# 2191. Sort the Jumbled Numbers

original index of that number in the nums array.

nums by their order determined by the mapped values.

and the original index of the number in the nums array.

mapping[0] as its mapped value.

[nums[i] for \_, i in arr].

## **Problem Description**

Medium <u>Array</u> <u>Sorting</u>

This mapping is given as an array where mapping[i] = j means the digit i should be mapped to digit j. For example, if mapping[1] = 5, every 1 in any number will be treated as 5. The task is to transform a list of integers (nums) according to this new

The problem presents a scenario where we have a custom mapping for digits from 0 to 9, implying a shuffled decimal system.

digit mapping, and then sort the list based on the new "mapped" values. After the transformation, the sorted list should maintain the order of numbers that have the same mapped value as they appear in

the original nums list. It is important to note that while the sorting is done based on the mapped values, the final sorted list should contain the original numbers, not their mapped counterparts. Intuition

## The intuition behind the implemented solution is to simulate this new decimal system by creating a corresponding mapped value

for each number in the given list nums. The first step is to convert each number to its mapped value according to the provided mapping. This means replacing each digit of the number with its correspondent in the mapping. While creating these mapped values, we have to handle them carefully to retain the original order of numbers. Hence, for each

return the original numbers in their new sorted order. The next step is to sort these pairs (which contain the mapped value and the original index) by the mapped value. If this results in

number in nums, we pair it with its index in the array; that way, after sorting the numbers by their mapped values, we can still

ties (i.e., two numbers having the same mapped value), the relative order of their indices will resolve which should come first,

achieving "stable sorting." Finally, we reconstruct the sorted list of original numbers by taking the second element from each sorted pair, which is the

Note that the way the mapped values are computed respects the magnitude of each digit in the original number (i.e., the tens

place is still the tens place after mapping, the hundreds place is still the hundreds place, and so forth). This means we have to multiply the mapped digit by its place value (k) as we form the new number y. The k value is started at 1 and is multiplied by 10 for each digit we move to the left.

Solution Approach

After all mapped values and their original indices are sorted, we can then create and return a list of the original numbers from

The solution provided follows a mapping and sorting approach to achieve the final sorted array based on custom mapped values. Firstly, the solution leverages a list called arr to store tuples. Each tuple contains two values: the mapped value of the number

### Here is a step-by-step breakdown of the algorithm: Iterate through each number in the nums array with its index. For each number x and its index i:

∘ Initialize a variable k which represents the current digit's place value (e.g., 1 for units, 10 for tens, etc.). For each digit v in the number x (obtained by repeatedly using divmod to split off the last digit):

Multiply the mapped digit by its current place value k and add it to y, building up the new mapped number digit by digit.

guarantees that arr will be sorted by y but maintain the relative order of elements that have the same y value.

 Multiply k by 10 to update the place value for the next digit to the left. After finishing the mapping for a number, append a tuple (y, i) to the list arr, where y is the mapped value obtained and i is the index of the number in nums.

o Initialize a variable y to be the mapped value of 0. This variable will accumulate the final mapped number. Note that if x is zero, we set y to

Sort the list arr by the first tuple element, which is the mapped value. Python's default sorting algorithm is stable, which

Calculate the mapped digit by using the given mapping (i.e., mapping [v]).

The overall time complexity of the solution is dominated by the sorting step, which is O(n log n), where n is the length of the nums array. The space complexity is O(n) due to the additional list arr used for sorting purposes.

Create the final sorted list by extracting and appending the original numbers from nums using the sorted indices found in arr:

standard manner while maintaining stability in sorting. **Example Walkthrough** 

Let's consider an example to walk through the solution approach. Suppose we have a mapping array and a list of numbers as

This solution pattern effectively applies a custom sort key defined by a map, which is a common way to sort elements in a non-

• mapping = [2, 1, 4, 3, 6, 5, 8, 7, 0, 9]• nums = [990, 332, 981] Based on the mapping, each digit in the numbers from nums should be converted as follows:

#### • '3' maps to '3' '4' maps to '6'

• '0' maps to '2'

• '1' maps to '1'

• '2' maps to '4'

• '5' maps to '5'

• '6' maps to '8'

• '7' maps to '7'

follows:

```
• '8' maps to '0'
• '9' maps to '9'
```

We'll initiate the list arr to store tuples of the mapped value and the original index.

 Mapping '9' to '9', '8' to '0', and '1' to '1', the mapped value of 981 is 901. • Append to arr the tuple (901, 2).

We start iterating through nums. For 990, its index i is 0:

Mapping '9' to '9' and '0' to '2', the mapped value of 990 is 992.

Mapping '3' to '3' and '2' to '4', the mapped value of 332 is 334.

