2605. Form Smallest Number From Two Digit Arrays

other array. We ensure to test both possible concatenations and choose the smaller one.

concatenate them in both possible ways, returning the smaller resulting number.

contains at least one digit from each array (nums1 and nums2). Here's a breakdown of this approach:

Problem Description

Easy

Hash Table

Enumeration

The task here involves finding the smallest number that comprises at least one digit from each of the two given arrays, nums1 and

build the smallest number possible using only one digit from each array, without repeating any digit. To achieve the smallest number, we generally should start with the smallest digit from the two given arrays. However, if both arrays contain a common digit, we can use just this digit as it fulfills the criteria of containing at least one digit from each array

nums2. The arrays contain unique digits, meaning each digit from 0 to 9 appears at most once in each array. We are looking to

while ensuring the number is as small as possible. If there are no common digits, we have to take the smallest digit from each array and combine them to form a two-digit number, carefully arranging them in an order that results in the smallest possible number. Intuition

The solution is intuitive once we understand the problem's requirements. If there is a common digit, our job becomes very

straightforward; we just output this digit. However, if there isn't, we need to create a two-digit number using the smallest digit

from each array.

Solution 1: Enumeration In this approach, we compare each digit from one array to every digit in the other array to check for common digits. If we find a common digit, we return it; if not, we proceed with concatenating the smallest digit from one array with the smallest from the

Solution Approach

x is set to 1.

* 10 + a).

Initializing Masks:

Building Bitmasks:

 $mask2).bit_length() - 1).$

Solution 2: Hash Table or Array + Enumeration Using a hash table or an array allows us to keep track of which digits are present in each array more efficiently. We need a fixedsized structure (since digits only go from 0 to 9), and we check each digit against this structure. The moment we find a common digit, we conclude the search and return it. If there are no common digits, we again proceed to find the smallest two digits and

Solution 3: Bit Operation

in nums1 and nums2, respectively. **Building Bitmasks**: We iterate through each number x in both nums1 and nums2 separately and update the corresponding masks using an 'OR' operation (|=1 << \times). The 'OR' operation ensures that the bit at the position corresponding to the digit

The solution implements the third approach, which leverages bit manipulation to efficiently determine the smallest number that

Initializing Masks: We initialize two bitmask variables, mask1 and mask2, to zero. These will represent the presence of digits

Finding Common Digits: Once we have our bitmasks, we check for common digits by performing a bitwise 'AND' operation

- between mask1 and mask2 (mask = mask1 & mask2). If any common digit exists, mask will not be zero. The common digit can be found by the position of the last 1 in mask, which is calculated as (mask & -mask).bit_length() - 1. Extracting Smallest Non-Common Digits: If mask is zero, it means there are no common digits. We find the position of the
- last 1 in both masks mask1 and mask2 to get the smallest digits a and b from each array. We use the bitwise 'AND' operation with the two's complement of each bitmask to isolate the last 1 bit ((mask1 & -mask1).bit_length() - 1 and (mask2 & -

Forming the Smallest Number: Depending on whether a common digit exists, the smallest number is either the position of

the last 1 in mask or the minimum of two numbers created by concatenating a and b in different orders (min(a * 10 + b, b

This bit manipulation method is very efficient because it reduces the problem of finding digits and comparing them to simple bitwise operations, which are performed quickly at the hardware level. Instead of needing to store and search through a hash table or array, this approach makes use of existing integer operations to encode the same information in a very compact space.

The space complexity of this algorithm is constant, 0(1), because the bitmasks do not grow with the size of the input arrays.

The time complexity is linear, 0(m + n), where m and n are the lengths of nums1 and nums2, because we only need single

passes through each array to build the bitmasks. **Example Walkthrough** Let's illustrate the solution approach using a small example:

 We iterate through nums1, updating mask1 for each digit: ∘ For digit 4, the binary representation of 1 << 4 is 10000. Performing mask1 |= 10000 results in mask1 = 10000. ∘ For digit 3, the binary representation of 1 << 3 is 01000. Performing mask1 |= 01000 updates mask1 to 11000.

