's a step-by-step analysis of the implementation:

of the for loop.

the cnt counter.

Solution Approach

1. A counter named cnt is initialized using Counter from the collections module. This counter will keep track of the frequency of each element within the current window. 2. Two pointers, 1 (left) and r (right), are used to define the boundaries of the sliding window. 1 starts at 0, and r is incremented in each iteration

3. A variable mx, initialized at 0, is used to store the maximum frequency of any element within the current window throughout the iterations. 4. The main loop iterates over the indices and values from the nums array. As r moves to the right, the corresponding value x in nums is added to

occurrences of each element within the current window.

- 5. After each new element is added to cnt, the mx variable is updated to the maximum value between itself and the updated count for that element, effectively tracking the highest frequency element in the window.
- 6. At any point, if the size of the current window (given by r l + 1) minus the maximum frequency (mx) exceeds k, it indicates that we cannot delete enough elements to make the subarray equal. Thus, it's necessary to shrink the window from the left by incrementing 1 and
- decrementing the frequency of the 1th element in cnt. 7. This process continues until the end of the array is reached, ensuring that at each step, the window size exceeds mx + k by no more than one element.
- input array, and operations within the loop are O(1) on average due to the use of the counter.

This solution efficiently finds the longest equal subarray with a complexity of O(n), since it only requires a single pass over the

8. Finally, the length of the longest possible equal subarray (mx) that satisfies the condition after deleting at most k elements is returned.

The aim is to find the maximum length of a subarray where all elements are equal after optionally deleting up to 2 elements. Now, here's a step-by-step walkthrough using the sliding window approach:

Move to nums[1], which is also 1. Update cnt[1] to 2 and mx = 2.

and integer k = 2.

class Solution:

Solution Implementation

from collections import Counter

max_frequency = 0

element_count = Counter()

for right, element in enumerate(nums):

element_count[element] += 1

left += 1

return right - left + 1

Example Walkthrough

```
Next, we encounter nums [2] which is 2. Add to cnt and mx remains = 2.
```

At nums [4], the element is 4. Add to cnt and mx remains 2.

Let's consider the array nums = [1, 1, 2, 2, 4, 4, 4, 2] and k = 2.

Initialize the counter cnt and pointers l = 0, r = 0. Also, initialize mx as 0.

Start with the right pointer at the first element, i.e., nums [0] which is 1.

Continue to nums[3], another 2, update cnt and still mx = 2.

Continue the process for the rest of the elements, adjusting 1 and r accordingly and maintaining cnt and mx.

Now, the window size is r - l + 1 = 5 - 0 + 1 = 6, but mx + k = 2 + 2 = 4. Hence, we must shrink the window from the left.

- Increment 1 to point at nums [1], decrement the count of nums [0] which is 1, and window size is now 5, which is again greater
- At nums [6], the element is 4; we add it to cnt, resulting in cnt [4] being 3 and mx being updated to 3.

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

Increase the count of the current element

if right - left + 1 - max_frequency > k:

element_count[nums[left]] -= 1

Counter to store the frequency of elements in the current subarray

Iterate through the array using 'right' as the right pointer of the sliding window

Check if the window size minus the max frequency is greater than k

Move the left pointer of the window to the right

which implies that we cannot make all elements equal within k operations

Return the size of the largest window where all elements can be made equal using k operations

This works because the loop maintains the largest window possible while satisfying the k constraint

than mx + k. Therefore, increment 1 again.

considering at most k deletions will be retained in mx.

be 5, using the subarray [4, 4, 4, 2 (deleted), 2 (deleted)].

In the end, the implementation will thus return 5, the maximum length of an equal subarray, given the constraints, for array nums

For the given example, after moving through the entire array with the process mentioned, you would find the maximum length to

When the entire array has been checked with this approach, the maximum length of the possible equal subarray at any point

Python

Initialize the left pointer for the sliding window at position 0 left = 0# Variable to store the maximum frequency of any element in the current subarray

Reduce the count of the leftmost element since we are going to slide the window to the right

```
# Update the maximum frequency with the highest occurrence of any element so far
max_frequency = max(max_frequency, element_count[element])
```

C++

public:

};(

};

TypeScript

class Solution {

int longestEqualSubarray(vector<int>& nums, int k) {

// So we need to slide the window

--count[nums[left++]];

This is the corrected function in context:

