2111. Minimum Operations to Make the Array K-Increasing

Hard Array Binary Search Leetcode Link

Problem Description

You are provided with an array `arr` of `n` positive integers and a positive integer `k`. An array is defined as `K-increasing` if for every index `i` such that `k <= i <= n-1`, the inequality `arr[i-k] <= arr[i]` is satisfied. In other words, for any element in the array, if you move `k` steps backward, you should not find a larger number.

For example, the array `[4, 1, 5, 2, 6, 2]` is K-increasing when `k=2` because every element is greater than or equal to the element which is 2 places before it. However, it is not K-increasing for `k=1` because `4` (element at index `0`) is greater than `1` (element at index `1`).

The task is to convert the array into a K-increasing array by performing the minimum number of operations. In one operation, you can choose an index `i` and change `arr[i]` to any positive integer.

The goal is to find out the minimum number of such operations needed to make the array K-increasing given the value of `k`.

Intuition

The solution utilizes a dynamic programming approach with a twist. The idea is that if you divide the array into `k` subarrays, where each subarray contains elements that are `k` indices apart in the original array, you'll notice that for the array to be K-increasing overall, each of these subarrays must be non-decreasing.

Here's the intuition broken down step by step:

1. **Subarray Division**: Consider the `arr=[4, 1, 5, 2, 6, 2]` and `k=2`; we have two subarrays `[4, 5, 6]` and `[1, 2, 2]`. If each of these subarrays is non-decreasing, the original array is K-increasing.
2. **Longest Increasing Subsequence (LIS)**: For each subarray, we want to keep it non-decreasing with the minimum number of changes. To achieve this, we need to find the length of the Longest Increasing Subsequence (LIS) of the subarray. The reason behind this is that elements in the LIS do not need to be changed, as they already contribute to making the subarray non-decreasing.
3. **Operations Count**: Once we have the LIS length, the number of operations required to make the subarray non-decreasing is the total number of elements minus the LIS length. This is because we can keep the LIS as is and change the other elements to fit around it.
4. **Summing Up**: Since our original array is divided into `k` subarrays, we apply the LIS strategy for each subarray and sum up the operations required. This will give us the total minimum number of operations needed to make the entire array K-increasing.

The provided solution uses a helper function `lis(arr)` which calculates the required operations for a given subarray by finding the length of the LIS using binary search (`bisect_right`). Then it uses list comprehension to sum up the operations for each subarray, which are slices of the original array (`arr[i::k]` for `i` in `range(k)`).

This approach is efficient because it reduces the problem to `k` individual LIS problems and avoids unnecessary changes to elements that are already part of the LIS, hence minimizing the number of operations.

Solution Approach

The key to implementing the solution is understanding the concept of Longest Increasing Subsequence (LIS) within the context of the given array and how it applies to each of the `k` subsequences.

The provided Python solution includes a nested function `lis`, which is the implementation of the LIS algorithm. This is not the traditional dynamic programming solution for LIS with `O(n^2)` complexity but a more efficient version that uses binary search (`bisect_right` from the `bisect` module) and has a time complexity of `O(n log n)` for each subsequence.

Here's the breakdown of the algorithms and data structures used in the solution:

1. **Nested Function `lis(arr)`**:
   - This function computes the length of the LIS for a given subarray.
   - It initializes an empty list `t` that will store the last element of the smallest increasing subsequence of each length found so far.
   - The function iterates over each element `x` in the subarray:
     - Using binary search (`bisect_right`), it finds the position `idx` in `t` where `x` could be placed to either extend an existing subsequence or replace an element to create a potential new subsequence.
     - If `idx` is equal to the length of `t`, it means `x` is larger than any element in `t`, and we can append `x` to `t`, effectively extending the longest subsequence seen so far.
     - Otherwise, we replace the element at `t[idx]` with `x`, as `x` might be the start of a new potential subsequence or a smaller end element for a subsequence of length `idx`.
   - After processing all elements in the subarray, the length of LIS is obtained by subtracting the length of `t` from the length of the subarray (`len(arr) - len(t)`).
2. **Combining the Results**:
   - The main part of the solution is a single line that sums up the operation counts for each `k` subsequence: `sum(lis(arr[i::k]) for i in range(k))`.
   - It iterates over each start index from `0` to `k-1` and takes every `k`-th element from the original array using slicing `arr[i::k]`. This yields `k` subsequences that must each individually be non-decreasing to satisfy the K-increasing property.
   - For each subsequence, it applies the `lis` function to find the number of operations needed, which is then accumulated to get the total minimum number of operations required for the entire array.

