2505. Bitwise OR of All Subsequence Sums

Array

Brainteaser

Problem Description

Bit Manipulation

Medium

explain further, a subsequence is a series of numbers that come from the original array either entirely or after removing some elements, but with the original order preserved. An empty subsequence is also considered valid. The special value that the problem requires is the bitwise OR of all the unique subsequence sums. For instance, given an array [1,2], the subsequences are [1], [2], and [1,2], plus the empty subsequence []. The sum of these subsequences are 1, 2, 3, and 0 respectively. The bitwise OR of these sums (1 | 2 | 3 | 0) equals 3.

This problem requests the calculation of a special value derived from all possible subsequence sums of an integer array nums. To

The problem's complexity comes from the potentially large number of subsequences, which increases exponentially with the size of the array. The challenge is to find an efficient way to compute the special value without having to actually generate and sum every possible subsequence.

count in cnt[i].

we observe that each bit position in the integers can be treated independently due to the properties of the bitwise OR operator. Here's the intuition behind the solution: 1. Initialize a counter array, cnt, which will keep track of the number of times a bit is set at different positions across all numbers in the array.

2. Iterate through each number v in the array nums. For each bit position i, check if the ith bit of v is set (i.e., equals 1). If so, increment the

4. Iterate through the counter array cnt. If at any bit position i, cnt[i] is non-zero, it implies that this bit appeared in our subsequence sum.

Thus, set the ith bit in ans to 1 by performing a bitwise OR between ans and 1 shifted i times to the left.

To devise a solution to the problem without generating every possible subsequence (which would be computationally expensive),

5. Additionally, we carry over half of the count from each bit position to the next higher position by adding cnt[i] // 2 to cnt[i + 1]. This represents the combination of bits when we add numbers together, as bits carry when there is a sum of 1 + 1 = 10 in binary. 6. The loop iterates up to 63, as it accounts for the carry that could propagate through all 32 bit positions twice (since every addition might cause a carry to the next position).

7. The variable ans now contains the bitwise OR of all the possible subsequence sums, which is the required answer.

interact with each other on the binary level.

Here's a detailed walk through of the solution code:

happen as we simulate adding the numbers together.

because that bit will appear in at least one subsequence sum.

3. Initialize a variable ans, which will hold the final answer.

The solution leverages the fact that we can analyze and manipulate the bits of the entire array rather than individual subsequences. It counts bit occurrences and then uses those counts to determine which bits will definitely be set in the final OR

result. The carrying of half the count to the next bit position simulates how bit addition would occur across all subsequence sums.

Solution Approach

The solution declares a Solution class with a method called subsequenceSumOr, which accepts an array nums.

The implementation of the solution is based on bit manipulation and the understanding of how bitwise OR and sum operations

Inside this method, a cnt array with 64 elements initialized to 0 is created. This array is used to count the occurrences of set bits in the binary representation of all numbers in nums. It reserves enough space to handle the bit carry operations that can

cnt = [0] * 64

• The ans variable is initialized to 0. This variable will hold the result of the bitwise OR of all possible subsequence sums. ans = 0

• The solution runs a loop over each number v in the nums array and then a nested loop over each bit position i, starting from 0 up to 30 (inclusive), which is sufficient to cover the bit-positions for a 32 bit integer. for v in nums:

• If the ith bit of v is set (i.e., v >> i bitwise right shift i and bitwise AND with 1 gives us the ith bit), the corresponding count in cnt[i] is

incremented.

if (v >> i) & 1:

cnt[i] += 1

for i in range(31):

 After the counts are determined, the solution iterates over all 63 possible bit positions—enough to account for carry-over during addition. For each i, if cnt[i] is non-zero, 1 << i (bitwise left shift, which effectively is 2^i) is bitwise ORed with ans. This sets the ith bit of ans

for i in range(63): if cnt[i]: ans |= 1 << i

to cnt[i + 1]. This simulates the carry-over process in binary addition across all subsequence sums. cnt[i + 1] += cnt[i] // 2

• The solution also ensures that the number of times a bit is carried over to the next higher position is taken into account by adding cnt[i] // 2

The implementation successfully leverages bit manipulation to efficiently solve a problem that would otherwise require an impractical brute-force approach due to the exponentially growing number of subsequences in the array.

Example Walkthrough

cnt = [0] * 64

ans = 0

return ans

We create ans variable, initialized to 0, which will store our answer:

We initialize a counter array cnt of 64 elements to keep track of the number of set bits at each bit position:

We iterate over each value v in nums (3 and 5), and for each value, we iterate over each bit position i from 0 to 30:

• Finally, ans is returned. It now holds the value of the bitwise OR of the sum of all possible subsequences.

After iterating through the array, our cnt array will have the counts for each bit position:

index: 0 1 2 ...

for i in range(63):

cnt[2] += cnt[1] // 2

Solution Implementation

bit count = [0] * 64

for value in nums:

return result

for i in range(31):

if bit count[i]:

public long subsequenceSumOr(int[] nums) {

++bitCounts[i];

bitCounts[i + 1] += bitCounts[i] / 2;

// Counting set bits for every position in all numbers

// Counting set bits for every position in all numbers.

for (int i = 0; i < 31; ++i) { // Loop through each bit position

++bitCounts[i]; // Increment count for this bit position

let ans: number = 0; // To store the result of sum of the bitwise OR of all subsequences.

for (let i = 0; i < 31; i++) { // Loop through each bit position.

bitCounts[i]++; // Increment count for this bit position.

