3018. Maximum Number of Removal Queries That Can Be Processed I

Hard Array Binary Search Prefix Sum

Problem Description

You are provided with two arrays: `nums` which is the main array with **0-indexed** elements, and `queries`, also a **0-indexed** array containing query elements. Your task is to process the elements in `queries` sequentially by following a set of rules:

1. You can initially choose to replace `nums` with any of its subsequences. This operation is optional and can be performed only once.
2. Each element in `queries` is processed according to this procedure:
   - Check the first and last element of `nums`. If both are smaller than the current query element, stop processing further queries.
   - If not, you must choose and remove an element from either end of `nums`, provided that the chosen element is greater than or equal to the query element.

Your goal is to determine the maximum number of queries that can be processed if you choose to perform the initial operation optimally.

Intuition

To arrive at the solution, we need to keep track of the range within `nums` we have available for answering queries. A dynamic programming (DP) approach is suitable for this situation. Wherein we define a DP table `f` where each entry `f[i][j]` represents the maximum number of queries we can process when the subarray of `nums` from `i` to `j` (inclusive) is still intact.

The solution must account for two possible actions for each element of `queries`:

- If we choose to delete the first element of our current subarray, we must have a way to store and use the result of this action for further decisions.
- Likewise, if we choose to delete the last element, that decision's impact should also be recorded and available for upcoming queries.

The answer can be built iteratively. For each entry `f[i][j]`, we consider the implications of removing an element from the beginning (`i-1`) or the end (`i+1`) of the subarray. The choice depends on whether the elements on the edges satisfy the condition with regard to the current query element.

By updating the DP table this way, when `f[i][j]` equals the total number of queries, we can immediately return this number, as we can't process more queries than we have. If this case isn't reached, the final answer is the maximum value of `f[i][j]` plus the condition that `nums[i]` must be greater than or equal to `queries[f[i][j]]`. This condition checks if we can take one more query when the subarray is reduced to a single element.

The intuition is that, with careful selection of which element to delete when processing each query and using DP to memorize our choices' outcomes, we can determine the optimal way to process the maximum number of queries.

Solution Approach

The given solution uses Dynamic Programming (DP), a common technique for solving problems where subproblems overlap, and solutions can be built upon previously solved subproblems to reach the overall solution.

The key data structure used is a two-dimensional array `f` of size `n x n`, where `n` is the length of `nums`. Each entry `f[i][j]` in this DP table holds the maximum number of queries the algorithm can process given that we are considering `nums[i]` through `nums[j]` as the current range.

Here's how the implementation of dynamic programming works step by step:

1. Iterate over all possible subarrays of `nums` by using two nested loops. The outer loop variable `i` refers to the beginning of the subarray, and the inner loop variable `j` (which starts from `n - 1` and moves backwards to `i`) refers to the end of the subarray.
2. At each iteration `(i, j)`, we try to update `f[i][j]` using two possible actions:
   - If we remove the first element of the subarray (`nums[i - 1]` when `i > 0`), we consider the value in `f[i - 1][j]`, as this represents the maximum number of queries processed when `nums[i - 1]` is included. If `nums[i - 1]` is greater than or equal to the current query (indicated by `queries[f[i - 1][j]]`), we can increment the number of processed queries by 1.
   - Similarly, if we decide to remove the last element of the subarray (`nums[j + 1]` when `j + 1 < n`), we consider the value in `f[i][j + 1]`. Again, if `nums[j + 1]` satisfies the current query, the value of `f[i][j]` can be incremented.
3. During the iteration, if at any point `f[i][j]` equals `n`, the total number of queries, we know that we can process all the queries. Thus, we return `n` immediately, as it's the maximum possible.
4. If the loop completes without returning, it means not all queries can be processed. The final step is to find the maximum of `f[i][i]` for all `i` from `i` to `n - 1`. We add 1 to `f[i][i]` if `nums[i]` is greater than or equal to the 'next' query (`queries[f[i][i]]`) to consider the possibility of processing one more query.

The choice of using a 2D DP table is necessitated by the need to maintain a count of processed queries across different subarray configurations. It allows us to make decisions based on the range of elements available at each step and optimize the number of queries processed.

Example Walkthrough

Let's say we have the following `nums` and `queries` arrays:

- `nums = [3, 5, 1, 4]`
- `queries = [2, 4, 6]`

With `nums` being [3, 5, 1, 4], we have the option to replace `nums` with any of its subsequences. Since we have a 0-based index, the initial `nums` array is from index 0 to index 3.

We'll initialize our DP table `f` with the dimensions of the `nums` array. In this case, we have 4 elements, so our table `f` will be a 4×4 DP grid.

