2354. Number of Excellent Pairs

Hard | Bit Manipulation | Array | Hash Table | Binary Search | Leetcode Link

Problem Description

In this problem, we're presented with an array `nums` consisting of 0-indexed positive integers and a positive integer `k`. Our goal is to find the number of distinct excellent pairs in the array, where a pair `(num1, num2)` is considered excellent if it satisfies two conditions:

1. Both `num1` and `num2` exist in `nums`.
2. The sum of the number of set bits (bits with value 1) in `num1 OR num2` and `num1 AND num2` must be greater than or equal to `k`.

We count the number of set bits using bitwise `OR` and `AND` operations, and we are looking for pairs that collectively have a large enough number of set bits to meet or exceed the threshold `k`.

Also, it's important to note that a pair where `num1` is equal to `num2` can also be considered excellent if the number of its set bits is sufficient and at least one occurrence of the number exists in the array. Moreover, we want the count of distinct excellent pairs, so the order matters—`(a, b)` is considered different from `(b, a)`.

Intuition

To find the number of excellent pairs efficiently, we need to think about the problem in terms of set bits because the conditions for an excellent pair depend on the sum of set bits in `num1 OR num2` and `num1 AND num2`.

One intuitive approach is to use the properties of bitwise operations. For any two numbers, `num1 OR num2` will have the highest possible set bit count of the two numbers because `OR` operation results in a 1 for each bit that is 1 in either `num1` or `num2`. On the other hand, `num1 AND num2` will have set bits only in positions where both `num1` and `num2` have set bits.

Since only the set bits matter, and we're looking for pairs `(num1, num2)` that meet a certain combined set bit count, we can reduce the complexity by avoiding the direct computation of `num1 OR num2` and `num1 AND num2` for all pairs. Instead, we preprocess by counting the set bits for each unique number in `nums`.

To ensure that we count each distinct pair only once, we eliminate duplicates in the array by converting it into a set. We then create a counter to keep track of how many numbers have a specific set bit count. This preprocessing step simplifies our task to just combining counts from the counter, avoiding the need to directly compute the set bit sum for every possible pair.

When we iterate over the unique numbers in `nums` set, we determine the number of set bits for each unique value using `bit_count()`. We then iterate through our set bit count counter and add the count of numbers that have enough set bits to complement the current number's set bit count to reach at least `k`. This way, we find all the pairs that, when combined, meet or exceed the threshold `k`.

By adding the counts for all such complementing set bit count pairs, we calculate the total number of excellent pairs, bypassing the problem of having to generate and check every possible pair, which would be computationally inefficient.

Solution Approach

The given solution employs a combination of bit manipulation and hash mapping to efficiently compute the number of excellent pairs. Let's break down the implementation step-by-step:

1. **Eliminating Duplicates**: The solution begins by converting the original list of numbers into a set. This serves two purposes:
   - Ensures that each number is considered only once, thereby eliminating redundant pairs like `(num1, num1)` when `num1` appears multiple times in the input list.
   - Helps later in avoiding double counting of excellent pairs with the same numbers but in different positions (since `(a, b)` and `(b, a)` are distinct).

```python
s = set(nums)
```

2. **Counting Set Bits**: Then, the algorithm uses a `Counter` from the `collections` module to keep track of how many numbers share the same set bit count.

```python
cnt = Counter()
for v in s:
    cnt[v.bit_count()] += 1
```

   The method `bit_count()` is used to determine the number of set bits in each number. These counts are stored such that `cnt[i]` reflects the number of unique numbers with exactly `i` set bits.

3. **Finding Excellent Pairs**: The core of the solution is to find all the pairs that satisfy the condition of having a combined set bit sum (via bitwise `OR` and `AND`) greater than or equal to `k`. Since an `OR` operation can never reduce the number of set bits, and an `AND` operation can only produce set bits that are already set in both numbers, the combined set bit count for a pair `(num1, num2)` will be equivalent to `bit_count(num1) + bit_count(num2)`.

   The solution iterates over each unique value `v` from the set of numbers and then checks for every bit count `i` stored in `cnt` if the sum of `bit_count(v)` and `i` is greater than or equal to `k`.

```python
ans = 0
for v in s:
    t = v.bit_count()
    for i, x in cnt.items():
        if t + i >= k:
            ans += x
```

   - `t` holds the number of set bits for the current unique number `v`.
   - If `t + i` is greater than or equal to `k`, all numbers with a set bit count `i` can form an excellent pair with `v`. The count of such numbers is `x`, and we add this to our total count of excellent pairs (`ans`). This step leverages the precomputed counts of unique set bit numbers to quickly find the number of complements needed to meet the set bit threshold `k`.

