2044. Count Number of Maximum Bitwise-OR Subsets

Medium Bit Manipulation Array Backtracking Leetcode Link

Problem Description

In this problem, you're given an array of integers called `nums`. Your task is to find the maximum bitwise OR value that can be achieved by any non-empty subset of `nums`. The bitwise OR operation is a binary operation that takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits. For two bits `a` and `b`, the result is `1` if either `a` or `b` is `1`. Moreover, you need to determine how many unique non-empty subsets yield this maximum bitwise OR value.

To clarify the terms used:

- A *subset* of an array is formed by deleting zero or more elements from the array without changing the order of the remaining elements.
- Two subsets are considered *different* if they involve different indices of `nums`.
- The *bitwise OR* of an array is calculated by applying the OR operation to all its elements, starting with an initial value of `0`. For example, if you have an array `[a, b, c]`, the bitwise OR would be `a OR b OR c`.

So, if the input array is `[3,1]`, there are three subsets: `[3]`, `[1]`, and `[3,1]`. The maximum bitwise OR value is `3 OR 1 = 3`. There are two subsets (`[3]` and `[3,1]`) that have a bitwise OR of `3`. Therefore, the answer would be `2`.

Intuition

To solve this problem, we can apply a depth-first search (DFS) strategy.

The base intuition lies in understanding that the maximum bitwise OR of a set is bounded by the OR of the entire set. No subset can have a higher OR value than the OR of the entire set since adding more numbers can only maintain or increase the OR value.

With that in mind, calculate the OR of the entire array. This will be the maximum OR value (`mx`) that we're seeking subsets for.

Next, use a DFS function to explore all possible subsets. Start with an initial value `t` as `0` (the OR of an empty set) and iterate through the array. For each element at index `i`, you have two choices: either include `nums[i]` in the current subset or not. This leads to two recursive calls: one where you pass the current OR value unchanged, and another where you include `nums[i]` in the OR.

Keep track of how many times you reach the end of the array (`i == len(nums)`) with a subset OR value (`t`) equal to the maximum OR value (`mx`). Each time you reach this condition, increment your answer (`ans`), which keeps count of the number of subsets meeting the criteria.

After exploring all possibilities, `ans` will hold the number of subsets that have the maximal OR value, which is what the problem asks for.

Solution Approach

The implementation of the solution involves using a recursive function, `dfs`, which stands for depth-first search, to explore all possible subsets of the input array `nums`.

Here's a step-by-step breakdown of the algorithm:

1. Initialize the variable `mx` to `0`. This will hold the maximum bitwise OR value that can be achieved by any subset of `nums`.

2. Go through each element in `nums` and perform a bitwise OR operation between `mx` and the current element. Update `mx` with the result. After iterating over all elements, `mx` will be the maximum bitwise OR value of the entire set of `nums`.

3. Define the recursive function `dfs(i, t)`. The parameter `i` represents the current index in the array `nums` that we're considering, and `t` is the current OR value of the subset being explored.

4. Base case: If `i` is equal to the length of `nums`, we've explored a complete subset. If `t` is equal to `mx`, increment the answer `ans` by `1`, since we've found a subset with the maximum OR value.

5. Recursive case: Make two recursive calls:

   - The first call, `dfs(i + 1, t)`, explores the possibility of not including the current element (at index `i`) in the subset, so we pass the `t` unchanged.
   - The second call, `dfs(i + 1, t | nums[i])`, includes the current element in the subset and OR's it with the current value `t`.
6. Start the DFS by calling `dfs(0, 0)`, considering an initially empty subset (OR value of `0`), and starting from index `0`.

7. After all recursive calls are completed, `ans` will contain the number of different non-empty subsets with the maximum bitwise OR, which is then returned as the final result.

The algorithm effectively uses the DFS pattern to explore all subsets. It utilizes recursion to traverse the search tree where each node represents a potential subset with a specific OR value. The data structures used are simple integers to store the maximum OR value (`mx`) and the count of subsets (`ans`).

By using the DFS pattern, the solution ensures that all possible subsets are accounted for, without the need to explicitly generate and store each subset, making it an efficient approach to solving the problem.

Example Walkthrough

Let's consider an example array `nums = [2, 5, 4]` to illustrate the solution approach.

Step 1: Initialize `mx`

We initialize `mx = 0`. This will hold the maximum bitwise OR value.

Step 2: Compute Maximum Bitwise OR for Entire Array

We iterate through the `nums` array to compute the maximum bitwise OR value which is the OR of all elements.

