2294. Partition Array Such That Maximum Difference Is K

Medium Greedy Array Sorting

Problem Description

In the given problem, we are presented with an array of integers named `nums` and an integer `k`. Our goal is to partition the array into one or more subsequences so that no number appears in more than one subsequence, and furthermore, within each subsequence, the largest difference between the maximum and minimum elements does not exceed `k`. The task is to determine the minimum number of such subsequences required.

A subsequence is defined as a sequence that can be obtained from the original sequence by possibly removing elements, without changing the order of the remaining elements.

Intuition

To solve this problem, we can take advantage of the property that if a sequence is sorted, the difference between consecutive elements will give us the smallest possible differences. Thus, an effective approach is to sort the `nums` array first. After sorting, we can iterate through the elements, starting a new subsequence whenever we encounter an element that would cause the difference between the current element and the smallest element of the current subsequence to exceed `k`.

By always tracking the smallest element, `a`, of the current subsequence as we traverse through the sorted list, we can simply increment the count of subsequences, `ans`, each time we need to start a new subsequence. The moment we need to create a new subsequence is identified when the difference between the current number `b` and the smallest number `a` in the subsequence is greater than `k`. When that happens, `b` becomes the starting element of the new subsequence, and therefore, `a` is updated to `b`, and `ans` is incremented to reflect the creation of a new subsequence.

This approach ensures that for each subsequence we form, every pair of elements within the subsequence has a difference less than or equal to `k`, and that we minimize the total number of subsequences needed by using the sorted nature of the array to keep each subsequence as long as possible before the condition necessitating a new subsequence is met.

Solution Approach

The solution to this problem leverages the sorting algorithm and basic iteration over the sorted array. Here is a step-by-step breakdown of the implementation:

1. First, we sort the `nums` array. Sorting is a common pre-processing step when dealing with problems that require ordering or arranging of elements. In Python, this is easily done with the `.sort()` method, which sorts the list in place in ascending order. The sorted array allows us to consider elements in their natural order and make decisions based on their relative positions.
2. We initialize two variables: `ans` and `a`. The `ans` is set to `1` since, at minimum, one subsequence is needed, irrespective of the value of `k`. The `a` is set to the first element of the sorted array; it represents the smallest element of the current subsequence.
3. We iterate over each element `b` in the sorted `nums` array. During this iteration, we compare each element with `a` to determine if a new subsequence is needed. If `b - a` exceeds `k`, it means adding `b` to the current subsequence would violate the condition that the difference between the maximum and minimum values in any subsequence should be at most `k`.
4. When the condition `b - a > k` is true, a new subsequence is started with `b` as the initial element. This is done by updating `a` to be `b` and incrementing `ans` by one, to account for this new subsequence.
5. The iteration continues until all elements have been assigned to some subsequence. Since we are assured that elements in each subsequence satisfy the difference condition and that elements are not assigned to more than one subsequence, the process guarantees that the final value of `ans` is the minimum number of subsequences needed.

The algorithm's time complexity is dominated by the sorting step, which is $O(n \log n)$, where n is the number of elements in `nums`. The iteration is a linear scan, adding only $O(n)$ to the time complexity. Therefore, the total time complexity is $O(n \log n)$. There's no additional significant space complexity overhead, as everything is done in place, making the space complexity $O(1)$ aside from the input.

Overall, this is an efficient and straightforward approach that uses sorting to ensure the minimum number of subsequences by keeping track of the minimum element and incrementing the count of subsequences whenever a difference condition requires starting a new subsequence.

Example Walkthrough

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

Problem Example:

Given an array `nums = [3, 1, 2, 7, 4, 6]` and an integer `k = 3`, we want to find the minimum number of subsequences such that the difference between the max and min elements in each subsequence is at most `k`.

Step-by-step Walkthrough:

1. **Sort the array `nums`:** First, we sort the array to easily find subsequences where the differences between numbers are minimal. After sorting, our `nums` array becomes `[1, 2, 3, 4, 6, 7]`.
2. **Initialize variables:** We initialize `ans` to `1`, as at least one subsequence is always required. We also take the first element of the sorted array as variable `a`, so `a = 1`.
3. **Iterate and check differences:** We go through each sorted element `b`, starting from the second one, and check whether including `b` in the current subsequence would exceed the difference of `k = 3`.
   - `b = 2`: `b - a = 2 - 1 = 1`, which is not greater than `k`, so `b` is part of the current subsequence.
   - `b = 3`: `b - a = 3 - 1 = 2`, which, again, is not greater than `k`, so `b` remains in the current subsequence.
   - `b = 4`: `b - a = 4 - 1 = 3`, equal to `k`, but still not greater, so `b` is still in the current subsequence.
   - `b = 6`: `b - a = 6 - 1 = 5`, which is greater than `k`, so we cannot include `b` in this subsequence. Therefore, we increment `ans` by 1, making `ans = 2`, and update `a` to `b`, so now `a = 6`.
   - `b = 7`: `b - a = 7 - 6 = 1`, which is less than `k`, so `b` is included in this new subsequence.
4. **Incrementing counts of subsequences:** Each time we find that `b - a` exceeds `k`, we start a new subsequence and increment `ans`. In this example, this only happened once.

