2856. Minimum Array Length After Pair Removals

Problem Description

You are provided with a sorted array of integers `nums`, where the sorting is in non-decreasing order. The problem presents an operation that can be performed repeatedly, as many times as you wish. Each operation involves picking two distinct indices `i` and `j` (where `i < j`) such that `nums[i] < nums[j]`, and then removing the elements at both of these indices from the array. The goal is to determine the minimum length of the array `nums` after doing this operation as many times as possible (you could also opt to not perform any operations).

Intuition

The solution to this problem involves recognizing that you can always remove pairs of distinct numbers due to the array's sorted nature. Since the array is sorted, `nums[i] < nums[j]` for any `i < j` where the elements are different.

A key observation here is that when you remove a pair of numbers, only two elements are removed. Therefore, to reach the minimum length of the array, you want to maximize the number of such removal operations. However, you need to ensure that you're pairing elements effectively.

How do we maximize the number of removals? Let's consider the frequency of each number in the sorted array. If you have a number that appears, say, three times, you can only form one complete pair out of it, leaving one instance of the number unpaired. It's clear that each unpaired number will remain in the final array, adding to its length.

The intuition behind the provided Python solution involves using a max-heap to efficiently pick out the two numbers with the highest frequencies for the pairing—each removal of a pair will decrease the count of these numbers by one. By continuously pairing off the numbers with the highest frequency (and hence the highest chance of having unpaired elements left over), we ensure that a minimal number of elements remain unpaired.

After all possible pairings, you are left with either an empty heap (all elements have been paired off), or a heap with one element remaining (an element that couldn't be paired). If the heap is not empty, the final count of the remaining element contributes to the final size of the array post-operations.

The answer is thus the initial length of the array minus twice the number of successful pairings (since two elements are removed for each pairing).

Solution Approach

The solution follows a greedy approach using a priority queue (max-heap) to perform operations that effectively pair off elements. Here's how the algorithm is implemented step-by-step:

1. **Frequency Count**: First, it counts the frequency of each unique number in the `nums` array using a `Counter` from Python's `collections` module. This is because the frequency of numbers will determine how many pairs can be formed. Numbers with a frequency of 1 cannot be paired, while those with higher frequencies can be paired multiple times.
2. **Convert to Max-Heap**: It then converts the frequency counts into a max-heap using the `-x` technique because Python's `heapq` module provides a min-heap by default. By negating the values, the solution turns it into a max-heap for our purpose. The `heapq.heapify` function is used to transform the list of frequencies into a heap, which now lets us easily access the two largest frequencies.
3. **Pair Up Elements**: The algorithm then enters a loop that runs as long as there is more than one element left in the max-heap. In each iteration, it pops the two largest elements (which are the most frequent numbers in the array), decrements their counts to reflect that a pair has been formed and removes them from the array, and then pushes the decremented frequencies back into the heap if they are greater than zero. This ensures that each time we are attempting to pair off the most frequent numbers first, gradually decreasing their frequencies and thus maximizing the array size by two elements for each pair.
4. **Decrease Array Length**: After successfully pairing two elements, the solution decreases the running total length of the array `ans` by 2 since that's how many elements are removed by the performed operation.
5. **Remaining Elements**: Once the loop finishes, the resulting length (`ans`) represents the minimum length of the array after performing the maximum number of valid operations. If one element is left in the heap, it cannot be paired and will remain in the final array.

Example Walkthrough

Let's consider a small array `nums = [1,2,2,3,3,3,4,4,4,4]` to illustrate the solution approach.

