3020. Find the Maximum Number of Elements in Subset

Medium Array Hash Table Enumeration

Problem Description

You are given an array called `nums` which only contains positive integers. Your goal is to create the longest possible subset from `nums` that can fit a specific pattern when arranged in a 0-indexed array. The pattern is such that it forms a palindrome with squares of the base number increasing until the middle of the array, and then decreasing symmetrically. For example, a valid pattern is `[x, x^2, x^4, ..., x^(k/2), x^k, x^(k/2), ..., x^4, x^2, x]`, where `x` is a base number and `k` is a non-negative power of `2`.

For example, the arrays `[2, 4, 16, 4, 2]` and `[3, 9, 3]` meet the criteria for the pattern (palindromic and squares of x), while `[2, 4, 8, 4, 2]` does not because `8` is not a repeated square of `2` within a palindromic structure. Your task is to return the maximum number of elements in such a subset that meets the described conditions.

Intuition

To solve this problem, we need to analyze the properties of the required pattern. We can see that we can always have at most one number that is `1` and can be squared multiple times to contribute to the length of our pattern (for example, one instance of `2`, two instances of `4`, four instances of `16`, and so forth). Numbers like `1` can be directly repeated since `1^2` is still `1`.

The main observation is that in the valid subset, other than `1`, each number can only appear once, and its square can appear at most once too (if present in the original array), forming pairs that will contribute to the pattern symmetrically.

Given such a pattern with a non-repeating core and symmetric structure, a hash table (dictionary in Python) can help us to count the occurrences of each number. By traversing the array and squaring the numbers while they still appear more than once, we determine the maximum length of the subset that can be created.

Solution Approach

The solution adopts a hash table to keep track of the occurrences of each element in the array `nums`. In Python, this is done using the `Counter` class from the `collections` module, which creates a dictionary-like object that maps elements to their respective counts.

The algorithm proceeds as follows:

- Initialize a counter `cnt` with the counts of each number in `nums`.
- Start with calculating the contribution of the number `1`. Since ones can be freely placed in the pattern without squaring, you can take all of them if there's an odd count, or one less if there's an even count, to ensure symmetry. This is handled by `ans = cnt[1] - (cnt[1] % 2 ^ 1)`.
- Remove `1` from the counter as it has been handled separately, `del cnt[1]`.

Next, for each unique number `x` in the counter that is not `1`:

- A temporary variable `t` is introduced to keep track of the contribution of a number to the subset pattern's length.
- While the count of `x` is more than `1`, square `x` and increase `t` by `2`. This represents adding both `x` and its square to the subset pattern.
- After exiting the while loop, if there is still one instance of `x` left, increment `t` by `1` because it can be used as the central element of the pattern. Otherwise, decrement `t` by `1` as you've gone one step too far (you've made an even number of selections when you can only have an odd number).
- Update the answer `ans` with the maximum of the current answer and the tentative contribution `t`.

The logic of squaring the numbers and checking their counts reflects the idea that numbers in our desired subset can appear either as a single center of the pattern or as pairs flanking the center. Since we are to find the maximum number of elements that can be included, we persistently square the elements and add to our count as long as they can appear in pairs.

After iterating over all numbers in the hash table, we end up with the `ans` variable holding the maximum possible number of elements that comprise a valid pattern. The result is then returned as the final answer.

Example Walkthrough

Suppose we are given the array `nums = [1, 1, 1, 2, 2, 3, 3, 3, 4, 16, 4]`.

Following the solution approach:

1. Initialize the counter `cnt` with counts of each number.

```
cnt = Counter({1: 3, 2: 2, 3: 3, 4: 2, 16: 1})
```

2. Calculate the contribution of number `1`:

```
ans = 3 - (3 % 2 ^ 1) = 3 - 0 = 3
```

   We can use all three `1`s as they are odd in count and can form a symmetric pattern around the center.