4. **Returning the result**: Finally, after iterating through all unique numbers and their possible pairings, the solution returns `ans`, which holds the total number of distinct excellent pairs.

```python
return ans
```

In summary, the solution follows a smart preprocessing step to calculate and use set bit counts for optimization. This eliminates redundant operations and directly navigates to the crux of the problem, which significantly improves the computational efficiency.

By employing a hash map to store unique set bit counts and identify complementing pairs, the solution reduces what would be a quadratic-time complexity task to a complexity proportional to the product of unique numbers and unique set bit counts present in the input.

Example Walkthrough

Let's say we have an array `nums = [3, 1, 2, 2]` and `k = 3`.

First, we'll apply step 1 and eliminate duplicates by converting `nums` into a set, which will give us `s = {1, 2, 3}`.

Next, we count set bits in step 2. Every number will be processed as follows:

- 1 has 1 set bit.
- 2 has 1 set bit.
- 3 has 2 set bits.

The `Counter` object `cnt` turns out to be `{1: 2, 2: 1}`, indicating that there are 2 numbers with 1 set bit and 1 number with 2 set bits.

Now for step 3, we find excellent pairs. We go through each number in `s` and compare the set bit count with our `k` value:

- For `v = 1` with 1 set bit, `cnt[1] = 2`. Since `1 (set bits of v) + 1 (set bits of another number)` is not `>=` `k`, no excellent pairs are formed with `v = 1`.
- For `v = 2` also with 1 set bit, the situation is the same as for `v = 1`.
- For `v = 3` with 2 set bits, we compare it against `cnt` entries.
  - `2 (set bits of v) + 1 (set bits of another number)` is `>=` `k`, thus excellent pairs are `(3, 1)` and `(3, 2)`. Since there are 2 numbers with 1 set bit, we add `2` to `ans`, resulting in `ans = 2` so far.

Finally, since the pairs `(1, 3)` and `(2, 3)` are also excellent pairs (order matters), we count them again. This adds another `2` to `ans`, making `ans = 4`.

After processing all the unique numbers, we follow step 4 and conclude that there are `4` distinct excellent pairs. The pairs that satisfy the conditions are `(3, 1)`, `(3, 2)`, `(1, 3)`, `(2, 3)`.

This example demonstrates the solution's efficiency, as it avoids checking all possible combinations of `nums` and directly focuses on the set bit counts to find excellent pairs, which is computationally faster.

Python Solution

```python
from collections import Counter

class Solution:
    def countExcellentPairs(self, nums: List[int], k: int) -> int:
        # Use a set to eliminate duplicates as each number contributes uniquely
        unique_nums = set(nums)

        # Counter to store the frequency of bit counts
        bit_count_freq = Counter()

        # Count the frequency of the bit count for each number
        for num in unique_nums:
            bit_count_freq[num.bit_count()] += 1

        excellent_pairs_count = 0  # Initialize the count of excellent pairs

        # Calculate the number of excellent pairs
        for num in unique_nums:
            current_bit_count = num.bit_count()  # Get the bit count of current number

            # Iterate over the bit count frequencies
            for bit_count, freq in bit_count_freq.items():
                # If the sum of bit counts is greater than or equal to k, add to the count
                if current_bit_count + bit_count >= k:
                    excellent_pairs_count += freq

        # Return the total count of excellent pairs
        return excellent_pairs_count