// Calculating the sum of bitwise OR of all subsequences using the bit counts.

for (let i = 0; i < 63; i++) { // Loop through up to the second-to-last bit position.

ans |= 1 << i; // Update result with the i-th bit set if count is not zero.

for i in range(63): # Use 63 because we're carrying over the bits to higher bits

result |= 1 << i # OR the result with the bit set if it's count is non-zero

bit_count[i + 1] += bit_count[i] // 2 # Carry over half the count to the next higher bit

// Since any set bit will contribute to two subsequences when paired with another bit,

if (num & (1 << i)) { // Check if the i-th bit is set.</pre>

// we add half the count of the current bit to the next bit.

Array to count the number of times each bit is set in all numbers

Store the final result - sum of ORs of all subsequences

bitCounts[i + 1] += Math.floor(bitCounts[i] / 2);

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

bit_count[i] += 1

Iterate over the bit counts to compute the result

if (num & (1 << i)) { // Check if the ith bit is set</pre>

if (bitCounts[i] > 0) {

// Return the calculated OR sum

return answer;

for (int num : nums) {

nums.forEach(num => {

if (bitCounts[i] > 0) {

return ans; // Return the final sum.

bit count = [0] * 64

if bit count[i]:

Time and Space Complexity

leading to a final time complexity of O(n).

});

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

bit_count[i] += 1

Iterate over the bit counts to compute the result

// Count the occurrence of each bit among the numbers

from typing import List

result = 0

• [] sum is 0,

• [3] sum is 3,

Python

Java

class Solution {

class Solution:

ans |= 1 << i

cnt[i + 1] += cnt[i] // 2

if cnt[i]:

nums = [3, 5]

for v in nums:

for i in range(31):

if (v >> i) & 1:

cnt[i] += 1

We then iterate through the counter cnt and update the ans variable with the bitwise OR operation for any non-zero counts, also handling the carry over:

In this process, the ans variable will be updated as follows:

loop i=1: ans |=1 << 1 # as cnt[1] had a count of 2

Simultaneously, the carry over for the next position is handled:

efficiently calculates the special value without having to enumerate all subsequences.

Array to count the number of times each bit is set in all numbers

if (value >> i) & 1: # Check if the i-th bit of value is set'

for i in range(63): # Use 63 because we're carrying over the bits to higher bits

result |= 1 << i # OR the result with the bit set if it's count is non-zero

long[] bitCounts = new long[64]; // Array to store the count of set bits at each position

long answer = 0; // Variable to store the final OR sum of all subsequences

// Calculate the OR sum by combining bits present in the subsequences

answer |= 1L << i; // Use bitwise OR to set the bits in the answer

// Propagate the carry-over of counting bits to the next higher bit

bit_count[i + 1] += bit_count[i] // 2 # Carry over half the count to the next higher bit

Store the final result — sum of ORs of all subsequences

Count the number of times each bit is set across all numbers

cnt: [0, 2, 1, ...] # Only showing relevant bits

Binary representations for 3 and 5 are 011 and 101 respectively.

Let's use the integer array [3,5] to illustrate the solution approach:

ans now becomes 2 loop i=2: ans |=1 << 2 # as cnt[2] had a count of 1 ans now becomes 6

cnt array becomes [0, 0, 2, ...] after carry, showing we carried 1 to the next significant bit.

The final value of ans is 6, which is the bitwise OR of sum of all possible subsequences of the array [3,5].

The returned result from our example array [3,5] is 6, which corroborates with the computation of the subsequence sums:

Here, the bit at index 1 occurred twice (1 from both 3 and 5), and the bit at index 2 occurred once (from 5).

• [5] sum is 5, • [3,5] sum is 8. The bitwise OR of all sums (0 | 3 | 5 | 8) is indeed 6. This example walk-through shows how the bit manipulation approach

for (int value : nums) { for (int i = 0; i < 31; ++i) { // Check the i-th bit of the number if (((value >> i) & 1) == 1) { // If it's set, increment the count of this bit in the array

for (int i = 0; i < 63; ++i) { // Iterate through all but the last element of the bitCounts array

public: // Function to calculate the sum of bitwise OR of all subsequence. long long subsequenceSumOr(vector<int>& nums) { vector<long long> bitCounts(64); // To store count of set bits for each bit position long long ans = 0; // To store the result of sum of the bitwise OR of all subsequences

class Solution {

C++

// Calculating the sum of the bitwise OR of all subsequences using the bit counts for (int i = 0; i < 63; ++i) { // Loop through up to the second-to-last bit position if (bitCounts[i]) { ans |= 1LL << i; // Update result with the ith bit set if count is not zero // Since any set bit will contribute to two subsequences when paired with another bit, // we add half the count of the current bit to the next bit. bitCounts[i + 1] += bitCounts[i] / 2; return ans; // Return the final sum **}**; **TypeScript** // A function to calculate the sum of bitwise OR of all subsequence. function subsequenceSumOr(nums: number[]): number { let bitCounts: number[] = new Array(64).fill(0); // To store count of set bits for each bit position.

Count the number of times each bit is set across all numbers for value in nums: for i in range(31): if (value >> i) & 1: # Check if the i-th bit of value is set'

class Solution:

from typing import List

result = 0

return result

Time Complexity The time complexity of the provided code is determined by the number of loops and operations within each loop: 1. The outer loop runs for each element in the nums list, thus it is O(n) where n is the length of the list. 2. The inner loop runs for a constant 31 times (assumes 32-bit integer processing), which is 0(1) with respect to the input size.

Space Complexity The space complexity of the code consists of: 1. The space used by the cnt array, which has a fixed size of 64, thus 0(1).

Since the inner loop is nested within the outer loop, we multiply the complexities of two loops, which results in 0(n) * 0(1),

2. The ans variable, which also occupies a constant space and does not depend on the input size. Considering both of these together, the overall space complexity of the provided code is 0(1) since the extra space required

does not scale with the size of the input (n).