- `mx = mx OR 2 = 0 OR 2 = 2`
- `mx = mx OR 5 = 2 OR 5 = 7`
- `mx = mx OR 4 = 7 OR 4 = 7`

After this step, `mx` holds the value `7` which is the maximum bitwise OR value for the entire array.

Step 3: Define Recursive Function `dfs(i, t)`

We will define a recursive function `dfs(i, t)` to explore subsets. `i` represents the current index and `t` is the current OR value of the subset.

Step 4: Base Case to Count Valid Subsets

When `i == len(nums)`, we check if `t == mx`. If yes, we found a valid subset and increment a counter.

Step 5: Recursive Case to Explore Subsets

We will consider two recursive calls starting from index 0:

1. Not including the current element: `dfs(i + 1, t)`
2. Including the current element: `dfs(i + 1, t | nums[i])`

Step 6: Start DFS

We call `dfs(0, 0)` to start the exploration.

Recursive Exploration

Beginning with `dfs(0, 0)`, we have an empty subset with an OR value of `0`.

First call: `dfs(1, 0 | 2) -> dfs(1, 2)` Second call: `dfs(1, 0)`

Now, from `dfs(1, 2)`:

First call: `dfs(2, 2 | 5) -> dfs(2, 7)` (since 2 OR 5 = 7) Second call: `dfs(2, 2)`

Similarly, `dfs(1, 0)` branches to `dfs(2, 0 | 5) -> dfs(2, 5)` and `dfs(2, 0)`.

Continuing this process:

- `dfs(2, 7)` branches to `dfs(3, 7 | 4) -> dfs(3, 7)` and `dfs(3, 7)`
- `dfs(2, 2)` branches to `dfs(3, 2 | 4) -> dfs(3, 6)` and `dfs(3, 2)`
- `dfs(2, 5)` branches to `dfs(3, 5 | 4) -> dfs(3, 5)` and `dfs(3, 5)`
- `dfs(2, 0)` branches to `dfs(3, 0 | 4) -> dfs(3, 4)` and `dfs(3, 0)`

Counting Valid Subsets

- When `i == len(nums)`, we check if `t == mx` (in this case, `7`). If yes, we found a valid subset.
- For our example, the calls that end with `t == 7` are `dfs(3, 7)`, `dfs(3, 7)` (from both `dfs(2, 7)` and `dfs(2, 5)`), which counts to `3`.

Conclusion

The subsets giving us the maximum OR of `7` are [2, 5], [5, 4], and [2, 5, 4]. After exploring all possible subsets recursively, we find that there are `3` non-empty subsets that yield the maximum bitwise OR value of `7`. Thus, the solution for the array `nums = [2, 5, 4]` returns `3`.

Python Solution

```python
from typing import List

class Solution:
    def countMaxOrSubsets(self, nums: List[int]) -> int:
        # Initialize the maximum OR value and answer to zero.
        max_or_value = answer = 0

        # Compute the maximum OR value across all numbers in the list.
        for num in nums:
            max_or_value |= num

        # Define the depth-first search function to explore all subsets.
        def dfs(index, current_or):
            nonlocal max_or_value, answer  # Access variables from the outer scope.

            # Base case: if the index is equal to the length of nums,
            # we've considered all elements.
            if index == len(nums):
                # If the current OR is equal to the max OR value, increment the answer.
                if current_or == max_or_value:
                    answer += 1
                return

            # Recursive call to explore the subset without including the current number.
            dfs(index + 1, current_or)

            # Recursive call to explore the subset including the current number.
            dfs(index + 1, current_or | nums[index])

        # Begin the depth-first search with the first index and an initial OR value of 0.
        dfs(0, 0)

        # Return the total count of subsets that have the maximum OR value.
        return answer

# The Solution can now be used to create an instance and call the countMaxOrSubsets method.

