Count Number Of Maximum Bitwise OR Subsets

solved by	Senan
Platform	LeetCode
difficulty	Medium
# Serial	2044
tags	Bit Manipulation Dynamic Programming
language	C++
solved on	@18/10/2024
link	https://leetcode.com/problems/count-number-of-maximum-bitwise-or-subsets/description/
Completion	☑

Intuition

The problem revolves around generating subsets of a given array `nums` and finding those whose bitwise OR value equals the maximum possible bitwise OR of the entire array. To solve this efficiently, we need to explore subsets and evaluate their OR results. The use of dynamic programming (DP) will help reduce redundant calculations.

Approach

1. **Max OR Calculation**:

   The first step is to compute the maximum bitwise OR of all elements in the array `nums`. This is done by applying the OR operation to all the elements in a single pass.

2. **Recursive Helper with DP**:

   We then use a recursive function with dynamic programming to explore the inclusion and exclusion of each element in the subsets.

   - At each recursive step, we either include the current element in the OR operation or exclude it.
   - If the current OR matches the maximum OR, we have found a valid subset, and we return the count of all possible subsets for the remaining elements.

3. **Memoization**:

   The recursive calls are memoized using a 2D DP array (`dp`), where `dp[i][curr]` represents the number of ways to form a subset from the remaining elements starting at index `i` with a current OR value of `curr`.

4. **Base Case**:

   If the current OR matches the maximum OR, we know that all remaining subsets (formed by picking or not picking further elements) are valid, so we return `2^(remaining elements)` as there are `2^n` ways to form subsets from `n` elements.

Complexity

Time Complexity:

- **Recursive Exploration**:

  The recursive function explores all subsets, which means it has an exponential time complexity, $O(2^n)$, where n is the size of the input array.

- **Memoization**:

  Memoization reduces the number of recursive calls by storing results in a DP array. The DP array has dimensions `n x maxOr`, where `maxOr` is the maximum bitwise OR value. Therefore, the time complexity is reduced to $\O(n \times maxOr)$.

Overall, the time complexity is $O(n \times maxOr)$.

Space Complexity:

- **DP Table**:

  We use a DP table of size $O(n \times maxOr)$, where n is the number of elements in `nums` and `maxOr` is the maximum OR value.

- **Recursive Stack**:

  The recursion depth can be at most $O(n)$, which is the number of elements.

Thus, the space complexity is $O(n \times maxOr)$.

Code

```
typedef vector<vector<int>> vvi;

class Solution {
    int helper(int i, int curr, int maxOr, vector<int>& nums, vvi &dp) {
        if (curr == maxOr) return 1 << (nums.size() - i);
        if (i >= nums.size()) return 0;
        if (dp[i][curr] != -1) return dp[i][curr];

        // Explore two possibilities: not taking or taking the current number
        int dontTake = helper(i + 1, curr, maxOr, nums, dp);
        int take = helper(i + 1, curr | nums[i], maxOr, nums, dp);

        return dp[i][curr] = take + dontTake;
    }

public:
    int countMaxOrSubsets(vector<int>& nums) {
        // Calculate the maximum possible OR
        int maxOr = 0;
        for (auto elem : nums) maxOr |= elem;
        vvi dp(nums.size() + 1, vector<int>(maxOr + 1, -1));
        return helper(0, 0, maxOr, nums, dp);
    }
};
```