• We look for common digits by performing mask = mask1 & mask2, resulting in mask = 0000000. Since there are no common digits (mask = 0), we

• The smallest digit in nums1 can be found by isolating the last 1 in mask1, which is 00001000. We get its bit position using (mask1 & -

So, the smallest number comprising at least one digit from each array using the bit operation method is the number 13.

mask1).bit_length() - 1. In this case, mask1 & -mask1 equals 00001000, whose bit_length() is 4, but we subtract 1 for zero-based

• Similarly, for mask2 equaling 100010, (mask2 & -mask2).bit_length() - 1 returns the bit position of the last 1, which is 1 in this case. So, b =

This example demonstrates how the bit manipulation method outlined in the solution approach can be applied to efficiently solve

∘ For digit 5, the binary representation of 1 << 5 is 100000. Performing mask2 |= 100000 results in mask2 = 100000.

• For digit 1, the binary representation of 1 << 1 is 00010. Performing mask2 |= 00010 updates mask2 to 100010.

Extracting Smallest Non-Common Digits:

of these two is 13.

the given problem.

Python

Finding Common Digits:

indexing, so a = 3.

Suppose we have two arrays: nums1 = [4, 3] and nums2 = [5, 1].

We start by initializing two bitmasks, mask1 and mask2, to zero (0).

• Then, we iterate through nums2, updating mask2 for each digit:

At this point, we have mask1 = 11000 and mask2 = 100010.

move on to extracting the smallest non-common digit from each array.

1. Forming the Smallest Number: • Since there's no common digit, we create two possible numbers by combining a and b: a * 10 + b = 31 and b * 10 + a = 13. The smallest

Iterate over the first list and update the mask1 to track the numbers.

Iterate over the second list and update the mask2 to track the numbers.

Create a mask to find common elements by applying bitwise AND on mask1 and mask2.

Find the lowest set bit for each mask to get the smallest number from each list.

// Iterate through nums2 and set corresponding bits in the mask for nums2

// If there are no common numbers, find the smallest numbers in each array

return Integer.numberOfTrailingZeros(commonMask);

// in both possible orders and return the smallest result

int smallestNums1 = Integer.numberOfTrailingZeros(maskNums1);

int smallestNums2 = Integer.numberOfTrailingZeros(maskNums2);

smallest num1 = (mask1 & -mask1).bit length() - 1

smallest_num2 = (mask2 & -mask2).bit_length() - 1

Return the smallest two-digit number.

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

maskNums1 |= 1 << num;

maskNums2 |= 1 << num;

for (int num : nums2) {

if (commonMask != 0) {

for (int num : nums2) {

if (commonMask) {

let bitMask2: number = 0;

for (const num of nums1) {

for (const num of nums2) {

if (i === 0) {

 $temp = i \ll 4$;

temp = i << 2;

if (temp !== 0) {

i = temp;

if (temp !== 0) {

position -= 4;

for num in nums1:

for num in nums2:

mask1 |= 1 << num

bitMask2 |= 1 << num;

// Calculate the common bits between both masks.

function numberOfTrailingZeros(i: number): number {

return 32; // Special case where i is 0.

bitMask1 |= 1 << num;

mask2 |= 1 << num;

int commonMask = mask1 & mask2;

// Intersection mask to find common elements

return __builtin_ctz(commonMask);

// If there's a common element, return the smallest one

// Create a bitmask to represent the presence of numbers in nums1.

// Create a bitmask to represent the presence of numbers in nums2.

// Return the smallest two-digit number that can be formed from the inputs.

// Function to count the number of trailing zeros in the binary representation of a number.

Iterate over the second list and update the mask2 to track the numbers.

Compute the smallest two-digit number using the smallest elements from both lists.

Because we need the lexicographically smallest number, we calculate both combinations.

smallest_combination = $min(smallest_num1 * 10 + smallest_num2, smallest_num2 * 10 + smallest_num1)$

// If there are no common elements, find the smallest elements in each vector

#include <vector>

class Solution {

public:

#include <algorithm>

class Solution: def minNumber(self, nums1: List[int], nums2: List[int]) -> int: # Initialize two bit masks to represent the numbers present in nums1 and nums2.