```

Java Solution

```java
class Solution {
    private int maxOrValue; // Variable to store the maximum OR value of any subset.
    private int count; // Counter for subsets equal to the maximum OR value.
    private int[] nums; // Array to store the input numbers.

    // Method to count the number of maximum OR subsets.
    public int countMaxOrSubsets(int[] nums) {
        maxOrValue = 0; // Initialize the maximum OR value.
        // Calculate the maximum OR value by OR'ing all elements in the array.
        for (int num : nums) {
            maxOrValue |= num;
        }
        this.nums = nums; // Store the input array for further use in the method.
        dfs(0, 0); // Start the Depth First Search (DFS) traversal.
        return count; // Return the count of subsets with maximum OR value.
    }

    // Helper method to perform DFS traversal to find all subsets.
    private void dfs(int index, int currentOr) {
        // Base case: if we've considered all elements,
        // check if the current OR equals the maximum OR value.
        if (index == nums.length) {
            if (currentOr == maxOrValue) {
                count++; // Increment count if current subset OR equals max OR value.
            }
            return; // Return to explore other subsets.
        }
        // Exclude the current element and proceed to the next index.
        dfs(index + 1, currentOr);
        // Include the current element, OR it with the current value, and proceed.
        dfs(index + 1, currentOr | nums[index]);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxOrValue;         // To store the maximum OR value calculated from all numbers
    int subsetCount;        // To store the count of subsets whose OR equals maxOrValue
    vector<int> numbers;    // To store the input numbers

    // Function to calculate the count of subsets whose bitwise OR equals the maximum OR of all numbers
    int countMaxOrSubsets(vector<int>& nums) {
        numbers = nums;     // Initialize class member with input
        maxOrValue = 0;     // Initialize max OR value to 0
        subsetCount = 0;    // Initialize count of subsets to 0

        // Calculate the maximum OR value from all the numbers
        for (int x : nums) {
            maxOrValue |= x;
        }

        // Start DFS traversal to explore all subsets and count those with max OR value
        dfs(0, 0);

        return subsetCount;
    }

    // Recursive function to perform DFS. It takes the current index and the accumulated OR value so far.
    void dfs(int index, int currentOrValue) {
        // Base case: if we have considered all elements
        if (index == numbers.size()) {
            // If the OR value equals the max OR value, increment the counter
            if (currentOrValue == maxOrValue) {
                subsetCount++;
            }
            return;
        }

        // Case 1: Exclude the current number from the subset and continue to the next element
        dfs(index + 1, currentOrValue);

        // Case 2: Include the current number in the subset (OR the current value with this number) and continue
        dfs(index + 1, currentOrValue | numbers[index]);
    }
};

```

Typescript Solution

```typescript
function countMaxOrSubsets(nums: number[]): number {
    // The total number of elements in the given array.
    let totalElements = nums.length;
    // Variable to store the maximum OR value across all subsets.
    let maxOrValue = 0;
    // Calculate the maximum OR value by iterating through the array.
    for (let i = 0; i < totalElements; i++) {
        maxOrValue |= nums[i];
    }
    // A count of subsets that have the maximum OR value.
    let maxOrSubsetsCount = 0;

    // Recursive helper function to perform depth-first search.
    function dfs(currentOr: number, depth: number): void {
        // If this is the end of the array, check if the current OR is the maximum OR.
        if (depth == totalElements) {
            if (currentOr == maxOrValue) {
                // Increment the count if the current OR is equal to max OR.
                maxOrSubsetsCount++;
            }
            return;
        }
        // case when the current number is not taken
        dfs(currentOr, depth + 1);
        // case when the current number is taken, it is ORed with the currentOr.
        dfs(currentOr | nums[depth], depth + 1);
    }

    // Start the depth-first search with an initial OR value of 0 and at depth 0.
    dfs(0, 0);

    // Return the number of subsets that have the maximum OR value.
    return maxOrSubsetsCount;
}

```

Time and Space Complexity

The given Python code defines the `countMaxOrSubsets` method that calculates the count of the maximum OR value subsets from a provided list of integers, `nums`.

Time Complexity

The time complexity of the algorithm is `O(2^N)`, where `N` is the length of the `nums` list. This is because the algorithm uses a depth-first search approach to consider every possible subset of `nums`. In the worst case, every element is either included or excluded in a subset, leading to `2^N` possible subsets.

The method `dfs` is called recursively twice for each element in the `nums` list - once to include the element in the current OR calculation (`t | nums[i]`) and once to exclude it (`t`). Hence, each call generates two more calls until the base case is reached when `i` equals the length of the `nums`.

Space Complexity

The space complexity is `O(N)` due to the recursive depth. In the worst-case scenario, the depth of the recursive call stack will be as deep as the number of elements in the list, since we are performing a depth-first search through the `dfs` function, and on each call to `dfs`, we are passing down a new index `i` which goes from 0 to `N-1`.

Additionally, two variables `mx` and `ans` are used, but their space usage is constant `O(1)` and does not depend on `N`, hence they don't contribute significantly to the space complexity.

It is important to note that the space complexity here refers to the auxiliary space, that is, the additional space excluding the space for the input itself.