2091. Removing Minimum and Maximum From Array

Medium Greedy Array

Problem Description

In this problem, we are given an array `nums` which contains distinct integers. It is 0-indexed, which means the indexing of elements in the array starts with 0. Among all the elements in the `nums` array, there is one element with the lowest value, termed as the *minimum*, and one with the highest value, known as the *maximum*. Our objective is to remove these two elements from the array.

A *deletion* involves removing an element either from the front (beginning) of the array or from the back (end) of the array. The task is to determine the minimum number of such deletions required to remove both the minimum and maximum elements from the array.

To succeed, we need to find a strategy that minimizes the total number of deletions by intelligently deciding whether to start from the front or the back of the array, as well as considering the possibility of removing elements from both ends.

Intuition

When approaching the solution, we focus on the positions (indexes) of the minimum and maximum elements in the array. The main idea hinges upon the realization that the number of deletions needed is influenced by the relative locations of these two elements.

Firstly, we identify the indexes of both the minimum and maximum elements by scanning through the array.

Then, our strategy involves considering three scenarios:

- Remove elements from the front until both the minimum and maximum elements are deleted.
- Remove elements from the back until both elements are deleted.
- Remove some elements from the front to delete one of the elements and some from the back to delete the other one.

To achieve the minimum number of deletions, we calculate the number of deletions needed for each of the scenarios and return the smallest one.

The insight here is to use the indexes of minimum and maximum elements efficiently to minimize the number of operations, as direct removals would be inefficient for large arrays.

Solution Approach

The solution approach for this problem leverages a straightforward scan of the array to determine the positions (indexes) of the minimum and maximum elements.

The following steps are taken in the implementation:

1. Initialize two variables, `mi` and `mx`, to store the indexes of the minimum and maximum elements, respectively. We start by assuming the first element (at index 0) is both the minimum and maximum, hence `mi = mx = 0`.
2. Loop through every element `num` in the `nums` list, keeping track of the current index `i`. During each iteration:
   - If the current element is less than `nums[mi]`, update `mi` with the current index `i` because a new minimum has been found.
   - Similarly, if the current element is greater than `nums[mx]`, update `mx` with the current index `i` as a new maximum has been found.
3. After identifying the indexes of the minimum and maximum elements, ensure that `mi` is less than or equal to `mx` by swapping them if that is not the case. This simplifies the calculation that follows, because we will be dealing with three scenarios represented by contiguous sections of the array to be deleted.
4. Calculate the total number of deletions required for each of the three above-mentioned deletion strategies:
   - Strategy 1: Remove elements from the beginning (front) up to and including the maximum index (`mx`). The number of deletions would be `mx + 1`, since array indexing starts at 0.
   - Strategy 2: Remove elements from the end (back) up to and including the minimum index (`mi`). The number of deletions here would be `len(nums) - mi`.
   - Strategy 3: Remove elements from the front up to and including the minimum index (`mi + 1`) and from the back to the maximum index (`len(nums) - mx`), combining these two gives `mi + 1 + len(nums) - mx`.
5. Return the minimum value out of the three calculated strategies to achieve the minimum number of deletions needed to remove both the minimum and maximum elements from the array. This is done using the built-in `min()` function with the above mentioned three results as arguments.

The Python code is elegant and concise due to its use of list enumeration for index tracking, conditional statements for dynamic updates, and the `min()` function for direct comparison.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Assume we have the array `nums = [3, 2, 5, 1, 4]`.

1. We initialize `mi = mx = 0`. Currently, `nums[mi] = nums[mx] = 3`.
2. We start looping through the elements:
   - `num = 2, i = 1`: `2` is not less than `3`, so `mi` remains unchanged. `2` is not greater than `3`, so `mx` also remains unchanged.
   - `num = 5, i = 2`: `5` is greater than `3`, so we update `mx` to `2`. `5` is not less than `3`, so `mi` remains unchanged.
   - `num = 1, i = 3`: `1` is less than `3`, so we update `mi` to `3`. `1` is not greater than `5`, so `mx` remains `2`.
   - `num = 4, i = 4`: `4` is not less than `1`, and it's not greater than `5`, so both `mi` and `mx` remain unchanged.

After the loop, `mi = 3` (minimum element `1` at index `3`) and `mx = 2` (maximum element `5` at index `2`).

3. Since `mi` is not less than `mx`, we do not swap them.
4. Calculate the number of deletions for each strategy:
   - Strategy 1: Removing from the front to the maximum index `mx` is `mx + 1`, which equals `3` deletions.
   - Strategy 2: Removing from the back to the minimum index `mi` is `len(nums) - mi`, which is `5 - 3`, resulting in `2` deletions.
   - Strategy 3: Removing from the front up to `mi` and from the back beyond `mx` is `mi + 1 + len(nums) - mx`, which is `3 + 1 + 5 - 2`, totaling `7` deletions.
5. Finally, return the minimum value out of the calculated strategies: `min(3, 2, 7)` which equals `2`.

Therefore, the minimum number of deletions required to remove both the minimum and maximum elements from the array `[3, 2, 5, 1, 4]` is `2`. This occurs by removing the last two elements (strategy 2).

