2275. Largest Combination With Bitwise AND Greater Than Zero

Medium | Bit Manipulation | Array | Hash Table | Counting

Problem Description

The challenge is to find the largest combination of numbers from an array `candidates` such that when the bitwise AND operation is applied to all numbers in the combination, the result is greater than 0. The bitwise AND of an array is calculated by performing the bitwise AND operation on every integer in the array. Remember, each number in `candidates` can be used only once in each combination.

For example, if we have the array `[1, 5, 3]`, the bitwise AND is calculated as `1 & 5 & 3` which equals `1`. In the case of a single-element array like `[7]`, the bitwise AND is simply that element itself, which is `7`.

The goal is to figure out the size of the largest such combination with a non-zero bitwise AND result.

Intuition

The solution relies on understanding how bitwise AND operations work. When you perform a bitwise AND across multiple numbers, the only way for a bit to remain set (remain 1) in the result is if that bit is set in all the numbers being compared.

Therefore, rather than looking at specific combinations which would lead to a potentially massive computational problem, we need to look at each bit position across all numbers in `candidates`.

Here's the intuition behind the solution:

- We loop over each bit position from 0 to 31 (covering all the bits for a standard 32-bit integer) since we are dealing with positive integers.
- In each iteration, we calculate how many numbers in `candidates` have their bit set at the current position.
- We keep track of the maximum count of set bits encountered at any position. This maximum count represents the largest combination size, as it shows us the largest group of numbers with a particular bit set. It ensures the bitwise AND will be greater than 0, as at least one bit will be set in the result by all numbers in that combination.
- Finally, we return the largest count which is the size of the largest combination with a bitwise AND greater than 0.

Solution Approach

The implementation of the solution is straightforward, reflecting the intuition discussed earlier. Let's walk through the essential parts of the code and the rationale behind them:

1. **Initialization of the answer variable:** `ans = 0`
   - This variable `ans` will hold the maximum count of numbers with a particular bit set that has been encountered so far.
2. **Iterating through each bit position:** `for i in range(32):`
   - The `for` loop iterates from `0` to `31`, corresponding to each bit position in a 32-bit integer.
3. **Counting set bits at the current position in all candidates:**

```
t = 0
for x in candidates:
    t += (x >> i) & 1
```

   - We introduce a temporary counter `t` for each bit position `i`.
   - For each candidate number `x`, we right-shift (`>>`) its bits by `i` positions, which moves the bit at position `i` to the least significant bit position (rightmost position).
   - We then perform a bitwise AND with `1` (`& 1`), which isolates the least significant bit.
   - If this bit is set, it contributes `1` to the count `t`; otherwise, it contributes `0`.
   - After iterating over all numbers, `t` will hold the total count of numbers with the `i`-th bit set.
4. **Updating the answer with the maximum count found:** `ans = max(ans, t)`
   - After counting the set bits for the current `i`-th position, we compare this count (`t`) with our current maximum `ans`.
   - If `t` is greater, it means we've found a larger combination of numbers with their `i`-th bit set, so we update `ans` accordingly.
5. **Return the largest combination size:** `return ans`
   - Once we have finished checking all bit positions, `ans` holds the size of the largest combination of `candidates` that can produce a non-zero bitwise AND.
   - This is the final result, which we return.

The solution elegantly bypasses the need for combination generation using a frequency count method that capitalizes on the nature of the bitwise AND operation. No additional data structures beyond simple variables are needed, and the solution runs in `O(32 * N)` time, where `N` is the number of elements in `candidates`, which is optimal given the problem constraints.

Example Walkthrough

Let's illustrate the solution approach using a small example with the `candidates` array `[3, 1, 2]`. We want to find the size of the largest combination where the bitwise AND is greater than 0.

1. **Initialization of the answer variable:** Our starting answer (max count) `ans` is set to `0`.
2. **Iterating through each bit position:** We loop over bit positions from `0` to `31`. Let's consider the first three positions as an example in this walkthrough.
3. **Counting set bits at the current position in all candidates:**
   - For bit position `0`:
     - `3 >> 0` is `3` (binary `11`), `(3 >> 0) & 1` is `1`.
     - `1 >> 0` is `1` (binary `01`), `(1 >> 0) & 1` is `1`.
     - `2 >> 0` is `2` (binary `10`), `(2 >> 0) & 1` is `0`.
     - Count `t` becomes `2`, as there are two numbers with the least significant bit set.
   - Update answer: `ans = max(0, 2)` which is `2`.
   - For bit position `1`:
     - `3 >> 1` is `1` (binary `01`), `(3 >> 1) & 1` is `1`.
     - `1 >> 1` is `0` (binary `00`), `(1 >> 1) & 1` is `0`.
     - `2 >> 1` is `1` (binary `01`), `(2 >> 1) & 1` is `1`.
     - Count `t` is again `2`, as there are two numbers with the second least significant bit set.
   - Update answer: `ans = max(2, 2)` does not change and remains `2`.
   - For bit position `2`:
     - `3 >> 2` is `0` (binary `00`), `(3 >> 2) & 1` is `0`.
     - `1 >> 2` is `0` (binary `00`), `(1 >> 2) & 1` is `0`.
     - `2 >> 2` is `0` (binary `00`), `(2 >> 2) & 1` is `0`.
     - Count `t` is `0`, as none of the numbers have the third bit set.
   - Update answer: `ans = max(2, 0)` remains `2`.

We would continue this process for all bit positions, but in this case, it's clear that the largest combination size doesn't change and `ans` would stay as `2` because there aren't higher bit positions set in any of the numbers in `candidates`.