By applying the LIS algorithm separately to each of the `k` subsequences, the solution effectively translates the problem of making an array K-increasing into multiple independent subproblems. Each subproblem aims to minimize the adjustments within its subsequence, and the sum of the solutions to these subproblems is the minimum number of operations needed for the whole array.

Example Walkthrough

Let's consider a small example array `arr = [3, 9, 4, 6, 7, 5]` and `k = 3`. According to the given solution approach, we need to divide this array into `k` subarrays where each subarray contains elements that are `k` indices apart in the original array. Let's do this step by step:

1. **Subarray Division**:
   - With `k = 3`, we divide the original array into 3 subarrays: `[3, 6], [9, 7], [4, 5]`.
   - These subarrays are created by taking every third element from the original array starting from indices `0`, `1`, and `2` respectively.
2. **Longest Increasing Subsequence (LIS)**:
   - We then need to determine the LIS for each subarray:
     - For the first subarray `[3, 6]`, the LIS is `[3, 6]` itself, and the length is 2.
     - For the second subarray `[9, 7]`, since `7` is not larger than `9`, the LIS is `[7]`, and the length is 1.
     - For the third subarray `[4, 5]`, the LIS is `[4, 5]`, and the length is 2.
3. **Operations Count**:
   - We calculate the number of operations required to make each subarray non-decreasing:
     - For `[3, 6]`, no operations are needed as it is already non-decreasing.
     - For `[9, 7]`, we need `2 - 1 = 1` operation, changing `9` to a number not greater than `7` (e.g., `7` or any smaller number).
     - For `[4, 5]`, no operations are needed as it is already non-decreasing.
4. **Summing Up**:
   - Sum up the operations required: `0` (for the first subarray) + `1` (for the second subarray) + `0` (for the third subarray) = `1`.
   - Therefore, we need a minimum of `1` operation to make the entire array `k-increasing`.

This walkthrough illustrates the solution steps using a simple example, showcasing how to divide the original problem into smaller ones by finding the LIS of the subarrays and then deducing the minimum number of operations needed to achieve a `k-increasing` array.

Python Solution

```python
from bisect import bisect_right
from typing import List

class Solution:
    def k_increasing(self, arr: List[int], k: int) -> int:
        # Function to calculate the length of longest increasing subsequence.
        def longest_increasing_subsequence(sub_arr):
            tails = []  # holds the smallest tail of all increasing subsequences with length i+1
            for val in sub_arr:
                # Find the index in the tails array where we can place the value
                # or where it can replace an existing value.
                idx = bisect_right(tails, val)
                # If the index is equal to the length of the list, it means
                # all elements in tails are smaller than val, hence val extends the subsequence.
                if idx == len(tails):
                    tails.append(val)
                else:
                    # Else, it replaces the value in tails, maintaining the smallest possible tail
                    # for longest increasing subsequences with their respective lengths.
                    tails[idx] = val
            # The difference between the length of the current sub-array
            # and the longest increasing subsequence gives the number of changes needed.
            return len(sub_arr) - len(tails)

        # Sum the changes needed for each of the 'k' subsequences
        # generated by taking every k-th element of arr, starting from index i.
        return sum(longest_increasing_subsequence(arr[i::k]) for i in range(k))

# Example usage:
# sol = Solution()
# arr_example = [5, 1, 3, 4, 2]
# k_example = 1
# result = sol.k_increasing(arr_example, k_example)
# print(result)  # Output will be the minimum number of operations needed