1. **Frequency Count**: We first count the frequency of each unique number in the array.
   - Frequency of `1` is `1`
   - Frequency of `2` is `2`
   - Frequency of `3` is `3`
   - Frequency of `4` is `4`
2. **Convert to Max-Heap**: Convert these frequencies to a max-heap using negative values (to implement the max-heap with Python's min-heap structure).
   - Max-heap would be represented as `[-4, -3, -2, -1]`
3. **Pair Up Elements**: Now, we enter a loop to start pairing elements.
   - We pop the two largest elements from the heap, which are `-4` and `-3` (representing `4` and `3` with frequencies `4` and `3`, respectively).
   - After pairing them, their frequencies decrease by `1`. The heap now has `[-3, -2, -1, -2]` (representing `4, 2, 1, 3`).
   - The array length is reduced by `2`, going from `10` to `8`.
   - Pair the next two largest elements, which are `-3` (representing `4`) and `-2` (representing `3`).
   - After pairing them, the heap is now `[-2, -1, -2, -1]` (representing `4, 1, 3, 3`).
   - The array length is reduced by another `2`, now `6`.
   - This process continues, pairing the two largest elements each time and updating the heap.
   - The final operations will be unable to pair the last frequency of `1` (meaning we end with one element having a frequency of `1`), which means the array cannot be reduced further.
4. **Decrease Array Length**: Each pairing operation decreases the `ans` variable by `2`. After the first operation, `ans = 8`; after the second, `ans = 6`, and so on.
5. **Remaining Elements**: If we are left with an element that cannot be paired, it adds to the final length of the array. In this example, we would end with one unpaired element (frequency `1`), which contributes to the final array size.

So, if we start with an array of length `10`, after performing the pairings, we could end up with a final array length that is the original length minus twice the number of operations plus any remaining unpaired elements. In this case, we might end up with either an empty array (if all elements can be paired) or an array of length `1` (if one unpaired element remains).

Python Solution

```python
from heapq import heapify, heappop, heappush
from collections import Counter
from typing import List

class Solution:
    def minLengthAfterRemovals(self, nums: List[int]) -> int:
        # Create a Counter object to count the frequency of each number in the input list
        frequency_counter = Counter(nums)

        # Create a max heap to store the negative counts (since Python's heapq is a min heap by default)
        max_heap = [-count for count in frequency_counter.values()]
        heapify(max_heap)

        # Initialize the answer as the length of the input list
        answer_length = len(nums)

        # Process the heap while there are at least two elements in the heap
        while len(max_heap) > 1:
            # Pop the two largest elements (most frequent numbers); invert them back to positive
            count1, count2 = -heappop(max_heap), -heappop(max_heap)

            # Decrease their counts (simulate removal)
            count1 -= 1
            count2 -= 1

            # If there are any remaining counts for the number, push it back into the heap
            if count1 > 0:
                heappush(max_heap, -count1)
            if count2 > 0:
                heappush(max_heap, -count2)

            # Since we remove 2 elements, decrease the total length of the array by 2, updating the answer
            answer_length -= 2

        # Return the final answer
        return answer_length
```

Java Solution

```java
class Solution {
    public int minLengthAfterRemovals(List<Integer> numbers) {
        // A map to store the frequency count of each unique number
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int number : numbers) {
            // Increment the count for each number in the map
            frequencyMap.merge(number, 1, Integer::sum);
        }

        // PriorityQueue to store frequencies in descending order (largest frequency at the top)
        PriorityQueue<Integer> frequencyQueue = new PriorityQueue<>(Comparator.reverseOrder());

        // Add all frequencies to the PriorityQueue
        for (int frequency : frequencyMap.values()) {
            frequencyQueue.offer(frequency);
        }

        // Initialize the answer with the total count of numbers in the list
        int minLength = numbers.size();

        // Process pairs until we only have one frequency or none left
        while (frequencyQueue.size() > 1) {
            // Pull the two top frequencies
            int firstFrequency = frequencyQueue.poll();
            int secondFrequency = frequencyQueue.poll();

            // Decrement the frequencies as we are pairing and removing one occurrence of each
            firstFrequency--;
            secondFrequency--;

            // If any updated frequencies are greater than 0, offer them back to the priority queue
            if (firstFrequency > 0) {
                frequencyQueue.offer(firstFrequency);
            }
            if (secondFrequency > 0) {
                frequencyQueue.offer(secondFrequency);
            }

            // Each removal reduces the length by 2 since we remove one occurrence of each of the two numbers
            minLength -= 2;
        }

        // Return the minimum length after making all possible removals
        return minLength;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <unordered_map>

class Solution {
public:
    int minLengthAfterRemovals(vector<int>& nums) {
        // Create a hashmap to count the occurrence of each number in the vector.
        unordered_map<int, int> countMap;
        for (int num : nums) {
            ++countMap[num]; // Increment the count for this number.
        }

        // Use a max-heap (priority queue) to keep track of the counts.
        priority_queue<int> maxHeap;
        for (auto& keyValue : countMap) {
            maxHeap.push(keyValue.second); // Push counts to the max-heap.
        }

        // Initialize final answer to the size of the input vector.
        int finalLength = nums.size();

        // Process the heap until there is one or no element left.
        while (maxHeap.size() > 1) {
            int topCount1 = maxHeap.top(); // Get the top (maximum) element from the max-heap.
            maxHeap.pop(); // Remove that element.
            int topCount2 = maxHeap.top(); // Get the next top element.
            maxHeap.pop(); // Remove that element as well.

            // Decrement both counts as we are planning to remove one occurrence of each.
            topCount1--;
            topCount2--;

            // If they are still occurrences left after removal, push the new count back to the heap.
            if (topCount1 > 0) {
                maxHeap.push(topCount1);
            }
            if (topCount2 > 0) {
                maxHeap.push(topCount2);
            }

            // Since we remove two elements from the array (one each for the two counts), decrease the length by 2.
            finalLength -= 2;
        }

        // Return the final length of the vector after the removals.
        return finalLength;
    }
};
```