Final result: After covering all elements, we've determined that a minimum of `ans = 2` subsequences are needed to satisfy the stated conditions, resulting in subsequences, say, `[1, 2, 3, 4]` and `[6, 7]`.

This example demonstrates that by sorting the array and smartly grouping elements, we minimize the number of subsequences needed while satisfying the conditions of the problem.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def partitionArray(self, nums: List[int], k: int) -> int:
        # First, sort the array to allow for linear partitioning.
        nums.sort()

        # Initialize the number of partitions needed with at least 1.
        partitions_count = 1

        # Set the first element as the initial value for comparison.
        min_value_in_current_partition = nums[0]

        # Iterate over sorted numbers to determine the need for partitions.
        for num in nums:
            # If the current number minus the minimum value in the current
            # partition exceeds k, this requires a new partition.
            if num - min_value_in_current_partition > k:
                # Set the current number as the new minimum value for the next partition.
                min_value_in_current_partition = num

                # Increment the partitions count as we need another partition.
                partitions_count += 1

        # Return the total number of partitions needed.
        return partitions_count
```

Java

```java
class Solution {

    public int partitionArray(int[] nums, int k) {
        // Sort the input array in ascending order
        Arrays.sort(nums);

        // Initialize the count of partitions needed, starting with 1
        int partitionCount = 1;

        // Store the first number as the starting point of the first partition
        int partitionStart = nums[0];

        // Iterate throuoh all numbers in the array
        for (int currentNumber : nums) {
            // If the current number minus the partition start is greater than k,
            // a new partition is needed
            if (currentNumber - partitionStart > k) {
                // Update the starting point to the current number
                partitionStart = currentNumber;
                // Increment the partition count
                ++partitionCount;
            }
        }

        // Return the total number of partitions required
        return partitionCount;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to determine the minimum number of groups with each group having differences
    // between numbers not greater than k.
    int partitionArray(vector<int>& nums, int k) {
        // Sort the input vector to make it easier to group elements.
        sort(nums.begin(), nums.end());

        // Initialize 'groupsCount' to 1 since the first element starts the first group.
        int groupsCount = 1;
        // The 'currentGroupStart' is the first element in the current group.
        int currentGroupStart = nums[0];

        // Iterate through the sorted numbers.
        for (int& currentNum : nums) {
            // If the difference between the current number and the start of the current group
            // is greater than k, a new group must be started.
            if (currentNum - currentGroupStart > k) {
                // Update 'currentGroupStart' to the current number.
                currentGroupStart = currentNum;
                // Increment the 'groupsCount' as a new group is created.
                groupsCount++;
            }
        }
        // Return the total number of groups created.
        return groupsCount;
    }
};
```

TypeScript

```typescript
function partitionArray(nums: number[], k: number): number {
    // Sort the array in non-decreasing order to allow for partition counting based on the difference
    nums.sort((a, b) => a - b);

    // Initialize the partition count to 1, as there will be at least one partition
    let partitionCount = 1;

    // Set the current number to track from the first number in the sorted array
    let currentNum = nums[0];

    // Iterate through the sorted numbers
    for (const nextNum of nums) {
        // Check if the current number and the next number's difference is greater than 'k'
        if (nextNum - currentNum > k) {
            // If so, set the current tracking number to the next number
            // and increment the partition count as we need a new partition
            currentNum = nextNum;
            partitionCount++;
        }
    }

    // Return the total number of partitions needed
    return partitionCount;
}
```

```python
from typing import List

class Solution:
    def partitionArray(self, nums: List[int], k: int) -> int:
        # First, sort the array to allow for linear partitioning.
        nums.sort()

        # Initialize the number of partitions needed with at least 1.
        partitions_count = 1

        # Set the first element as the initial value for comparison.
        min_value_in_current_partition = nums[0]

        # Iterate over sorted numbers to determine the need for partitions.
        for num in nums:
            # If the current number minus the minimum value in the current
            # partition exceeds k, this requires a new partition.
            if num - min_value_in_current_partition > k:
                # Set the current number as the new minimum value for the next partition.
                min_value_in_current_partition = num

                # Increment the partitions count as we need another partition.
                partitions_count += 1

        # Return the total number of partitions needed.
        return partitions_count
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code primarily comes from sorting the `nums` list. The sort operation in Python typically has an average case time complexity of $O(n \log n)$, where n is the length of the list being sorted. After sorting, the code iterates through the sorted list once, which has a time complexity of $O(n)$. Since $O(n \log n)$ dominates the overall time complexity, we can conclude that the total time complexity of the function is $O(n \log n)$.

Space Complexity

The space complexity of the code is the amount of additional memory space required by the algorithm as a function of the input size. The sorting of the list `nums` in place does not require extra space proportional to the size of the input, resulting in a space complexity of $O(1)$, which means it is constant space complexity. However, it is worth mentioning that the sort algorithm may have a space complexity of $O(\log n)$ under the hood because of the stack frames used in recursive calls if the sort algorithm is based on recursion (like quicksort or mergesort). But this is not reflected in the auxiliary space usage as it is not part of the explicit space used by the program.

Thus, the space complexity is $O(1)$ or $O(\log n)$ depending on the implementation details of the sorting algorithm which is an implementation detail of Python's `sort()` method.