3. Remove `1` from the counter since it's been handled.

```
del cnt[1]
// cnt = {2: 2, 3: 3, 4: 2, 16: 1}
```

4. Process other numbers starting from the smallest.
   - For `2` in `nums`:
     - `t` is initialized to `0`.
     - `x` is now `2`, and we have `2` twos.
     - We cannot square `2` to find another `2` in the array, so the contribution of `2` to `t` remains `0`.
   - For `3` in `nums`:
     - `t` is initialized to `0`.
     - We have three `3`s, but we can't square `3` to find another `3` in the array. However, there is an odd count, so we can use one of them as the center. Hence, `t` is increased by `1`.
   - For `4` in `nums`:
     - `t` is initialized to `0`.
     - We have two `4`s. We can square `4`, and indeed there is a `16` in the array, which is `4` squared. This means we can add two `4`s, and one `16` to the pattern, increasing `t` by `2` for the `4`s, and by `1` for the `16` as the center. Thus, `t` is increased by `3`.
     - Update `ans` with the max of current `ans` and `t`:

```
ans = max(3, 0 + 3) = 3
```

   So, our maximum length is `3` from the `3`s and `3` from the `4`, `16`, `4`.

5. The algorithm has traversed all numbers in `cnt` and now `ans` holds the maximum possible number of elements, which is `6`.

Therefore, for `nums = [1, 1, 1, 2, 2, 3, 3, 3, 4, 16, 4]`, the subset `[1, 1, 1, 4, 16, 4]` reflects the longest subset that can fit the pattern with a length of `6`.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def maximumLength(self, nums: List[int]) -> int:
        # Create a counter object for the occurrences of each number in `nums`
        num_counter = Counter(nums)

        # Initialize `answer` to the number of 1's in the list, adjusting for odd counts by subtracting 1 if count of 1's is odd
        answer = num_counter[1] - (num_counter[1] % 2 ^ 1)

        # Remove the count for 1's, since we have processed it already
        del num_counter[1]

        # Iterate over the items in the counter
        for number in num_counter:
            temp_max_length = 0  # Initialize the temporary max sequence length

            # While there are at least two occurrences of the number, we can make a square
            while num_counter[number] > 1:
                number *= number  # Square the number
                temp_max_length += 2  # Increase the length of the sequence by 2 for each pairing

            # Add 1 to the length if there is a single occurrence left, or subtract 1 if there was none
            temp_max_length += 1 if num_counter[number] else -1

            # Update the running maximum length answer with the maximum of current and temporary max length
            answer = max(answer, temp_max_length)

        # Return the calculated maximum length
        return answer
```

Java

```java
class Solution {
    public int maximumLength(int[] nums) {
        // Create a map to store the occurrence count of each number.
        Map<Integer, Integer> countMap = new HashMap<>();
        // Populate the map with the occurrence count.
        for (int num : nums) {
            countMap.merge((long) num, 1, Integer::sum);
        }

        // Remove the count of 1s from the map and handle it specially.
        // Ensure that if present, even number of 1s are counted towards the maximum length.
        Integer countOfOnes = countMap.remove(1L);
        int maxLen = (countOfOnes == null) ? 0 : countOfOnes - (countOfOnes % 2 ^ 1);

        // Iterate through the remaining numbers in the count map.
        for (long num : countMap.keySet()) {
            int localMaxLen = 0;
            // While the count is more than one occurrence of the number, square it and increment the local max length by 2.
            while (countMap.getOrDefault(num, 0) > 1) {
                num *= num;
                localMaxLen += 2;
            }
            // Add to the local max length the count of the squared number, if it exists, otherwise subtract one.
            localMaxLen += countMap.getOrDefault(num, -1);
            // Update the global max length if the local max length is greater.
            maxLen = Math.max(maxLen, localMaxLen);
        }
        // Return the maximum length found.
        return maxLen;
    }
}
```

C++

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

class Solution {
public:
    int maximumLength(std::vector<int>& nums) {
        // Creating a hash map to store the frequency of each number.
        std::unordered_map<long long, int> frequencyMap;

        // Counting the frequency of each number in the nums vector.
        for (int num : nums) {
            ++frequencyMap[num];
        }

        // Initializing the maximum length with the frequency of 1 subtracted by 1
        // if the count is odd (to make it even), as we're looking for pairs.
        int maxLength = frequencyMap[1] - (frequencyMap[1] % 2 ^ 1);

        // Erasing the count of 1 from the map to not consider it again.
        frequencyMap.erase(1);

        // Iterating through each unique number in the map (excluding 1).
        for (auto& entry : frequencyMap) {
            // Temporary variable to keep track of the sequence length.
            int sequenceLength = 0;
            // We'll be squaring the number to check for powers of the original number
            // (e.g., 2, 4, 16 for original number 2).
            long long currentValue = entry.first;

            // While the current value is in the map and its frequency is more than 1,
            // it means we have at least a pair, and we can form a longer sequence.
            while (frequencyMap.count(currentValue) && frequencyMap[currentValue] > 1) {
                currentValue *= currentValue; // Square the number.
                sequenceLength += 2; // Increment the sequence length by 2 (for a pair).
            }

            // If the current value still exists in the map, add 1 more to sequence length.
            // Otherwise, subtract 1 to remove the extra value added when there's no pair.
            sequenceLength += frequencyMap.count(currentValue) ? 1 : -1;

            // Set maxLength to the maximum value between its current value and
            // sequenceLength for the current number.
            maxLength = std::max(maxLength, sequenceLength);
        }

        // Return the maximum sequence length calculated.
        return maxLength;
    }
};
```