4. **Return the largest combination size:** After checking all possible bit positions in the loop, we find that the answer `ans` is `2`, which is the size of the largest combination that can produce a non-zero bitwise AND.

In this example, the combinations `[3, 1]` and `[1, 2]` have at least one common bit set (either the least or second-least significant bit), and thus both combinations would result in a bitwise AND greater than `0`. No combination of three numbers exists that would satisfy the condition, so `2` is the largest size for such a combination.

Solution Implementation

Python

```python
class Solution:
    def largestCombination(self, candidates: List[int]) -> int:
        # Initialize the maximum count of candidates that have
        # the same bit set (starting with the count set to 0)
        max_count = 0

        # Iterate over each bit position (0 to 31 since we're working with 32-bit integers)
        for bit_position in range(32):
            # Counter to hold the number of candidates with the current bit set
            count_with_bit_set = 0

            # Iterate over the candidate numbers
            for candidate in candidates:
                # Check if the bit at the current bit position is set (1)
                if (candidate >> bit_position) & 1:
                    # If so, increment our counter
                    count_with_bit_set += 1

            # Update the maximum count if the current count is greater than the previously recorded maximum
            max_count = max(max_count, count_with_bit_set)

        # Return the maximum count of candidates that have a particular bit set
        return max_count
```

Java

```java
class Solution {
    // Method to find the largest combination of numbers that have the highest number of
    // set bits in the same position when aligned in binary representation.
    public int largestCombination(int[] candidates) {
        int maxCombinationSize = 0; // This will hold the maximum combination size

        // Iterate through each bit position from 0 to 31 (for 32-bit integers)
        for (int i = 0; i < 32; ++i) {
            int count = 0; // Count of candidates that have the current bit set

            // Iterate through each candidate number
            for (int candidate : candidates) {
                // Right shift the candidate by i bits and check if the least significant bit is set
                // If it's set, increment the count for this bit position
                count += (candidate >> i) & 1;
            }

            // Compare the count for the current bit position with the maximum combination size
            // and update the maxCombinationSize if necessary
            maxCombinationSize = Math.max(maxCombinationSize, count);
        }

        // Return the maximum combination size found
        return maxCombinationSize;
    }
}
```

C++

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

class Solution {
public:
    int largestCombination(std::vector<int>& candidates) {
        // Declare 'BITS' to specify the number of bits to check in each number.
        const int BITS = 24;

        // Initialize 'maxCount' to 0 to keep track of the maximum count of numbers that have the same bit set.
        int maxCount = 0;

        // Iterate over each bit position from 0 to 'BITS - 1'.
        for (int i = 0; i < BITS; ++i) {
            // Initialize 'bitCount' to count the number of candidates with the current bit set.
            int bitCount = 0;

            // Iterate through each number in the 'candidates' vector.
            for (int num : candidates) {
                // Use bitwise operations to check if the bit at position 'i' is set.
                if ((num >> i) & 1) {
                    // If the current bit is set, increment 'bitCount'.
                    bitCount++;
                }
            }

            // Update 'maxCount' if 'bitCount' of the current bit is greater than the 'maxCount' found so far.
            maxCount = std::max(maxCount, bitCount);
        }

        // Return the maximum count of numbers that have the same bit set.
        return maxCount;
    }
};
```

TypeScript

```typescript
function largestCombination(candidates: number[]): number {
    // Declare 'BITS' to specify the number of bits to check in each number.
    const BITS = 24;

    // Initialize 'maxCount' to 0 to keep track of the maximum count of numbers that have the same bit set.
    let maxCount = 0;

    // Iterate over each bit position from 0 to 'BITS - 1'.
    for (let i = 0; i < BITS; i++) {
        // Initialize 'bitCount' to count the number of candidates with the current bit set.
        let bitCount = 0;

        // Iterate through each number in the candidates array.
        for (let num of candidates) {
            // Use bitwise operations to check if the bit at position 'i' is set.
            if ((num >> i) & 1) {
                // If the current bit is set, increment 'bitCount'.
                bitCount++;
            }
        }

        // Update 'maxCount' if 'bitCount' of the current bit is greater than the 'maxCount' found so far.
        maxCount = Math.max(maxCount, bitCount);
    }

    // Return the maximum count of numbers that have the same bit set.
    return maxCount;
}
```

```python
class Solution:
    def largestCombination(self, candidates: List[int]) -> int:
        # Initialize the maximum count of candidates that have
        # the same bit set (starting with the count set to 0)
        max_count = 0

        # Iterate over each bit position (0 to 31 since we're working with 32-bit integers)
        for bit_position in range(32):
            # Counter to hold the number of candidates with the current bit set
            count_with_bit_set = 0

            # Iterate over the candidate numbers
            for candidate in candidates:
                # Check if the bit at the current bit position is set (1)
                if (candidate >> bit_position) & 1:
                    # If so, increment our counter
                    count_with_bit_set += 1

            # Update the maximum count if the current count is greater than the previously recorded maximum
            max_count = max(max_count, count_with_bit_set)

        # Return the maximum count of candidates that have a particular bit set
        return max_count
```

Time and Space Complexity

The time complexity of the given code is `O(n * m)`, where `n` is the number of elements in `candidates` and `m` is the number of bits considered for each number (here, `m` is fixed at 32, since we're dealing with 32-bit integers). This is because we have one outer loop running for each of the 32 bits, and an inner loop that goes through each of the `n` candidates to count the bits at the `i-th` position.

The space complexity is `O(1)` since we only use a constant amount of extra space, including variables for the answer `ans`, the counter `t`, and the loop index `i`, irrespective of the input size.