2966. Divide Array Into Arrays With Max Difference

Medium Arrays Array Sorting

Problem Description

In this problem, we're given an array of integers, `nums`, with a size `n` and a positive integer `k`. The objective is to split this array into one or more subarrays where each subarray has exactly 3 elements. There are certain conditions that we have to follow when creating these subarrays. Firstly, each element from the original array must be used once and only once—this means that each element in `nums` must be put into exactly one subarray. Secondly, for any given subarray, the difference between the largest and smallest values within that subarray can't be greater than `k`. The task is to return a 2D array with all the constructed subarrays that fit these criteria. If it's not possible to divide the array under these conditions, we must return an empty array. Also, if there are multiple ways to divide the array that fit the requirements, we are free to return any one of them.

Intuition

To solve this problem, a straightforward approach is to first impose an order by sorting the array. Once sorted, we can confidently compare adjacent elements knowing that they represent the nearest possible grouping by value.

After sorting the array, we start taking chunks of three elements at a time from the start since we're required to have subarrays of size three. For each set of three elements, we inspect the difference between the maximum element and the minimum element. Because the array is sorted, these elements will be the first and the last in the chunk.

If the difference exceeds `k`, then we know it's not possible to divide the array while satisfying the conditions (because a sorted array ensures that this set of three has the smallest possible maximum difference). Therefore, we can terminate and return an empty array.

If the difference is within `k`, this grouping is valid, and we can add it to the list of answers. We repeat this process, moving forward in the array by three elements each time until we've successfully created subarrays out of all elements in `nums`. The solution approach ensures that we consider each element exactly once and check for the condition without any backtracking, thus providing an efficient and correct way of dividing the array into subarrays, or determining if it cannot be done.

Solution Approach

The implementation of the solution uses a straightforward algorithm and the basic data structure of arrays (or lists in Python). The key pattern used here is sorting, which is a common first step in a variety of problems to arrange elements in a non-decreasing order.

Here's a step-by-step walk-through of the algorithm, referring to the reference solution approach:

1. **Sort the array**: We apply a built-in sorting function to the `nums` array, rearranging its elements in ascending order. This is a crucial step because it allows us to easily check the difference between the smallest and largest numbers in any subsequent groups of three.
2. **Initialize the answer list**: An empty list `ans` is initialized to store the valid subarrays if we can form them.
3. **Iterate through the array in chunks of three**: The sorted `nums` array is traversed using a for-loop with a step of `3` in `range(0, n, 3)`. Each iteration corresponds to a potential subarray.
4. **Check the difference constraint**: In the loop, we take a slice of the array from index `i` to `i + 3`, which includes three elements. We then check if the difference between the last element (which is the maximum because of sorting) and the first element (which is the minimum) exceeds `k`.
   - **Violation of the constraint**: If this difference is greater than `k`, we know it is impossible to form a subarray that meets the condition, and therefore an empty array is immediately returned, as it indicates that we cannot divide the array successfully.
   - **Constraint satisfied**: If the difference does not exceed `k`, this slice of three elements is a valid subarray, and we add it to the answer list `ans`.
5. **Return the result**: Once the loop has finished and no constraint has been violated, we return the `ans` list, which contains all successfully formed subarrays.

Throughout this process, no additional data structures are needed other than the input array and the output list. The time complexity of the algorithm is primarily driven by the sorting step, which is typically `O(n log n)`. Since traversing the sorted array and checking for the conditions is done in linear time — `O(n)` — the total time complexity remains `O(n log n)`.

This solution is elegant in its simplicity, using a commonly understood and implemented pattern of sorting, and demonstrates the power of transforming a problem space to make conditions easier to verify.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the algorithm described above.

