2009. Minimum Number of Operations to Make Array Continuous

Hard Array Binary Search Leetcode Link

Problem Description

The problem presents an integer array `nums`. The goal is to make the given array `nums` continuous by performing a certain operation. The operation consists of replacing any element in `nums` with any integer. An array is defined as continuous if it satisfies the following two conditions:

1. All elements in the array are unique.
2. The difference between the maximum element and the minimum element in the array is equal to the length of the array minus one.

The requirement is to return the minimum number of operations needed to make the array `nums` continuous.

Intuition

To find the minimum number of operations to make the `nums` array continuous, we use a two-pointer approach. Here's the general intuition behind this approach:

1. **Removing Duplicates**: Since all numbers must be unique in a continuous array, we first remove duplicates by converting the array to a set and then back to a sorted list.

2. **Finding Subarrays with Potential**: We iterate through the sorted and deduplicated `nums` to look for subarrays that could potentially be continuous with minimal changes. Each subarray is characterized by a fixed starting point (`i`) and a dynamically found endpoint (`j`), where the difference between the maximum and minimum element (which is the first and last in the sorted subarray) is not greater than the length of the array minus one.

3. **Greedy Selection**: For each starting point `i`, we increment the endpoint `j` until the next element would break the continuity criterion. The size of the subarray between points `i` and `j` represents a potential continuous subarray.

4. **Calculating Operations**: For each of these subarrays, we calculate the number of operations needed by subtracting the number of elements in the subarray from the total number of elements in `nums`. The rationale is that elements not in the subarray would need to be replaced to make the entire array continuous.

5. **Finding the Minimum**: As we want the minimum number of operations, we track the smallest number of operations needed throughout the iteration by using the `min()` function, updating the `ans` variable accordingly.

The loop efficiently finds the largest subarray that can be made continuous without adding any additional elements (since adding elements is not an option as per problem constraints). The remaining elements—those not included in this largest subarray—are the ones that would need to be replaced. The count of these elements gives us the minimum operations required.

Solution Approach

The solution uses a sorted array without duplicates and a sliding window to find the minimum number of operations. The steps involved in the implementation are as follows:

1. **Sorting and Deduplication**: The input array `nums` is first converted to a set to remove any duplicates since our final array needs to have all unique elements. This set is then converted back into a sorted list to allow for easy identification of subarrays with potential to be continuous. This is important for step 2, the sliding window approach.

   ```
   nums = sorted(set(nums))
   ```

2. **Initial Variables**: The variable `n` stores the length of the original array. The variable `ans` is initialized to `n`, representing the worst-case scenario where all elements need to be replaced. We also initialize a variable `j` to 0, which will serve as our sliding window's endpoint.

3. **Sliding Window**: We then use a sliding window to find the largest subarray where the elements can remain unchanged. To do this, we iterate over the sorted array with a variable `i` that represents the starting point of our subarray.

   ```
   for i, v in enumerate(nums):
   ```

   Inside the loop, `j` is incremented until the condition `nums[j] - v <= n - 1` is no longer valid. This condition checks whether the subarray starting from `i` up to `j` can remain continuous if we were to fill in the numbers between `nums[i]` and `nums[j]`.

   ```
   while j < len(nums) and nums[j] - v <= n - 1:
       j += 1
   ```

4. **Calculating the Minimum**: For each valid subarray, we calculate the number of elements that need to be replaced:

   ```
   ans = min(ans, n - (j - i))
   ```

   This calculates how many elements are not included in the largest potential continuous subarray and takes the minimum of the current answer and the number of elements outside the subarray. The difference `n - (j - i)` gives us the number of operations needed to fill in the missing numbers, since we skipped over `n - (j - i)` numbers to achieve the length `n`.

By the end of the loop, `ans` contains the minimum number of operations required to make the array continuous, which is then returned.

This implementation efficiently solves the problem using a sorted set for uniqueness and a sliding window to find the best subarray. The selected subarray has the most elements that are already part of a hypothetical continuous array, thus minimizing the required operations.

Example Walkthrough

Let's take the array `nums = [4, 2, 5, 3, 5, 7, 6]` as an example to illustrate the solution approach.

1. **Sorting and Deduplication:**

   ```
   nums = sorted(set(nums)) // nums = [2, 3, 4, 5, 6, 7]
   ```

   We first remove the duplicate number 5 and then sort the array. The array becomes `[2, 3, 4, 5, 6, 7]`.

2. **Initial Variables:**

   ```
   n = len(nums) // n = 7 (original array length)
   ans = n // ans = 7
   j = 0
   ```

3. **Sliding Window:** We iterate through the sorted and deduplicated array using a sliding window technique. The sliding window starts at each element `i` in `nums` and we try to expand the window by increasing `j`.

   a. When `i = 0` (`nums[i] = 2`):

   ```
   nums[j] - nums[i] <= n - 1
   nums[0] - 2 <= 6 // 0 - 2 <= 6, condition is true, try next
   nums[1] - 2 <= 6 // 3 - 2 <= 6, condition is true, try next
   nums[2] - 2 <= 6 // 4 - 2 <= 6, condition is true, try next
   nums[3] - 2 <= 6 // 5 - 2 <= 6, condition is true, try next
   nums[4] - 2 <= 6 // 6 - 2 <= 6, condition is true, try next
   nums[5] - 2 <= 6 // 7 - 2 <= 6, condition is true
   ```

   At this point, the subarray `[2, 3, 4, 5, 6, 7]` is the largest we can get starting from `i = 0`, without needing addition.

   We calculate the operations needed for this subarray:

   ```
   ans = min(ans, n - (j - i)) // ans = min(7, 7 - (5 - 0)) = 2
   ```

   So, we need to replace 2 elements in the original array to make the subarray from `2 to 7` continuous.

   b. The loop continues for `i = 1` to `i = 5`, with the window size becoming smaller each time because the maximum possible value a continuous array can have also decreases.