```

Java Solution

```java
class Solution {
    public int kIncreasing(int[] arr, int k) {
        int n = arr.length; // Get the length of the array
        int ans = 0; // Initialize answer to 0

        // Iterate over the first k elements
        for (int i = 0; i < k; ++i) {
            List<Integer> subsequence = new ArrayList<>(); // List to hold subsequences
            // Populate the subsequences with elements spaced k apart
            for (int j = i; j < n; j += k) {
                subsequence.add(arr[j]);
            }
            // Increment the answer by the number of modifications needed
            // to make the subsequence strictly increasing
            ans += leastIncrementsNeeded(subsequence);
        }

        return ans; // Return the total number of modifications for all subsequences
    }

    // Determine the least number of increments needed to make the given list strictly increasing
    private int leastIncrementsNeeded(List<Integer> arr) {
        List<Integer> temp = new ArrayList<>(); // Temporary list to hold the longest increasing subsequence
        for (int x : arr) {
            int idx = findInsertionIndex(temp, x);
            // If the element is greater than all elements in temp, add it to the end
            // Otherwise, replace the first element that is greater or equal to x
            if (idx == temp.size()) {
                temp.add(x);
            } else {
                temp.set(idx, x);
            }
        }
        // The difference between the list size and temp size is the number of increments needed
        return arr.size() - temp.size();
    }

    // Binary search to find the rightmost index to insert the element
    private int findInsertionIndex(List<Integer> arr, int x) {
        int left = 0; // Left pointer for the binary search
        int right = arr.size(); // Right pointer for the binary search

        // Perform the binary search
        while (left < right) {
            // Compute mid-point, equivalent to (left + right) / 2 but avoids potential overflow
            int mid = (left + right) >> 1;
            // If current element is greater, ignore the right half
            if (arr.get(mid) > x) {
                right = mid;
            } else { // Otherwise, ignore the left half
                left = mid + 1;
            }
        }

        return left; // Return the computed index
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine the minimum number of elements to change to make
    // each subsequence formed by taking every k-th element non-decreasing
    int kIncreasing(vector<int>& arr, int k) {
        int changesNeeded = 0; // This will hold the total number of changes needed
        int n = arr.size(); // Size of the original array

        // Loop through each of the k subsequences
        for (int i = 0; i < k; ++i) {
            vector<int> subsequence; // This vector will hold the elements of the i-th subsequence

            // Construct the subsequence by taking every k-th element starting from index i
            for (int j = i; j < n; j += k) {
                subsequence.push_back(arr[j]);
            }

            // Add the number of changes needed for this subsequence to the total count
            changesNeeded += calculateLIS(subsequence);
        }

        return changesNeeded; // Return the total number of changes needed
    }

    // Function to calculate the length of the longest increasing subsequence (LIS)
    // and by extension, the minimum number of changes needed to make the subsequence increasing
    int calculateLIS(vector<int>& subsequence) {
        vector<int> lis; // This will hold the longest increasing subsequence

        // Iterate through the elements of the subsequence to construct the LIS
        for (int num : subsequence) {
            // Find the first element in the LIS which is greater than the current element
            auto it = upper_bound(lis.begin(), lis.end(), num);

            // If no such element is found, this means current element can be placed at the end of the LIS
            if (it == lis.end())
                lis.push_back(num); // Add current element to the LIS
            else
                *it = num; // Otherwise, replace the found element with the current element
        }

        // The number of changes needed is equal to the size of the original subsequence
        // minus the size of the longest increasing subsequence
        return subsequence.size() - lis.size();
    }
};