TypeScript

```typescript
function maximumLength(nums: number[]): number {
    // Create a map to store the frequency of each number in the array
    const frequencyMap: Map<number, number> = new Map();

    // Populate the frequency map with the numbers from the nums array
    for (const num of nums) {
        frequencyMap.set(num, (frequencyMap.get(num) ?? 0) + 1);
    }

    // Initialize ans with the frequency of the number 1 if it exists, adjusting for parity
    let maxCountSequence = frequencyMap.has(1) ? frequencyMap.get(1)! - (frequencyMap.get(1)! % 2 ^ 1) : 0;
    frequencyMap.delete(1); // Remove 1 from the map as it is handled separately

    // Iterate over the remaining entries in the frequency map
    for (let [base, _] of frequencyMap) {
        let sequenceCount = 0;

        // While there are at least 2 occurrences of the base, multiply it by itself (square it)
        // and increment the sequence count by 2
        while (frequencyMap.has(base) && frequencyMap.get(base)! > 1) {
            base = base * base;
            sequenceCount += 2;
        }

        // If there is an occurrence of the squared number, increment sequence count by 1
        // otherwise, decrement it by 1
        sequenceCount += frequencyMap.has(base) ? 1 : -1;

        // Update the maxCountSequence to the maximum of current maxCountSequence and sequenceCount
        maxCountSequence = Math.max(maxCountSequence, sequenceCount);
    }

    // Return the length of the longest sequence found
    return maxCountSequence;
}
```

```python
from collections import Counter

class Solution:
    def maximumLength(self, nums: List[int]) -> int:
        # Create a counter object for the occurrences of each number in `nums`
        num_counter = Counter(nums)

        # Initialize `answer` to the number of 1's in the list, adjusting for odd counts by subtracting 1 if count of 1's is odd
        answer = num_counter[1] - (num_counter[1] % 2 ^ 1)

        # Remove the count for 1's, since we have processed it already
        del num_counter[1]

        # Iterate over the items in the counter
        for number in num_counter:
            temp_max_length = 0  # Initialize the temporary max sequence length

            # While there are at least two occurrences of the number, we can make a square
            while num_counter[number] > 1:
                number *= number  # Square the number
                temp_max_length += 2  # Increase the length of the sequence by 2 for each pairing

            # Add 1 to the length if there is a single occurrence left, or subtract 1 if there was none
            temp_max_length += 1 if num_counter[number] else -1

            # Update the running maximum length answer with the maximum of current and temporary max length
            answer = max(answer, temp_max_length)

        # Return the calculated maximum length
        return answer
```

Time and Space Complexity

The time complexity of the given code is indeed $O(n * \log \log M)$ where n is the length of the `nums` array and M is the maximum value in the `nums` array. This is because we iterate through all elements ($O(n)$) to populate the `Counter`, and for each unique number x that is not 1, we may square it repeatedly until it's no longer found in `cnt`. Squaring a number repeatedly like this takes $O(\log \log x)$ time for each unique x, assuming x is the maximum value M. Since the squaring process is bounded by the maximum value M, and since it is done for each unique element that is not 1, the complexity is $O(n * \log \log M)$.

The space complexity is $O(n)$ primarily due to the storage required for the `Counter` object, which would, in the worst case, store as many elements as are present in the array `nums` if all elements are unique.