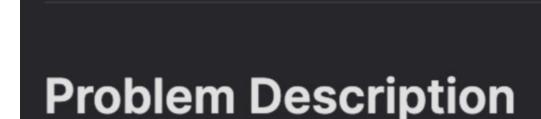
2044. Count Number of Maximum Bitwise-OR Subsets

Backtracking

Two subsets are considered different if they involve different indices of nums.

Array



Bit Manipulation

Medium

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.

Leetcode Link

Moreover, you need to determine how many unique non-empty subsets yield this maximum bitwise OR value.

A subset of an array is formed by deleting zero or more elements from the array without changing the order of the remaining

To clarify the terms used:

elements.

- 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.
- two subsets ([3] and [3,1]) that have a bitwise OR of 3. Therefore, the answer would be 2. Intuition

So, if the input array is [3,1], there are three subsets: [3], [1], and [3,1]. The maximum bitwise OR value is [3,1], there are [3,1], there are

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.

The implementation of the solution involves using a recursive function, dfs, which stands for depth-first search, to explore all

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

result. After iterating over all elements, mx will be the maximum bitwise OR value of the entire set of nums.

possible subsets of the input array nums. Here's a step-by-step breakdown of the algorithm:

2. Go through each element in nums and perform a bitwise OR operation between mx and the current element. Update mx with the

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

• The first call, dfs(i + 1, t), explores the possibility of not including the current element (at index i) in the subset, so we

• 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.

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.

pass the t unchanged.

which is then returned as the final result.

Step 2: Compute Maximum Bitwise OR for Entire Array

value (mx) and the count of subsets (ans).

Example Walkthrough

• mx = mx OR 2 = 0 OR 2 = 2

• $mx = mx \ OR \ 5 = 2 \ OR \ 5 = 7$

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

Step 4: Base Case to Count Valid Subsets

1. Not including the current element: dfs(i + 1, t)

2. Including the current element: dfs(i + 1, t | nums[i])

First call: $dfs(1, 0 \mid 2) \rightarrow dfs(1, 2)$ Second call: dfs(1, 0)

Similarly, dfs(1, 0) branches to $dfs(2, 0 | 5) \rightarrow dfs(2, 5)$ and dfs(2, 0).

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)

Solution Approach

1, since we've found a subset with the maximum OR value. 5. Recursive case: Make two recursive calls:

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,

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

and store each subset, making it an efficient approach to solving the problem.

By using the DFS pattern, the solution ensures that all possible subsets are accounted for, without the need to explicitly generate

Step 1: Initialize mx We initialize mx = 0. This will hold the maximum bitwise OR value.

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

• mx = mx OR 4 = 7 OR 4 = 7After this step, mx holds the value 7 which is the maximum bitwise OR value for the entire array.

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 6: Start DFS

Now, from dfs(1, 2):

Continuing this process:

Counting Valid Subsets

Python Solution

class Solution:

from typing import List

3.

returns 3.

8

9

10

11

19

20

21

22

23

24

25

26

27

28

29

34

35

37

6

Step 5: Recursive Case to Explore Subsets We will consider two recursive calls starting from index 0:

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

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

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

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

First call: $dfs(2, 2 \mid 5) \rightarrow dfs(2, 7)$ (since 2 OR 5 = 7) Second call: dfs(2, 2)

• When i == len(nums), we check if t == mx (in this case, 7). If yes, we found a valid subset.

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)

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

def countMaxOrSubsets(self, nums: List[int]) -> int:

if current_or == max_or_value:

dfs(index + 1, current_or | nums[index])

private int[] nums; // Array to store the input numbers.

// Method to count the number of maximum OR subsets.

public int countMaxOrSubsets(int[] nums) {

answer += 1

dfs(index + 1, current_or)

return

return answer

max_or_value = answer = 0

max_or_value |= num

for num in nums:

Initialize the maximum OR value and answer to zero.

Compute the maximum OR value across all numbers in the list.

12 # Define the depth-first search function to explore all subsets. def dfs(index, current_or): 13 nonlocal max_or_value, answer # Access variables from the outer scope. 14 15 # Base case: if the index is equal to the length of nums, 16 # we've considered all elements. 17 if index == len(nums): 18

If the current OR is equal to the max OR value, increment the answer.

Recursive call to explore the subset without including the current number.