Now, we'll iterate over the `queries` and `nums` to fill in the DP table:

1. The first query is 2. We start with subarray `i=0, j=3`, which is [3, 5, 1, 4].
   - At `f[0][3]`, none of the edge elements (3 and 4) are smaller than the query (2), so we can remove a value. We choose to remove 3 from the beginning, which makes our subarray now [5, 1, 4], and set `f[0][3]` to 1, as we have processed one query.
2. We continue with subarray `i=1, j=3`, and the same query.
   - At `f[1][3]`, again the edge elements (5 and 4) are not smaller than 2, so we can remove another value. We can choose either 5 or 4. We remove 5 and set `f[1][3]` to 2, as we have processed another query.
3. Now the subarray is `i=2, j=3` in [1, 4], and we move on to the next query, which is 4.
   - At `f[2][3]`, the edge elements (1 and 4) are not both smaller than the query (4), so we can only remove the last element 4. Now our subarray is [1], and `f[3][3]` is set to 3.
4. As our subarray `i=3, j=3` is just [1] and we have the final query 6, we cannot process this query because the remaining element is smaller than 6.

Now we have completed the iterations. We look at our DP table `f` and find the maximum value of `f[i][i]`:

- `f[0][0]` would consider only [3], but since there's no query `<= 3` after processing 2, it stays 0.
- `f[1][1]` would consider only [5], which is `>= 2` and 4, so we could have processed 2 queries if it was a singleton from the start.
- `f[2][2]` would consider only [1], which is `< 2`, so it stays 0.
- `f[3][3]` ends up as 3 because we remove 4 in response to query 4.

The max value we find is `f[3][3]`, so the answer is 3. We processed 3 queries successfully before we were left with an element smaller than the next query.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def maximumProcessableQueries(self, processes: List[int], queries: List[int]) -> int:
        # Number of processes
        num_processes = len(processes)
        # Initialize a 2D array for the dynamic programming table with zeros
        dp_table = [[0] * num_processes for _ in range(num_processes)]
        # Number of queries
        num_queries = len(queries)

        # Fill in the dp_table for all possible segments [i, j] of processes
        for i in range(num_processes - 1, -1, -1):
            for j in range(num_processes - 1, i - 1, -1):
                # If we're not in the first row, check whether we can include the process at i-1
                if i:
                    dp_table[i][j] = max(
                        dp_table[i][j],
                        # Current value
                        # If the current process can handle the next query,
                        # add 1 to the value from the row above
                        dp_table[i - 1][j] + (processes[i - 1] >= queries[dp_table[i - 1][j]])
                    )
                # If we're not in the last column, check whether we can include the process at j+1
                if j + 1 < num_processes:
                    dp_table[i][j] = max(
                        dp_table[i][j],
                        # Current value
                        # If the current process can handle the next query,
                        # add 1 to the value from the next column
                        dp_table[i][j + 1] + (processes[i] >= queries[dp_table[i][j + 1]])
                    )
                # If we have found a solution for all queries, return the total number of queries
                if dp_table[i][j] == num_queries:
                    return num_queries

        # Iterate over the main diagonal of the dp_table to return the maximum value
        # This indicates how many queries can be processed starting and ending at each process
        return max(dp_table[i][i] * (processes[i] >= queries[dp_table[i][i]]) for i in range(num_processes))
```

Java

```java

class Solution {
    public int maximumProcessableQueries(int[] nums, int[] queries) {
        int n = nums.length; // Total number of nums
        // Create a 2D array to store the results of subproblems.
        int[][] dp = new int[n][n];
        int m = queries.length; // Total number of queries.

        // Build the dp array in bottom-up fashion.
        for (int i = n; i >= 0; --i) {
            for (int j = n - 1; j >= i; --j) {
                // Check and update using the previous row's data if not in the first row.
                if (i > 0) {
                    dp[i][j] = Math.max(
                        dp[i][j], // Current value.
                        dp[i - 1][j] + (nums[i - 1] >= queries[dp[i - 1][j]]) ? 1 : 0)); // Compare with previous row.
                }
                // Check and update using the data from the next column if not in the last column.
                if (j < n - 1) {
                    dp[i][j] = Math.max(
                        dp[i][j], // Current value.
                        dp[i][j + 1] + (nums[i] >= queries[dp[i][j + 1]] ? 1 : 0)); // Compare with next column.
                }
                // If we have found all queries to be processable, return the total count.
                if (dp[i][j] == m) {
                    return m;
                }
            }
        }

        // Variable to hold the maximum number of processable queries observed.
        int maxQueries = 0;
        // Iterate through the diagonal elements of dp array to find the maximum.
        for (int i = 0; i < n; ++i) {
            maxQueries = Math.max(maxQueries, dp[i][i] + (nums[i] >= queries[dp[i][i]] ? 1 : 0));
        }

        // Return the maximum number of processable queries.
        return maxQueries;
    }
}
```

C++

```cpp

class Solution {
public:
    int maximumProcessableQueries(vector<int>& nums, vector<int>& queries) {
        int numElements = nums.size(); // Number of elements in nums
        int dpCountElements[numElements][numElements]; // Dynamic programming table
        memset(dpCountElements, 0, sizeof(dpCountElements)); // Initialize table to 0
        int totalQueries = queries.size(); // Total number of queries

