Brute Force Approach

Intuition

The simple idea is to run a nested loop and calculate the bitwise OR for all possible combinations and calculate the sum with modulo 1e9+7 as mentioned in the question which is our desired result.

Implementation

1. Initialize a variable ans to store the result.
2. Define a constant mod as 10^9 + 7 to handle the modulo operation.
3. Use two nested loops to iterate through both arrays, A and B.
    - For each pair of elements (A[i], B[j]), calculate their bitwise OR (A[i] | B[j]) and add it to the ans.
    - Update ans with the modulo operation to prevent integer overflow.
4. Return the final result ans as the sum of bitwise OR of all pairs modulo 10^9 + 7.

Code

C++

```
class Solution{
    public:
    int orSum(int N, vector<int> A, int M, vector<int> B){
        // code here
        int ans=0;
        const int mod = 1e9+7;
        for(int i=0;i<N;i++){
            for(int j=0;j<M;j++){
                ans += (A[i]|B[j]);
                ans %= mod;
            }
        }
        return ans;
    }
};
```

Complexity

- **Time Complexity: O(N*M),** As the algorithm uses two nested loops to iterate through both arrays, A and B, and number of iterations is proportional to the product of the sizes of A and B, which is N * M.

- **Space Complexity: O(1),** The algorithm uses a constant amount of extra space for variables (ans, mod), so the space complexity is O(1).

Expected Approach

Intuition

- Instead of computing each pair's OR directly, we can break down the problem into bitwise operations on individual bits.

- The key insight is to compute the contribution of each bit position separately and then combine them to find the final result.

- To do this, we can iterate through the bits of the elements in both arrays and calculate how many times each bit is set in the entire array B and how many times it is set in the total array A.

- Then, we multiply these counts by 2^bit_position and add them to the final result, taking care of the modulo operation to avoid overflow.

Implementation

- **Initialize key variables:**
    - **mod** to store the modulo value 10^9 + 7.
    - **ans** to keep track of the final answer.
    - Two arrays, **fre_b and fre_total,** both of size 30 (since integers are typically represented using 32 bits). These arrays will be used to store the frequency of each bit in the binary representation of the numbers.

- **Calculate the frequency of each bit in array B (fre_b):**
    - Iterate through each bit position from 0 to 29 (30 bits in total).
    - For each bit position, create a bitmask temp (2^bit).
    - Iterate through array B and check if the bit at the current position is set (not equal to 0). If it's set, increment the frequency count fre_b for that bit. This step calculates the frequency of each bit in array B.

- **Calculate the frequency of each bit in array A (fre_total) while considering the contribution from array B:**
    - Similar to step 2, iterate through each bit position.

    - For each bit position, create the bitmask temp.
    - Iterate through array A and check if the bit at the current position is set.
    - If the bit is set, add M to the frequency count fre_total for that bit, indicating that this bit contributes to M pairs.
    - If the bit is not set in A, add the frequency of that bit from array B to fre_total, indicating that this bit contributes to the count for pairs with B.

- **Calculate the contribution of each bit to the final answer:**
    - Iterate through each bit position.
    - For each bit, calculate its contribution:
        - Calculate temp as 2^bit modulo mod.
        - Make fre_total modulo mod to avoid overflow.
        - Multiply temp by fre_total modulo mod.
        - Add the result to ans modulo mod.

- Return **ans** as the final answer, which represents the sum of bitwise OR of all pairs A[i]*B[j] modulo 10^9 + 7.

Complexity

Time Complexity: O((N + M) * log(max(A[i], B[j]))) because it iterates through both arrays and performs bitwise operations on each element. The dominant factor is the maximum element in A and B.
Space Complexity: O(1) because the code uses a fixed amount of memory for storing variables regardless of the input sizes N and M.