Recursive call to explore the subset including the current number.

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

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.

• 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

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]

30 # Begin the depth-first search with the first index and an initial OR value of 0. dfs(0, 0) 31 32 33 # Return the total count of subsets that have the maximum OR value.

Java Solution

1 class Solution {

int subsetCount;

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

29

30

31

32

33

34

35

36

37

38

39

42

9

10

11

12

13

14

16

17

18

19

20

21

22

23

24

25

26

27

28

29

41 };

vector<int> numbers;

dfs(0, 0);

numbers = nums;

subsetCount = 0;

for (int x : nums) {

return subsetCount;

return;

max0rValue |= x;

int countMaxOrSubsets(vector<int>& nums) {

void dfs(int index, int currentOrValue) {

if (index == numbers.size()) {

subsetCount++;

dfs(index + 1, current0rValue);

function countMaxOrSubsets(nums: number[]): number {

for (let i = 0; i < totalElements; i++) {</pre>

if (depth == totalElements) {

dfs(current0r, depth + 1);

if (current0r == max0rValue) {

max0rSubsetsCount++;

let totalElements = nums.length;

maxOrValue |= nums[i];

let max0rSubsetsCount = 0;

return;

let max0rValue = 0;

// The total number of elements in the given array.

// A count of subsets that have the maximum OR value.

function dfs(currentOr: number, depth: number): void {

// case when the current number is not taken

dfs(current0r | nums[depth], depth + 1);

// Recursive helper function to perform depth-first search.

// Variable to store the maximum OR value across all subsets.

// Calculate the maximum OR value by iterating through the array.

// If this is the end of the array, check if the current OR is the maximum OR.

// Increment the count if the current OR is equal to max OR.

// case when the current number is taken, it is ORed with the currentOr.

maxOrValue = 0; // Initialize max OR value to 0

// Base case: if we have considered all elements

dfs(index + 1, currentOrValue | numbers[index]);

if (currentOrValue == maxOrValue) {

// Calculate the maximum OR value from all the numbers

```
maxOrValue = 0; // Initialize the maximum OR value.
           // Calculate the maximum OR value by OR'ing all elements in the array.
           for (int num : nums) {
10
               maxOrValue |= num;
11
12
           this.nums = nums; // Store the input array for further use in the method.
13
14
           dfs(0, 0); // Start the Depth First Search (DFS) traversal.
           return count; // Return the count of subsets with maximum OR value.
15
16
17
       // Helper method to perform DFS traversal to find all subsets.
18
       private void dfs(int index, int current0r) {
19
20
           // Base case: if we've considered all elements,
           // check if the current OR equals the maximum OR value.
           if (index == nums.length) {
22
23
               if (current0r == max0rValue) {
24
                   count++; // Increment count if current subset OR equals max OR value.
25
26
               return; // Return to explore other subsets.
27
28
           // Exclude the current element and proceed to the next index.
29
           dfs(index + 1, current0r);
30
           // Include the current element, OR it with the current value, and proceed.
31
           dfs(index + 1, current0r | nums[index]);
32
33 }
34
C++ Solution
1 class Solution {
2 public:
       int maxOrValue;
                               // To store the maximum OR value calculated from all numbers
```

// To store the count of subsets whose OR equals maxOrValue

// Function to calculate the count of subsets whose bitwise OR equals the maximum OR of all numbers

// Recursive function to perform DFS. It takes the current index and the accumulated OR value so far.

// Case 1: Exclude the current number from the subset and continue to the next element

// Case 2: Include the current number in the subset (OR the current value with this number) and continue

// To store the input numbers

// Initialize class member with input

// Start DFS traversal to explore all subsets and count those with max OR value

// If the OR value equals the max OR value, increment the counter

// Initialize count of subsets to 0

Typescript Solution

```
// Start the depth-first search with an initial OR value of 0 and at depth 0.
       dfs(0, 0);
30
31
       // Return the number of subsets that have the maximum OR value.
32
33
       return maxOrSubsetsCount;
34 }
35
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 0(2^N), where N is the length of the nums list. This is because the algorithm uses a depth-first
```

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.

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

search approach to consider every possible subset of nums. In the worst case, every element is either included or excluded in a

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 0(1) and does not depend on N, hence they don't

Space Complexity

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.