int left = 0; // Left pointer for the sliding window

for (int right = 0; right < nums.size(); ++right) {</pre>

if (right - left + 1 - maxFrequency > k) {

maxFrequency = max(maxFrequency, ++count[nums[right]]);

```
Java
import java.util.HashMap;
import java.util.List;
import java.util.Map;
class Solution {
    /**
    * Finds the length of the longest subarray with at most k different numbers.
    * @param nums List of integers representing the array of numbers.
    * @param k The maximum number of different integers allowed in the subarray.
    * @return The length of longest subarray with at most k different numbers.
    */
    public int longestEqualSubarray(List<Integer> nums, int k) {
       // A map to store the count of each number in the current window
       Map<Integer, Integer> countMap = new HashMap<>();
       // maxFrequency stores the max frequency of a single number in the current window
       int maxFrequency = 0;
       // Initialize the left pointer of the window
       int left = 0;
       // Iterate over the array using the right pointer
       for (int right = 0; right < nums.size(); ++right) {</pre>
           // Increment the count of the rightmost number in the window
            countMap.merge(nums.get(right), 1, Integer::sum);
           // Update the max frequency if the current number's frequency is greater
           maxFrequency = Math.max(maxFrequency, countMap.get(nums.get(right)));
           // Check if the window is invalid, i.e.,
           // if the number of elements that are not the same as the most frequent one is greater than k
           if (right - left + 1 - maxFrequency > k) {
               // If invalid, move the left pointer to the right
               // and decrement the count of the number at the left pointer
                countMap.merge(nums.get(left), -1, Integer::sum);
                left++;
       // The window size is the length of the longest subarray with at most k different numbers
       return nums.size() - left;
```

```
· · · cpp
class Solution {
public:
    int longestEqualSubarray(vector<int>& nums, int k) {
        unordered_map<int, int> count; // Stores the frequency of each number encountered
        int maxFrequency = 0; // Keeps track of the maximum frequency of any number in the current window
        int left = 0; // Left pointer for the sliding window
        int maxLength = 0; // The length of the longest subarray
        for (int right = 0; right < nums.size(); ++right) {</pre>
            // Update the frequency of the current number and the max frequency in the window
```

// If the current window size minus the max frequency is greater than k

// it means we can't make the entire window equal by changing at most k elements

maxFrequency = max(maxFrequency, ++count[nums[right]]);

// Keep track of the maximum length of subarray found so far

return maxLength; // Return the length of the longest valid subarray found

// so we need to slide the window

--count[nums[left++]];

while (right - left + 1 - maxFrequency > k) {

maxLength = max(maxLength, right - left + 1);

function longestEqualSubarray(nums: number[], k: number): number {

Initialize the left pointer for the sliding window at position 0

max_frequency = max(max_frequency, element_count[element])

Move the left pointer of the window to the right

Check if the window size minus the max frequency is greater than k

which implies that we cannot make all elements equal within k operations

Variable to store the maximum frequency of any element in the current subarray

Iterate through the array using 'right' as the right pointer of the sliding window

Update the maximum frequency with the highest occurrence of any element so far

Return the size of the largest window where all elements can be made equal using k operations

This works because the loop maintains the largest window possible while satisfying the k constraint

Reduce the count of the leftmost element since we are going to slide the window to the right

// Create a map to store the count of each number in nums

unordered_map<int, int> count; // Stores the frequency of each number encountered

// If the current window size minus the max frequency is greater than k

int maxFrequency = 0; // Keeps track of the maximum frequency of any number in the current window

// Before moving the left pointer, decrease the frequency of the number going out of the window

There is an issue with the original code's `return` statement: `mx` is returned, but it seems that the goal of the function is to

// Before moving the left pointer, decrease the frequency of the number going out of the window

return right - left; // Here, 'right' is out of scope. This line should be at the end of the above for loop or right before

// Iterate through the nums array using 'right' as the right pointer of the sliding window

// Update the frequency of the current number and the max frequency in the window

// It means we can't make the entire window equal by changing at most k elements

// The size of the largest window we managed to create represents the longest subarray

// where we can make all elements equal by changing at most k elements

```
const countMap: Map<number, number> = new Map();
      // Variable to keep track of the maximum frequency of any number
      let maxFrequency = 0;
      // Left pointer for the sliding window
      let left = 0;
      // Iterate over the array with right as the right pointer of the sliding window
      for (let right = 0; right < nums.length; ++right) {</pre>
          // Increment the count of the current number by 1 or set it to 1 if it doesn't exist
          countMap.set(nums[right], (countMap.get(nums[right]) ?? 0) + 1);
          // Update the maximum frequency
          maxFrequency = Math.max(maxFrequency, countMap.get(nums[right])!);
          // If the window size minus max frequency is greater than k, shrink the window from the left
          if (right - left + 1 - maxFrequency > k) {
              // Decrement the count of the number at the left pointer
              countMap.set(nums[left], countMap.get(nums[left])! - 1);
              // Move the left pointer to the right
              left++;
      // The maximum size of the subarray is the size of the window when the loop completes
      return right - left;
from collections import Counter
class Solution:
   def longestEqualSubarray(self, nums: List[int], k: int) -> int:
       # Counter to store the frequency of elements in the current subarray
        element_count = Counter()
```

```
Time and Space Complexity
Time Complexity
```

return right - left + 1

left += 1

left = 0

max frequency = 0

for right, element in enumerate(nums):

element_count[element] += 1

Increase the count of the current element

if right - left + 1 - max_frequency > k:

element_count[nums[left]] -= 1

possibly decrementing a count in the Counter. However, as the Counter operations could in the worst case take O(n) when the Counter has grown to the size of the distinct elements in the array and we're decrementing, the dominant operation is still the for loop. Hence, the time complexity of the code is O(n), where n is the length of the nums array. **Space Complexity** Space complexity is influenced by the additional data structures used in the algorithm. The use of a Counter to store the

frequency of each distinct number results in a space complexity proportional to the number of distinct numbers in the nums array.

The time complexity of the given code is primarily determined by the for loop which iterates through each element of the nums

operations are performed in constant time, including updating the Counter, comparing and assigning the maximum value, and

array once. Therefore, the complexity is dependent on the number of elements n in the nums array. Within the loop, various

Thus, the space complexity is also 0(n), where n is the length of the nums array, representing the worst-case scenario where all elements are unique.

```
In the worst case scenario, all numbers are distinct, leading to a space complexity equal to the number of distinct elements,
which is O(n). However, if numbers repeat often, the space complexity could be much less.
```