1879. Minimum XOR Sum of Two Arrays

Bit Manipulation <u>Array</u> <u>Dynamic Programming</u>

Problem Description

Hard

these arrays which is obtained by taking the XOR of each corresponding pair of elements from the two arrays and then summing those results up. The XOR operation is a bitwise operation where the result is 1 if the two bits are different and 0 if they are the same. The XOR sum is computed as follows:

In this problem, you have two integer arrays nums1 and nums1 each of the same length n. You need to calculate the XOR sum of

Bitmask

words, you want to find an optimal permutation of nums2 that when paired with the elements of nums1, yields the minimum possible XOR sum.

(nums1[0] XOR nums2[0]) + (nums1[1] XOR nums2[1]) + ... + (nums1[n - 1] XOR nums2[n - 1])

For example, if nums1 is [1,2,3] and nums2 is [3,2,1], the XOR sum of these arrays before any rearrangement is (1 XOR 3) + (2 XOR 2) + (3 XOR 1) = 2 + 0 + 2 = 4. By rearranging nums2 properly, you might get a smaller XOR sum. Your task is to determine this minimum XOR sum after rearranging nums2, and to return it.

The challenge is to rearrange the elements in nums2 in such a way that the resulting XOR sum is as small as possible. In other

Intuition The solution to this problem is using a <u>dynamic programming</u> approach that involves trying out different combinations of pairings

between elements in nums1 and nums2. Specifically, we use a bitmask to represent the elements from nums2 that have been

paired up with elements in nums1. The bitmask is a binary number where each bit corresponds to an element in nums2. If a bit is

set (i.e., it is 1), it means the corresponding element in nums2 has been used in a pairing.

as compared to a naive approach that might solve the same subproblems repeatedly.

track of which elements in nums2 have already been paired up with elements in nums1.

We start with an array f to keep track of the minimum XOR sum that can be achieved with each possible bitmask. Initially, f [0]

nums1 that we are considering pairing up. Now, for each bit that is set in the current bitmask, we try unsetting it (which means we're considering that the j-th element in nums2 could be paired with the k-th element in nums1) and calculate the XOR of nums1[k] and nums2[j], and add it to the minimum XOR sum stored in f for the bitmask without the j-th bit set. This is done for each set bit in the bitmask to find the minimum XOR sum possible for that bitmask and store it back in f. Finally,

Then, we iterate over all possible bitmasks. For each bitmask, we figure out how many bits are set; this tells us the position (k) in

(representing no elements paired) is set to 0 and all other entries are set to infinity because we haven't computed them yet.

f[-1] (which corresponds to all elements in nums2 being paired up) will contain the minimum XOR sum we can achieve, and that's the value we return as the solution.

The solution approach for minimizing the XOR sum of two integer arrays is a dynamic programming approach that utilizes bit masking to explore the state space of possible pairings. The algorithm uses the following concepts: **<u>Dynamic Programming</u>** (DP): The idea is to break the problem into overlap subproblems and use DP to remember results of already solved subproblems, such that each subproblem is solved only once. This significantly reduces the computation time

Bitmasking: This technique is used to represent the pairing state of elements from the second array nums2. A bitmask is an

array that uses each individual bit to represent a binary state (on/off, used/not used). In this case, we are using it to keep

Now, let's go through the code:

of nums1.

since we are zero-indexed.

Solution Approach

Next, the algorithm iterates over all possible indices j in the range of 0 to n-1 to find the index where the bit is set in the current mask (checked by i >> j & 1) which means that the j-th element of nums2 is considered for pairing up with the k-th element

For every masked state (ranging from 1 to 2ⁿ - 1, where n is the size of the arrays), the bit count is retrieved. Here

i.bit_count() - 1 calculates the number of set bits in the bitmask and decrements it by one to get the correct index in nums1

Then, the XOR of nums1[k] and nums2[j] is computed and added to the previously computed value of f[i ^ (1 << j)] (which represents the minimum XOR sum for the state where j-th bit is not included in the mask). The algorithm selects the minimum of these sums and updates the DP table at f[i]. The DP table f is a one-dimensional array with 2ⁿ elements (since this is the number of possible states for n bits), initialized

with infinity (inf) to represent that those XOR sums have not been computed yet, except for the base case f[0] which is set to

Finally, after the algorithm iterates through all subproblems, f[-1] will hold the minimum XOR sum which is returned as the

solution. Note that f[-1] is Python's way of accessing the last element of the list, which corresponds to the mask where all bits

4. For each set bit j in the current bitmask, calculate the new XOR sum and update f[i] with the minimum value obtained. 5. Return f[-1] as the minimum XOR sum after considering all pairings. This dynamic programming solution ensures that the minimum XOR sum for any possible pairing is calculated efficiently and accurately by considering all possible combinations without redundant computations.