```

Java Solution

```java
class Solution {

    // Method to count the number of excellent pairs
    public long countExcellentPairs(int[] nums, int k) {

        // Use a set to eliminate duplicate values from 'nums'
        Set<Integer> uniqueNumbers = new HashSet<>();
        for (int num : nums) {
            uniqueNumbers.add(num);
        }

        long totalPairs = 0;             // To store the total number of excellent pairs
        int[] bitCounts = new int[32];   // Array to store how many numbers have a certain bit count

        // Count the occurrence of each bit count for the unique elements
        for (int num : uniqueNumbers) {
            int bits = Integer.bitCount(num);  // Count the 1-bits in the binary representation of 'num'
            ++bitCounts[bits];                 // Increase the count for this number of 1-bits
        }

        // Iterate over the unique numbers to find pairs
        for (int num : uniqueNumbers) {
            int bits = Integer.bitCount(num); // Count the 1-bits for this number

            // Check for each possible bit count that could form an excellent pair with 'num'
            for (int i = 0; i < 32; ++i) {
                // Check if the sum of 1-bits is at least 'k'
                if (bits + i >= k) {
                    totalPairs += bitCounts[i]; // If it is, add the count of numbers with 'i' bits to the total
                }
            }
        }

        return totalPairs; // Return the total count of excellent pairs
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>

class Solution {
public:
    long long countExcellentPairs(std::vector<int>& nums, int k) {
        // Create a set to eliminate duplicates from the input vector
        std::unordered_set<int> unique_numbers(nums.begin(), nums.end());

        // Array to count the frequency of set bits (1s) in the binary representation of numbers
        // There are at most 32 bits in an int, so we create an array of size 32
        std::vector<int> bit_count(32, 0);

        // Count the number of times a number with a particular bit count appears
        for (int number : unique_numbers) {
            ++bit_count[__builtin_popcount(number)]; // __builtin_popcount returns the number of bits set to 1
        }

        // Variable to store the final count of excellent pairs
        long long excellent_pairs_count = 0;

        // Iterate over each unique number and find the count of numbers that
        // have the required number of bits for forming an excellent pair with the current number
        for (int number : unique_numbers) {
            int current_bit_count = __builtin_popcount(number); // Get bit count of the current number
            for (int i = 0; i < 32; ++i) {
                if (current_bit_count + i >= k) {
                    // If the sum of the bit counts of both numbers meets or exceeds k,
                    // add the frequency of the corresponding bit count to the answer
                    excellent_pairs_count += bit_count[i];
                }
            }
        }

        // Return the final count of excellent pairs
        return excellent_pairs_count;
    }
};

```

Typescript Solution

```typescript
function countSetBits(num: number): number {
    // Function to count the number of set bits in the binary representation of a number
    let count = 0;
    while (num > 0) {
        count += num & 1; // Increment count if the least significant bit is set
        num >>>= 1;       // Logical Right Shift to process the next bit
    }
    return count;
}

function countExcellentPairs(nums: number[], k: number): number {
    // Set to eliminate duplicates from the input array
    const uniqueNumbers = new Set<number>(nums);

    // Array to count the frequency of set bits (1s) in the binary representation of numbers
    // Since integers in JavaScript are represented using 32 bits, we create an array of size 32
    const bitCount = new Array(32).fill(0);

    // Count the number of times a number with a particular bit count appears
    uniqueNumbers.forEach(number => {
        bitCount[countSetBits(number)]++; // Increment the frequency of the bit count
    });

    // Variable to store the final count of excellent pairs
    let excellentPairsCount = 0;

    // Iterate over each unique number and find the count of numbers that
    // have the required number of bits for forming an excellent pair with the current number
    uniqueNumbers.forEach(number => {
        const currentBitCount = countSetBits(number); // Get bit count of the current number
        for (let i = 0; i < 32; i++) {
            if (currentBitCount + i >= k) {
                // If the sum of the bit counts of both numbers meets or exceeds k,
                // add the frequency of the corresponding bit count to the answer
                excellentPairsCount += bitCount[i];
            }
        }
    });

    // Return the final count of excellent pairs
    return excellentPairsCount;
}

```

Time and Space Complexity

The given code counts the number of "excellent" pairs in an array, with a pair `(a, b)` being excellent if the number of bits set to `1` in their binary representation (`a OR b`) is at least `k`.

Time Complexity

The time complexity of the function involves several steps:

1. Creating a set `s` from `nums`: This operation has a time complexity of `O(N)` where `N` is the number of elements in `nums`, as each element is added to the set once.

2. Counting the bit_count `t` of each distinct number in `s`: Each call to `v.bit_count()` is `O(1)`. Since we do this for each number in the set `s`, this step has a complexity of `O(U)`, where `U` is the number of unique numbers in `nums`.

3. Populating the `Counter`: This again is `O(U)` since we're iterating over the set `s` once.

4. Running the nested loops, where the outer loop is over each unique value `v` in `s` and the inner loop is over the counts in `Counter`: Since the outer loop runs `O(U)` times and the inner loop runs a maximum of `O(U)` times, the nested loop part has a worse-case complexity of `O(U^2)`.

The total time complexity is, therefore, `O(N + U^2)`.

Space Complexity

The space complexity consists of:

1. The set `s`, which takes `O(U)` space.

2. The `Counter` object `cnt`, which takes another `O(U)` space.

The additional space for the variable `ans` and loop counters is `O(1)`.

Hence, the total space complexity of the function is `O(U)`, where `U` is the count of unique numbers in `nums`.