Typescript Solution

```typescript
import { MaxPriorityQueue } from 'datastructures-js/priority-queue';

// This function calculates the minimum length of the array after performing
// operations where pairs of the same number are removed from the same.
function minLengthAfterRemovals(nums: number[]): number {
    // A map to store the frequency count of each number in the array.
    const frequencyCount: Map<number, number> = new Map();

    // Count the frequency of each number in the input array.
    for (const num of nums) {
        frequencyCount.set(num, (frequencyCount.get(num) ?? 0) + 1);
    }

    // A priority queue to order the counts in decreasing order.
    const priorityQueue = new MaxPriorityQueue<number>();

    // Enqueue the frequency counts into the priority queue.
    for (const [, count] of frequencyCount) {
        priorityQueue.enqueue(count);
    }

    // Variable to track the answer which is initialized to the length of the input array.
    let minimumLength = nums.length;

    // Keep removing elements in pairs until one or no elements remain in the priority queue.
    while (priorityQueue.size() > 1) {
        // Get counts of the two most frequent elements.
        let count1 = priorityQueue.dequeue().element;
        let count2 = priorityQueue.dequeue().element;

        // Decrease the counts and re-enqueue if they are still greater than zero.
        if (--count1 > 0) {
            priorityQueue.enqueue(count1);
        }
        if (--count2 > 0) {
            priorityQueue.enqueue(count2);
        }

        // Each valid removal decreases the minimum length by 2.
        minimumLength -= 2;
    }

    // Return the min length of the array after removals.
    return minimumLength;
}

// You will need to install `datastructures-js/priority-queue` package to use MaxPriorityQueue.
// If you're using npm, you can install with `npm install @datastructures-js/priority-queue`
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by several factors:

1. Counting the elements in `nums` using `Counter` constructor has a time complexity of $O(n)$, where n is the number of elements in `nums`.
2. The list comprehension used to create `pq` from the counts also has a time complexity of $O(m)$, assuming there are m unique elements. This is because it traverses the counter once.
3. Constructing a heap from the list of counts using `heapify` has a time complexity of $O(m)$.
4. The `while` loop runs until there is only one or no element left in the priority queue. In the worst case, the loop runs for $n/2$ iterations if every element is unique. The total number of heappop and heappush operations will together contribute to the time complexity.
   - For each heappop operation, the time complexity is $O(\log m)$.
   - For each heappush operation after decrementing the element, if it is still greater than 0, the time complexity is also $O(\log m)$.

Taking into account that each iteration may involve up to two heappop and two heappush operations, the total complexity for the loop is $O(n \log n)$ for $n/2$ iterations, leading to an overall complexity of $O(n \log n)$.

Considering the complexities together, the dominant term is $O(n \log n)$ due to the while loop. Hence, the overall time complexity of the code is $O(n \log n)$.

Space Complexity

The space complexity is determined by the additional storage used by the code:

1. The `Counter` data structure requires $O(m)$ space, where m is the number of unique elements in `nums`.
2. The priority queue (heap) also requires $O(m)$ space.
3. No other significant space is used since the operations inside the loop use a constant amount of space.

Thus, the overall space complexity of the code is $O(m)$, where m represents the number of unique elements in `nums`.