We iterate over all possible bit masks from 1 to $2^n - 1$ to calculate the minimum XOR sum for all pairings, where n is the size of the input arrays. Since our arrays have 2 elements, we iterate over masks '01' and '10', representing the different

The bit count minus one is 0 (1.bit_count() - 1) which means we are looking to pair the first element of nums1 with

First, we initialize our DP table f with values set to infinity and f[0] to 0 because at the start, no elements are paired, so the

pairings. f[1] will track the results for mask 01 (where the second element of nums2 is used, but not the first).

Pair nums1[0] with nums2[1] (bit set at position 1 in mask 01)

Updating DP Table f with Subproblem solutions

So we update f[1] to min(inf, 5) which is 5.

Consider the bit mask 10 (binary for 2):

of input arrays, we stick to the subproblem results.

Consider the bit mask 01 (binary for 1):

So f = [0, inf, inf, inf], corresponding to the 2-bit masks from 00 to 11.

0 because no elements are paired and thus the XOR sum is 0.

Here is a simplified outline of the algorithm applied in the code:

1. Initialize the DP table f with inf and set f[0] to 0.

2. Iterate over all possible bit masks from 1 to 2ⁿ - 1.

Example Walkthrough

Initialization

XOR sum is 0.

Iterating Over Bit Masks

elements of nums2.

2.

Python

class Solution:

Returning the Result

Solution Implementation

length = len(nums2)

memo[0] = 0

equates to min(5, 3) = 3.

are set and all elements in nums2 have been paired with elements in nums1.

3. Calculate the current position k in nums1 based on the bitmask using bit count.

The bit count minus one is 0 (1.bit_count() - 1) which means we're again pairing the first element of nums1. Pair nums1[0] with nums2[0] (bit set at position 0 in mask 10)

XOR sum is 1 XOR 4 = 5. With the base case f[0] = 0, the result is 0 + 5 = 5.

XOR sum is 1 XOR 2 = 3. With the base case f[0] = 0, the result is 0 + 3 = 3.

f[2] will track results for mask 10 (where the first element of nums2 is used, but not the second).

Let's take nums1 = [1, 3] and nums2 = [2, 4] as an example to illustrate the solution approach.

So we update f[2] to min(inf, 3) which is 3. Finding the Minimum XOR Sum Now, we consider all elements as paired (mask 11 represents all bits set), which in this simplified example means we've

already made our optimal pairings. We'd use f[1] and f[2] to calculate the XOR sum for mask 11, but as it's beyond the size

Since we compared all possible combinations, we already have our answer and do not need this step for input arrays of size

The minimum XOR sum after rearranging nums2 is the minimum of f[1] and f[2]. In this case, it is min(f[1], f[2]) which

So, the answer for nums1 = [1, 3] and nums2 = [2, 4] after rearranging nums2 for the minimum XOR sum is 3.

This walk-through covers the dynamic programming solution approach involving bit masks to optimize the XOR sum between two arrays by finding the best possible rearrangement of elements in nums2.

def minimum xor sum(self, nums1: List[int], nums2: List[int]) -> int:

Initialize a memoization table with infinity values,

Base case: the minimum XOR sum for an empty subset is 0

the index k in nums1 that is being considered

memo[bitmask] = min(memo[bitmask],

Count the number of bits set in bitmask to determine

Check all elements of nums2 by iterating over bits of bitmask

The last element of memo contains the minimum XOR sum for the full set

// Find the current number of bits set to 1 in the bitmask `i`.

// Check if the i-th bit in the mask `i` is set to 1.

// Initialize the `dp` array with the max possible values (using shift to get 2^30).

representing the minimum XOR sum for each subset

Determine the length of the second list

Iterate over all possible subsets of nums2

public int minimumXORSum(int[] nums1, int[] nums2) {

// Get the length of the array.

for (int i = 0; i < (1 << n); ++i) {

for (int j = 0; j < n; ++j) {

if ((i & (1 << j)) != 0) {

int bitCount = Integer.bitCount(i) - 1;

int prevState = i ^ (1 << j);</pre>

int minimumXORSum(vector<int>& nums1, vector<int>& nums2) {

int n = nums1.size(); // Size of the input vectors

// dp[(1 << n) - 1] contains the answer for the full set

// Helper function that returns the count of set bits in the binary representation of i

def minimum xor sum(self, nums1: List[int], nums2: List[int]) -> int:

Initialize a memoization table with infinity values,

Base case: the minimum XOR sum for an empty subset is 0

the index k in nums1 that is being considered

and update the memo table accordingly

previous_bitmask = bitmask ^ (1 << j)</pre>

different combinations of pairings between elements in nums1 and nums1.

and the inner loop iterates over every individual element in nums2.