        // Outer loop to handle the start of the subarray
        for (int start = 0; start < numElements; ++start) {
            // Inner loop to handle the end of the subarray, going backwards
            for (int end = numElements - 1; end >= start; --end) {
                // Check if start == 0
                if (start == 0) {
                    // If start == 0, it can process the current query, we increment the count from the previous state
                    dpCountElements[end] = max(dpCountElements[end], dpCountElements[end + 1] + (nums[start] >= queries[dpCountElements[end + 1]]));
                }
                // If end == numElements - 1, it can process the current query, we increment the count from the previous state
                else if (end == numElements - 1) {
                    dpCountElements[end] = max(dpCountElements[end], dpCountElements[end - 1] + (nums[end] >= queries[dpCountElements[end - 1]]));
                }
                // If we have processed all queries, we can return the total number of queries as answer
                if (dpCountElements[end] == numQueries) {
                    return numQueries;
                }
            }
        }

        int answer = 0; // Variable to store the maximum number of queries processed

        // Iterate over all elements to find the maximum number of queries that can be processed
        for (int i = 0; i < numElements; ++i) {
            // Check if the current element can process the next query and
            // update the answer with the maximum value obtained from the DP table
            answer = max(answer, dpCountElements[i][i] + (nums[i] >= queries[dpCountElements[i][i]]));
        }

        return answer; // Return the final answer
    }
};
```

TypeScript

```typescript

function maximumProcessableQueries(nums: number[], queries: number[]): number {
    const numLength = nums.length;
    // Initialize the memoization array with zeros
    const dp: number[][] = Array.from({ length: numLength }, () => new Array(numLength).fill(0));

    const queriesLength = queries.length;

    // Iterate over the range [0, n) for both start and end indices
    for (let start = 0; start < numLength; ++start) {
        for (let end = numLength - 1; end >= start; --end) {
            // If not the first element, update the dp array considering the previous element
            if (start > 0) {
                dp[start][end] = Math.max(
                    dp[start][end],
                    dp[start - 1][end] + (nums[start - 1] >= queries[dp[start - 1][end]] ? 1 : 0),
                );
            }

            // If not the last element, update the dp array considering the next element
            if (end < numLength - 1) {
                dp[start][end] = Math.max(
                    dp[start][end],
                    dp[start][end + 1] + (nums[end + 1] >= queries[dp[start][end + 1]] ? 1 : 0),
                );
            }

            // If all queries are processed, return the query count
            if (dp[start][end] == queriesLength) {
                return queriesLength;
            }
        }
    }

    // Find the maximum number of queries that can be processed from any single position
    let maxQueries = 0;
    for (let index = 0; index < numLength; ++index) {
        maxQueries = Math.max(maxQueries, dp[index][index] + (nums[index] >= queries[dp[index][index]] ? 1 : 0));
    }

    // Return the maximum number of queries that can be processed
    return maxQueries;
}
```

Python3

```python

from typing import List

class Solution:
    def maximumProcessableQueries(self, processes: List[int], queries: List[int]) -> int:
        # Number of processes
        num_processes = len(processes)
        # Initialize a 2D array for the dynamic programming table with zeros
        dp_table = [[0] * num_processes for _ in range(num_processes)]
        # Number of queries
        num_queries = len(queries)

        # Fill in the dp_table for all possible segments [i, j] of processes
        for i in range(num_processes - 1, -1, -1):
            for j in range(num_processes - 1, i - 1, -1):
                # If we're not in the first row, check whether we can include the process at i-1
                if i:
                    dp_table[i][j] = max(
                        dp_table[i][j],
                        # Current value
                        # If the current process can handle the next query,
                        # add 1 to the value from the row above
                        dp_table[i - 1][j] + (processes[i - 1] >= queries[dp_table[i - 1][j]])
                    )
                # If we're not in the last column, check whether we can include the process at j+1
                if j + 1 < num_processes:
                    dp_table[i][j] = max(
                        dp_table[i][j],
                        # Current value
                        # If the current process can handle the next query,
                        # add 1 to the value from the next column
                        dp_table[i][j + 1] + (processes[i] >= queries[dp_table[i][j + 1]])
                    )
                # If we have found a solution for all queries, return the total number of queries
                if dp_table[i][j] == num_queries:
                    return num_queries

        # Iterate over the main diagonal of the dp_table to return the maximum value
        # This indicates how many queries can be processed starting and ending at each process
        return max(dp_table[i][i] * (processes[i] >= queries[dp_table[i][i]]) for i in range(num_processes))
```

Time and Space Complexity

Time Complexity

The time complexity of the code is derived from the nested loops it contains. There are two loops each iterating over the length of `nums`, indexed by `i` and `j`. The outer loop runs `n` times where `n` is the length of `nums`. The inner loop runs `n` times for each iteration of the outer loop, effectively resulting in `n * n` iterations. Within these loops, the code performs a constant amount of work for each iteration, mainly comparisons and assignments which are `O(1)` operations.

Therefore, the time complexity is `O(n^2)`.

Space Complexity

The space complexity of the code is determined by the amount of memory used to store variables and data structures. Here, the 2D array `f` of size `n * n` is the dominant factor. Since it stores `n` subtasks of length `n`, the total space used by this array is proportional to `n^2`. No other data structures in the code use space that is dependent on `n` in a way that would increase the overall space complexity.

Thus, the space complexity is also `O(n^2)`.