```

Typescript Solution

```typescript
// Determines the minimum number of elements to change to make
// each subsequence formed by taking every k-th element non-decreasing
function kIncreasing(arr: number[], k: number): number {
    let changesNeeded = 0; // This will hold the total number of changes needed
    let n = arr.length; // Size of the original array

    // Loop through each of the k subsequences
    for (let i = 0; i < k; ++i) {
        let subsequence: number[] = []; // This array will hold the elements of the i-th subsequence

        // Construct the subsequence by taking every k-th element starting from index i
        for (let j = i; j < n; j += k) {
            subsequence.push(arr[j]);
        }

        // Add the number of changes needed for this subsequence to the total count
        changesNeeded += calculateLIS(subsequence);
    }

    return changesNeeded; // Return the total number of changes needed
}

// Calculates the length of the longest increasing subsequence (LIS)
// and, by extension, the minimum number of changes needed to make the subsequence increasing
function calculateLIS(subsequence: number[]): number {
    let lis: number[] = []; // This will hold the longest increasing subsequence

    // Iterate through the elements of the subsequence to construct the LIS
    for (const num of subsequence) {
        // Find the first element in the LIS which is greater than the current element
        let it = upperBound(lis, num);

        // If no such element is found, this means current element can be placed at the end of the LIS
        if (it === lis.length)
            lis.push(num); // Add current element to the LIS
        else
            lis[it] = num; // Otherwise, replace the found element with the current element
    }

    // The number of changes needed is equal to the size of the original subsequence
    // minus the size of the longest increasing subsequence
    return subsequence.length - lis.length;
}

// TypeScript doesn't have a standard library function like C++'s upper_bound,
// so we define it here. It finds the first index where `value` should be inserted
// into `array` to maintain order.
function upperBound(array: number[], value: number): number {
    let low = 0, high = array.length;
    while (low < high) {
        let mid = Math.floor((low + high) / 2);
        let midVal = array[mid];
        if (midVal <= value) { // Change this to '<' to make function work exactly like std::upper_bound
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return low; // Returns the correct index to insert value to maintain sorted order
}

```

Time and Space Complexity

The given code implements a function that, for a given list `arr` and an integer `k`, finds the minimum number of elements to change to ensure that every `k`-th subsequence of the list is non-decreasing.

Time Complexity:

To analyze the time complexity, let's break down the process:

1. The `lis` function is called `k` times, once for each of the `k` subsequences.
2. Inside the `lis` function, there's a loop that goes through the elements of a subsequence (which has a length of about `n/k`, where `n` is the length of `arr`).
3. In the worst case, `bisect_right` performs a binary search, which has a time complexity of `O(log m)` where `m` is the size of the temporary list `t`.
4. The size of `t` can grow up to the size of the subsequence being considered, in the worst case approximated to `n/k`.

Putting it all together:

- Single call to `lis`: `O((n/k) * log(n/k))`
- `lis` called `k` times: `O(k * (n/k) * log(n/k))` = `O((n * log(n/k)))`

Since we have a `log` term that depends on `n/k`, the overall time complexity isn't perfectly linear with respect to `n`. However, as `k` increases, the time complexity approaches `O(n log n)` since the subsequences processed by each call get shorter.

Space Complexity:

The space complexity can be evaluated by considering:

1. The temporary list `t` used inside the `lis` function, which holds the elements of the longest increasing subsequence (LIS) within a `k`-th subsequence;
2. `t`'s size is at most `n/k` for a given `k`-th subsequence; However, `t` is reused for each subsequence and does not grow with `k`.
3. There are no additional data structures that grow with the size of the input, other than the input itself and the function call stack.

Hence, the space complexity is `O(n/k)`, which simplifies to `O(n)` because we keep a single `t` for each subsequence.

Please note: since the actual maximum length of `t` can vary depending on the input `arr`, the space complexity in practice can be less than `O(n)` if the subsequences have a strong increasing trend, but in the worst case, it is `O(n)`.