Count the number of bits set in bitmask to determine

Check all elements of nums2 by iterating over bits of bitmask

If the i-th bit of bitmask is set, calculate the potential XOR sum

Clear the i-th bit to find the XOR sum of the previous subset

Update the memo table entry for the current bitmask with the minimum

XOR sum obtained by either taking the current element from nums2 or not

memo[previous_bitmask] + (nums1[k] ^ nums2[j]))

representing the minimum XOR sum for each subset

return dp[(1 << n) - 1];

i = i + (i >>> 8);

return i & 0x3f;

class Solution:

i = i + (i >>> 16);

memo[0] = 0

function bitCount(i: number): number {

length = len(nums2)

 $i = i - ((i >>> 1) \& 0 \times 55555555);$

 $i = (i + (i >>> 4)) \& 0 \times 0 f 0 f 0 f 0 f;$

// Binary magic to count number of 1s

i = (i & 0x333333333) + ((i >>> 2) & 0x333333333);

Determine the length of the second list

Iterate over all possible subsets of nums2

memo = [float('inf')] * (1 << length)

for bitmask in range(1, 1 << length):</pre>

k = bin(bitmask).count('1') - 1

if bitmask & (1 << j):</pre>

for j in range(length):

int n = nums1.length:

return dp[(1 << n) - 1];

memo = [float('inf')] * (1 << length)

for bitmask in range(1, 1 << length):</pre>

k = bin(bitmask).count('1') - 1

for j in range(length): # If the j-th bit of bitmask is set, calculate the potential XOR sum # and update the memo table accordingly if bitmask & (1 << j):</pre> # Clear the i-th bit to find the XOR sum of the previous subset previous bitmask = bitmask ^ (1 << j)</pre>

memo[previous_bitmask] + (nums1[k] ^ nums2[j]))

Update the memo table entry for the current bitmask with the minimum

XOR sum obtained by either taking the current element from nums2 or not

// Calculate the new state by unsetting the j-th bit from the bitmask `i`.

dp[i] = Math.min(dp[i], dp[prevState] + (nums1[bitCount] ^ nums2[j]));

// Return the minimum XOR sum for all pairs by examining the last element in `dp` array.

vector<int> dp(1 << n, INT_MAX); // Initialize the dp array with maximum integer values

// Calculate the minimum XOR sum by comparing the previous state with the new masked value.

```
int[] dp = new int[1 << n];</pre>
Arrays.fill(dp, 1 << 30);
// The starting state has a minimum XOR sum of 0.
dp[0] = 0;
// Iterate over all possible combinations of pairs.
```

return memo[-1]

Java

C++

public:

class Solution {

class Solution {

```
dp[0] = 0; // Initial state: no numbers are paired, so the XOR sum is 0
        // Iterate over all possible states
        for (int i = 0; i < (1 << n); ++i) {
            // k represents the number of elements already included from nums1
            int k = builtin popcount(i) - 1:
            // Iterate over all elements in nums2
            for (int j = 0; j < n; ++j) {
                // Check if the i—th element in nums2 has already been paired
                if (i & (1 << j)) {
                    // If paired, calculate the new value for the dp state
                    // This is done by removing the i-th element from the current state (using XOR)
                    // Then, add the XOR of nums1[k] and nums2[j] to the dp value of the previous state
                    // Update the dp value with the minimum result between its current value and the new calculated value
                    dp[i] = min(dp[i], dp[i ^ (1 << j)] + (nums1[k] ^ nums2[j]));
        // Return the result for the state where all elements are included
        return dp[(1 << n) - 1];
};
TypeScript
function minimumXORSum(nums1: number[], nums2: number[]): number {
    const n = nums1.length; // length of the arrays
    const dp: number[] = Array(1 << n).fill(1 << 30); // dynamic programming array initialized with high values</pre>
    dp[0] = 0; // base case: XOR sum is 0 when there are no numbers to pair
    // Iterate over all possible subsets of pairs created from nums2
    for (let i = 0; i < (1 << n); ++i) {
        const bitsSet = bitCount(i) - 1; // calculate how many bits are set in i
        // Try matching each element in nums2 with nums1 based on bits set
        for (let j = 0; j < n; ++j) {
            if (((i >> i) & 1) === 1) { // if the i-th bit is set
                // Calculate new minimum XOR for the new subset by toggling i-th bit
                dp[i] = Math.min(dp[i], dp[i ^ (1 << j)] + (nums1[bitsSet] ^ nums2[j]));
```

memo[bitmask] = min(memo[bitmask]. # The last element of memo contains the minimum XOR sum for the full set return memo[-1]

Time and Space Complexity

Time Complexity

The time complexity can be analyzed by looking at the two nested loops in which the outer loop iterates over all subsets of nums2

The code provided is a solution to the Minimum XOR Sum problem using dynamic programming with bit masking to represent

• The outer loop runs for 2ⁿ iterations because it loops over all possible subsets of a set with n elements, which are represented as bit masks. • The inner loop runs for n iterations because it checks each position in the bit mask to see if it is set (which corresponds to nums2[j] being selected).

Space Complexity The space complexity is primarily determined by the storage of the array f, which has a length of 2ⁿ to represent all possible

So, the space complexity is $O(2^n)$ as that is the size of the array f.

combinations of matching nums2 elements with elements in nums1.

Since the inner loop operates within the outer loop, the total time complexity is $O(n * 2^n)$.