By the end of the loop, we find that the minimum number of operations required is 2, which is the case when we consider the subarray `[2, 3, 4, 5, 6, 7]`. The two operations would involve replacing the two remaining numbers `4` and `5` (from the original `nums`) to get a continuous range that includes the largest possible number of the original elements.

Therefore, the answer for the example array is `2`. This demonstrates how the approach uses a sliding window to minimize the number of operations needed to make the array continuous.

Python Solution

```python
from typing import List

class Solution:
    def minOperations(self, nums: List[int]) -> int:
        # Get the length of the original nums list
        list_length = len(nums)

        # Use a set to eliminate duplicates, then convert back to a sorted list
        nums = sorted(set(nums))

        # Initialize the minimum number of operations as the length of the list
        min_ops = list_length

        # Initialize a pointer for the sliding window
        window_start = 0

        # Iterate through the list using the enumerate function, which provides both index and value
        for window_end, value in enumerate(nums):
            # Expand the window while the difference between the current value and the window's start value
            # is less than the length of the original list
            while window_start < len(nums) and nums[window_start] - value <= list_length - 1:
                window_start += 1

            # Update the minimum number of operations required by finding the minimum between
            # the current min_ops and the operations calculated using the size of the window.
            # The size of the window is the total number of elements that can be made consecutive by some operations.
            min_ops = min(min_ops, list_length - (window_start - window_end))

        # Return the minimum number of operations needed to have all integers in nums consecutively
        return min_ops

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    public int minOperations(int[] nums) {
        // Sort the array to bring duplicates together and ease the operation count process
        Arrays.sort(nums);

        // Start uniqueNumbers counter at 1 since the first number is always unique
        int uniqueNumbers = 1;

        // Step through the sorted array and remove duplicates
        for (int i = 1; i < nums.length; ++i) {
            if (nums[i] != nums[i - 1]) {
                nums[uniqueNumbers++] = nums[i];
            }
        }

        // Initialize variable to track the minimum number of operations
        int minOperations = nums.length;

        // Use a sliding window to count the number of operations
        for (int i = 0, j = 0; i < uniqueNumbers; ++i) {
            // Expand the window to the right as long as the condition is met
            while (j < uniqueNumbers && nums[j] - nums[i] <= nums.length - 1) {
                ++j;
            }
            // Calculate the minimum operations needed and store the result
            minOperations = Math.min(minOperations, nums.length - (j - i));
        }

        // Return the minimum number of operations found
        return minOperations;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Required for std::sort and std::unique

class Solution {
public:
    int minOperations(std::vector<int>& nums) {
        // Sort the vector in non-decreasing order
        std::sort(nums.begin(), nums.end());

        // Remove duplicate elements from the vector
        int uniqueCount = std::unique(nums.begin(), nums.end()) - nums.begin();

        // Store the total number of elements in the vector
        int totalCount = nums.size();

        // Initialize the answer to the max possible value, i.e., the total number of elements
        int minOperations = totalCount;

        // Use two pointers to find the least number of operations needed
        for (int left = 0, right = 0; left < uniqueCount; ++left) {
            // Move the right pointer as long as the difference between nums[right] and nums[left]
            // is less than or equal to the length of the array minus 1
            while (right < uniqueCount && nums[right] - nums[left] <= totalCount - 1) {
                ++right;
            }

            // Calculate the minimum operations needed by subtracting the length of the current
            // consecutive sequence from the total number of elements
            minOperations = std::min(minOperations, totalCount - (right - left));
        }

        // Return the minimum number of operations required
        return minOperations;
    }
};

```

Typescript Solution

```typescript
function minOperations(nums: number[]): number {
    // Sort the array in non-decreasing order
    nums.sort((a, b) => a - b);

    // Remove duplicate elements from the array and get the count of unique elements
    const uniqueElements: number[] = Array.from(new Set(nums));
    const uniqueCount: number = uniqueElements.length;

    // Store the total number of elements in the array
    const totalCount: number = nums.length;

    // Initialize the answer to the max possible value, i.e., the total number of elements
    let minOps: number = totalCount;

    // Use two pointers to find the least number of operations needed
    for (let left = 0, right = 0; left < uniqueCount; ++left) {

        // Move the right pointer as long as the difference between the unique elements at 'right' and 'left'
        // is less than or equal to the length of the array minus 1
        while (right < uniqueCount && uniqueElements[right] - uniqueElements[left] <= totalCount - 1) {
            ++right;
        }

        // Calculate the minimum operations needed by subtracting the length of the current
        // consecutive sequence of unique elements from the total number of elements
        minOps = Math.min(minOps, totalCount - (right - left));
    }

    // Return the minimum number of operations required
    return minOps;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code snippet involves several operations:

1. Sorting the unique elements in the array: This operation has a time complexity of `O(k log k)`, where `k` is the number of unique elements in the array.

2. The for-loop runs `k` times, where `k` is the number of unique elements after removing duplicates.

3. Inside the for-loop, we have a while-loop; but notice that each element is visited at most once by the while-loop because `j` only increases. This implies the while-loop total times through all for-loop iterations is `O(k)`.

Combining these complexities, we have a total time complexity of `O(k log k + k)`, which simplifies to `O(k log k)` because `k log k` will dominate for larger `k`.

Space Complexity

The space complexity is determined by:

1. Storing the sorted unique elements, which takes `O(k)` space.

2. Miscellaneous variables (`ans`, `j`, `n`), which use constant space `O(1)`.

Hence, the total space complexity is `O(k)` for storing the unique elements set. Note that `k` here represents the count of unique elements in the original `nums` list.