Solution Implementation

Python

```python
class Solution:
    def minimumDeletions(self, nums: List[int]) -> int:
        # Initialize indices for the minimum and maximum value elements
        min_index = max_index = 0

        # Iterate over the list to find indices of the minimum and maximum elements
        for i, num in enumerate(nums):
            if num < nums[min_index]:  # A new minimum element found
                min_index = i
            if num > nums[max_index]:  # A new maximum element found
                max_index = i

        # Ensure that min_index is always less than or equal to max_index
        if min_index > max_index:
            min_index, max_index = max_index, min_index

        # Calculate the minimum number of deletions using three strategies:
        # 1. Delete from the front to the max element
        # 2. Delete from the min element to the end of the list
        # 3. Delete from both ends to the min and max elements
        # Then return the strategy that results in the fewest deletions
        return min(
            max_index + 1,  # Deleting from the front to the max element
            len(nums) - min_index,  # Deleting from the min element to the end
            min_index + 1 + len(nums) - max_index  # Deleting from both ends
        )
```

Java

```java
class Solution {
    public int minimumDeletions(int[] nums) {
        // Initialize the variables to store the index of minimum and maximum element in array
        int minIndex = 0;
        int maxIndex = 0;
        // Length of the array
        int n = nums.length;

        // Iterate through the array to find the indices of the minimum and maximum elements
        for (int i = 0; i < n; ++i) {
            // Update the index of the minimum element if a smaller element is found
            if (nums[i] < nums[minIndex]) {
                minIndex = i;
            }
            // Update the index of the maximum element if a larger element is found
            if (nums[i] > nums[maxIndex]) {
                maxIndex = i;
            }
        }

        // Make sure the minIndex is always less than maxIndex for ease of calculation
        if (minIndex > maxIndex) {
            // Swap minIndex with maxIndex
            int temp = maxIndex;
            maxIndex = minIndex;
            minIndex = temp;
        }

        // Calculate the minimum deletions needed, considering three different scenarios:
        // 1. Removing elements from the beginning up to the maxIndex
        // 2. Removing elements from the minIndex to the end
        // 3. Removing elements from both ends surrounding the min and max
        // Return the smallest of these three options
        return Math.min(Math.min(maxIndex + 1, n - minIndex),
                        minIndex + 1 + n - maxIndex);
    }
}
```

C++

```cpp
class Solution {
public:
    int minimumDeletions(vector<int>& nums) {
        int minIndex = 0; // Index of the smallest number
        int maxIndex = 0; // Index of the largest number
        int n = nums.size(); // The total number of elements in nums

        // Loop to find indices of the smallest and largest numbers
        for (int i = 0; i < n; ++i) {
            if (nums[i] < nums[minIndex]) { // Check for new minimum
                minIndex = i;
            }
            if (nums[i] > nums[maxIndex]) { // Check for new maximum
                maxIndex = i;
            }
        }

        // Ensure minIndex is smaller than maxIndex for easier calculation
        if (minIndex > maxIndex) {
            int temp = minIndex;
            minIndex = maxIndex;
            maxIndex = temp;
        }

        // Three possible ways to remove the min and max, take the minimum deletions needed
        return min(
            min(
                maxIndex + 1, // Delete all from start to maxIndex, inclusive
                n - minIndex // Delete all from minIndex to end
            ),
            minIndex + 1 + n - maxIndex // Delete from start to minIndex and from maxIndex to end
        );
    }
};
```

TypeScript

```typescript
function minimumDeletions(nums: number[]): number {
    // Find the number of elements in the array
    const numElements = nums.length;

    // If the array contains only one element, one deletion is required
    if (numElements === 1) return 1;

    // Find the indices of the minimum and maximum elements in the array
    let minIndex = nums.indexOf(Math.min(...nums));
    let maxIndex = nums.indexOf(Math.max(...nums));

    // Calculate the leftmost and rightmost positions among the min and max indices
    let leftIndex = Math.min(minIndex, maxIndex);
    let rightIndex = Math.max(minIndex, maxIndex);

    // Calculate the possible numbers of deletions
    // if removing from the left towards the right
    let deleteLeftRight = leftIndex + 1 + numElements - rightIndex;
    // if removing from the left only
    let deleteLeftOnly = rightIndex + 1;
    // if removing from the right only
    let deleteRightOnly = numElements - leftIndex;

    // Return the minimum number of deletions amongst the calculated options
    return Math.min(deleteLeftRight, deleteLeftOnly, deleteRightOnly);
}
```

```python
class Solution:
    def minimumDeletions(self, nums: List[int]) -> int:
        # Initialize indices for the minimum and maximum value elements
        min_index = max_index = 0

        # Iterate over the list to find indices of the minimum and maximum elements
        for i, num in enumerate(nums):
            if num < nums[min_index]:  # A new minimum element found
                min_index = i
            if num > nums[max_index]:  # A new maximum element found
                max_index = i

        # Ensure that min_index is always less than or equal to max_index
        if min_index > max_index:
            min_index, max_index = max_index, min_index

        # Calculate the minimum number of deletions using three strategies:
        # 1. Delete from the front to the max element
        # 2. Delete from the min element to the end of the list
        # 3. Delete from both ends to the min and max elements
        # Then return the strategy that results in the fewest deletions
        return min(
            max_index + 1,  # Deleting from the front to the max element
            len(nums) - min_index,  # Deleting from the min element to the end
            min_index + 1 + len(nums) - max_index  # Deleting from both ends
        )
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input list `nums`. This is because the code consists of a single loop that traverses the entire list to find the minimum and maximum elements' indices, which takes linear time in the size of the input.

The space complexity of the code is `O(1)`. The extra space used by the code includes only a fixed number of integer variables `mi` (minimum index) and `mx` (maximum index), which do not depend on the input size, therefore the space used is constant.