Suppose our input array is `nums = [4, 8, 2, 7, 6, 1, 9]` and `k = 3`. We want to create subarrays with exactly 3 elements each where the difference between the maximum and smallest values in a subarray should not exceed `k`. Let's follow the steps:

1. **Sort the array**: We sort `nums` to get `[1, 2, 4, 6, 7, 8, 9]`.
2. **Initialize the answer list**: We create an empty list `ans = []` to hold our subarrays.
3. **Iterate through the array in chunks of three**: We consider subarrays `nums[0:3]`, `nums[3:6]`, and `nums[6:9]`. These are `[1, 2, 4]`, `[6, 7, 8]`, and the single element `[9]` left out (which can't form a subarray of size three).
4. **Check the difference constraint**:
   - For the first subarray `[1, 2, 4]`, the difference between `4` (max) and `1` (min) is `3`, which is equal to `k`, so this subarray is valid. We add `[1, 2, 4]` to `ans`.
   - For the second subarray `[6, 7, 8]`, the difference between `8` (max) and `6` (min) is `3`, which is less than `k`, so this subarray is also valid. We add `[6, 7, 8]` to `ans`.
   - The last element `[9]` cannot form a subarray because there are not enough elements to make a set of three. However, had it been possible to create another subarray with exactly 3 elements, we would have checked it following the same method.
5. **Return the result**: We finish the loop and return `ans` which now contains `[[1, 2, 4], [6, 7, 8]]`.

Here, the key takeaways from the example are:

- Sorting the array helps us group the nearest numbers together and check if they satisfy the condition.
- Since the array is processed in sorted order, if any subset of three elements doesn't satisfy the condition, we can immediately conclude that it's not possible to split the array as required, because any other grouping would only increase the difference.
- The entire process requires only the sorted array and an additional list to store valid subarrays, which is very space-efficient.

The resulting subarrays from our example adhere to the rules of the problem, and the example has demonstrated how the algorithm successfully applies the concepts described in the solution approach.

Solution Implementation

Python

```python
class Solution:
    def dividArray(self, nums: List[int], k: int) -> List[List[int]]:
        # Sort the input array to make sure that subarrays with a maximum size difference of k can be found.
        nums.sort()
        # Initialize an empty list to store the resulting subarrays.
        divided_arrays = []
        # Calculate the length of the input list.
        nums_length = len(nums)

        # Iterate over the array in steps of 3, as we want subarrays of size 3.
        for i in range(0, nums_length, 3):
            # Generate a subarray of size 3 from the sorted list.
            subarray = nums[i: i + 3]

            # Check if the subarray has 3 elements and the maximum size difference condition holds.
            # If not, return an emptv list as the condition cannot be met.
            if len(subarray) < 3 or subarray[2] - subarray[0] > k:
                return []

            # If the condition is met, add the valid subarray to the result list.
            divided_arrays.append(subarray)

        # Return the list of all valid subarrays after iterating through the entire input list.
        return divided_arrays
```

Java

```java
import java.util.Arrays;

class Solution {

    /**
     * Divides an array into smaller sub-arrays of size 3, ensuring the difference between
     * the maximum and minimum values within each sub-array does not exceed a given value k.
     *
     * @param nums the input array of integers to be divided.
     * @param k    the maximum allowable difference between the largest
     *             and smallest numbers in each sub-array.
     * @return a 2D array where each sub-array has 3 elements and the above condition is met,
     * or an empty 2D array if the condition cannot be met.
     */
    public int[][] dividArray(int[] nums, int k) {
        // Sort the array to organize numbers and facilitate the process of division
        Arrays.sort(nums);
        // Get the length of the input array
        int n = nums.length;
        // Initialize the result array with the correct size based on the input array
        int[][] result = new int[n / 3][];
        // Iterate through the array, incrementing by 3 each time to create sub-arrays of size 3
        for (int i = 0; i < n; i += 3) {
            // Copy a range of the sorted array to form a sub-array of size 3
            int[] subArray = Arrays.copyOfRange(nums, i, i + 3);

            // Check if the largest difference in the current sub-array exceeds the limit k
            if (subArray[2] - subArray[0] > k) {
                // If it does, return an empty array as the condition can't be met
                return new int[][] {};
            }

            // Assign the sub-array to the correct position in the result array
            result[i / 3] = subArray;
        }
        // Return the duly formed result array
        return result;
    }
}
```