mask1 = mask2 = 0

for num in nums1:

for num in nums2:

mask1 |= 1 << num

mask2 |= 1 << num

Solution Implementation

- common_mask = mask1 & mask2 # If there is a common number, return the smallest one. if common mask:
- # Find the smallest number by isolating the lowest set bit and calculating its index. return (common_mask & -common_mask).bit_length() - 1 # If no common number is found, find the smallest number from each list.
- # Compute the smallest two-digit number using the smallest elements from both lists. # Because we need the lexicographically smallest number, we calculate both combinations. smallest_combination = $min(smallest_num1 * 10 + smallest_num2, smallest_num2 * 10 + smallest_num1)$
- return smallest_combination Java class Solution {
 - // Initialize bit masks for both arrays to track the numbers present int maskNums1 = 0, maskNums2 = 0; // Iterate through nums1 and set corresponding bits in the mask for nums1 for (int num : nums1) {
 - // Calculate the bitwise AND of both masks to find common numbers int commonMask = maskNums1 & maskNums2; // If there is a common number, return the smallest one
- return Math.min(smallestNums1 * 10 + smallestNums2, smallestNums2 * 10 + smallestNums1); C++

// Calculate the minimum number by concatenating the smallest numbers from both arrays

- // Function to find the minimum number by analyzing two vectors int minNumber(std::vector<int>& nums1, std::vector<int>& nums2) { int mask1 = 0; // Binary mask for the first vector int mask2 = 0; // Binary mask for the second vector // Populate the binary mask for nums1 for (int num : nums1) { mask1 |= 1 << num; // Populate the binary mask for nums2
- int smallestNums1 = builtin ctz(mask1); int smallestNums2 = __builtin_ctz(mask2); // Construct and return the minimum of the two possible two-digit numbers return std::min(smallestNums1 * 10 + smallestNums2, smallestNums2 * 10 + smallestNums1); **}**; **TypeScript** // Function to find the minimum number that can be formed from the elements of two arrays. function minNumber(nums1: number[], nums2: number[]): number { let bitMask1: number = 0;
- const commonBitMask = bitMask1 & bitMask2; // If there's a common number between nums1 and nums2, return its bit position. if (commonBitMask !== 0) { return numberOfTrailingZeros(commonBitMask); // Find the smallest number from each array. const smallestNumInNums1 = numberOfTrailingZeros(bitMask1); const smallestNumInNums2 = numberOfTrailingZeros(bitMask2);

return Math.min(smallestNumInNums1 * 10 + smallestNumInNums2, smallestNumInNums2 * 10 + smallestNumInNums1);

- let position = 31; let temp = 0; // For each segment, shift `i` left and decrease position if the result is non-zero. temp = i << 16; **if** (temp !== 0) { position -= 16; i = temp;temp = $i \ll 8$; **if** (temp !== 0) { position -= 8; i = temp;
- position -= 2; i = temp;// Return the number of trailing zeros by examining the final bit position. return position - ((i << 1) >>> 31); class Solution: def minNumber(self, nums1: List[int], nums2: List[int]) -> int: # Initialize two bit masks to represent the numbers present in nums1 and nums2. mask1 = mask2 = 0# Iterate over the first list and update the mask1 to track the numbers.
 - mask2 |= 1 << num # Create a mask to find common elements by applying bitwise AND on mask1 and mask2. common_mask = mask1 & mask2 # If there is a common number, return the smallest one. if common mask: # Find the smallest number by isolating the lowest set bit and calculating its index. return (common_mask & -common_mask).bit_length() - 1 # If no common number is found, find the smallest number from each list. # Find the lowest set bit for each mask to get the smallest number from each list. smallest num1 = (mask1 & -mask1).bit length() - 1
 - The time complexity of the code is 0(m + n) where m is the length of nums1 and n is the length of nums2. This is because the code iterates through each of the two lists exactly once to create two bitmasks. The bitwise OR operations inside the loops take

return smallest_combination

Time and Space Complexity

smallest_num2 = (mask2 & -mask2).bit_length() - 1

Return the smallest two-digit number.

constant time per element. After creating the bitmasks, the subsequent operations involving bitwise AND and finding the rightmost set bit also execute in constant time, as they are not dependent on the size of the input but instead on the fixed size of an integer (typically 32 or 64 bits in modern architectures).

The space complexity of the code is 0(1). This constant space usage comes from the fact that no matter the size of the input lists, the code only uses a fixed number of integer variables (mask1, mask2, mask, a, and b). These variables do not scale with the input size, resulting in constant space consumption.