C++

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

class Solution {
public:
    // Method to divide the array into subarrays where
    // the largest number and smallest number difference is no greater than k
    // nums: The input array of integers
    // k: The maximum allowed difference between the largest and smallest
    // numbers in each subarray
    std::vector<std::vector<int>> dividArray(std::vector<int>& nums, int k) {
        // Sort the input array
        std::sort(nums.begin(), nums.end());

        // Initialize a vector of vectors to store the resulting subarrays
        std::vector<std::vector<int>> result;

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

        // Iterate over the array in steps of 3 since we are creating trinary subarrays
        for (int i = 0; i < n; i += 3) {
            // Check if there are enough elements remaining to create a subarray
            if (i + 2 >= n) {
                return {}; // Returning an empty vector as a failure case
            }

            // Creating a temporary vector for the current subarray
            std::vector<int> currentSubarray = { nums[i], nums[i + 1], nums[i + 2] };

            // Check if the difference between the largest and smallest numbers
            // in the current subarray is greater than k
            if (currentSubarray[2] - currentSubarray[0] > k) {
                return {}; // Returning an empty vector as a failure case
            }

            // Add the current subarray to the result
            result.emplace_back(currentSubarray);
        }

        // Return the resultant vector of subarrays
        return result;
    }
};
```

TypeScript

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

    // Initialize an empty array to store the groups formed
    const groups: number[][] = [];

    // Iterate through the sorted array in steps of 3
    for (let i = 0; i < nums.length; i += k) {
        // Extract a subarray of size k from the current position
        const subArray = nums.slice(i, i + k);

        // Check if the difference between max and min of subArray is more than k
        if (subArray[k - 1] - subArray[0] > k) {
            // Return an empty array if the condition is not satisfied
            return [];
        }

        // If the condition is satisfied, push the subArray into the groups array
        groups.push(subArray);
    }

    // Return the groups array containing all subarrays
    return groups;
}
```

```python
class Solution:
    def dividArray(self, nums: List[int], k: int) -> List[List[int]]:
        # Sort the input array to make sure that subarrays with a maximum size difference of k can be found.
        nums.sort()
        # Initialize an empty list to store the resulting subarrays.
        divided_arrays = []
        # Calculate the length of the input list.
        nums_length = len(nums)

        # Iterate over the array in steps of 3, as we want subarrays of size 3.
        for i in range(0, nums_length, 3):
            # Generate a subarray of size 3 from the sorted list.
            subarray = nums[i: i + 3]

            # Check if the subarray has 3 elements and the maximum size difference condition holds.
            # If not, return an emptv list as the condition cannot be met.
            if len(subarray) < 3 or subarray[2] - subarray[0] > k:
                return []

            # If the condition is met, add the valid subarray to the result list.
            divided_arrays.append(subarray)

        # Return the list of all valid subarrays after iterating through the entire input list.
        return divided_arrays
```

Time and Space Complexity

The time complexity of the code is $O(n \times \log m)$ because the `sort` function is used, which typically has a time complexity of $O(m \log m)$ where n is the number of elements in the array to be sorted. After sorting, the code iterates through the list in steps of 3 using a for loop, which is $O(n/3)$. However, this simplifies to $O(n)$ because constants are dropped in Big O notation. Therefore, the combined time complexity, taking the most significant term, remains $O(n \log m)$.

The space complexity of the code is $O(n)$ because a new list `ans` is created to store the subarrays. The size of `ans` will be proportional to the size of the input array `nums`. Thus, as the length of the input array increases, the space consumption of `ans` scales linearly.