Append to arr the tuple (992, 0).

• Append to arr the tuple (334, 1).

Lastly, for 981, its index i is 2:

their original form.

**Python** 

Solution Implementation

while num:

mapped\_with\_index.sort()

int n = nums.length;

Next, for 332, its index i is 1:

- Now arr looks like this: [(992, 0), (334, 1), (901, 2)].
- After sorting, we create the final sorted list based on the original indices: [nums[i] for \_, i in arr]. Using the sorted indices, we get nums [2], nums [1], nums [0], which correspond to the numbers [981, 332, 990].

# Iterate over the input numbers' list

mapped\_num = mapping[0] if num == 0 else 0

num, digit = divmod(num, 10)

power\_of\_ten = 1 # To keep track of the decimal place

# Reconstruct the sorted list using the original indices

return [nums[i] for \_, i in mapped\_with\_index]

public int[] sortJumbled(int[] mapping, int[] nums) {

// Get the length of the nums array.

int[][] mappedWithIndex = new int[n][2];

// Iterate over the array of numbers.

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

while (originalNum > 0) {

for index, num in enumerate(nums):

class Solution: def sortJumbled(self, mapping: List[int], nums: List[int]) -> List[int]: # List to hold tuples of the mapped value and its original index mapped\_with\_index = []

# If the number is 0, get the mapped value for 0, else start with 0

# Decompose the number into digits and map using the provided mapping

# Sort the list according to the mapped numbers, stable for identical values

// Create a 2D array to store the mapped number and the original index.

// Map each digit of the original number based on the 'mapping' array.

int digit = originalNum % 10; // Retrieve the last digit.

// Store the mapped number and the original index in the array.

int originalNum = nums[i]; // Original number from nums.

originalNum /= 10; // Drop the last digit.

// Return the sorted numbers as per the jumbled mapping order

function sortJumbled(mapping: number[], nums: number[]): number[] {

// Calculate the mapped value of an individual number

let mappedValue = originalNum === 0 ? mapping[0] : 0;

// Store the mapped value along with the original index

return mappedNums.map(mappedPair => nums[mappedPair[1]]);

mappedNums.sort((a, b) => (a[0] === b[0] ? a[1] - b[1] : a[0] - b[0]));

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

# List to hold tuples of the mapped value and its original index

// Map the sorted array back to the original numbers using their stored indices

// If the number is zero, map it directly using the mapping array;

let positionMultiplier = 1; // Used to place the digit at the correct position

mappedWithIndex[i] = new int[] {mappedNum, i};

So, the transformed and sorted list is [981, 332, 990] based on the new "mapped" values, yet the numbers themselves retain

We sort arr using the first element of the tuples, so the list becomes [(901, 2), (334, 1), (992, 0)].

# Map the digit, adjust decimal place and add to the mapped number mapped\_num = mapping[digit] \* power\_of\_ten + mapped\_num power\_of\_ten \*= 10 # Increase the decimal place # Append the tuple of mapped number and original index to the list mapped\_with\_index.append((mapped\_num, index))

int mappedNum = originalNum == 0 ? mapping[0] : 0; // Map the number based on mapping rules.

mappedNum += placeValue \* mapping[digit]; // Map the digit and add to mappedNum considering the place value.

int placeValue = 1; // To reconstruct the mapped number based on individual digits.

placeValue \*= 10; // Move to the next place value (tens, hundreds, etc.).

```
import java.util.Arrays;
class Solution {
   // Method to sort the array nums based on a custom mapping.
```

Java

```
// Sort the array 'mappedWithIndex' based on the mapped numbers and indices.
       Arrays.sort(mappedWithIndex, (a, b) ->
            a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]); // If mapped numbers are equal, compare index.
       // Prepare the final sorted array based on the mapped values.
       int[] sortedArray = new int[n];
        for (int i = 0; i < n; ++i) {
            sortedArray[i] = nums[mappedWithIndex[i][1]];
       // Return the sorted array.
       return sortedArray;
C++
class Solution {
public:
    vector<int> sortJumbled(vector<int>& mapping, vector<int>& nums) {
        int numsSize = nums.size(); // Number of elements in nums
       vector<pair<int, int>> mappedAndIndexPairs(numsSize); // Pair to store the mapped value and original index
       // Transform each number as per the mapping and associate it with its original index
        for (int i = 0; i < numsSize; ++i) {</pre>
            int originalNum = nums[i];
            int mappedNum = originalNum == 0 ? mapping[0] : 0; // If the number is 0, directly map it
            int placeValue = 1; // Represents the place value in the mapped number
            // Decompose the number into its digits and transform it according to the mapping
            while (originalNum > 0) {
                int digit = originalNum % 10; // Get the last digit
                mappedNum += placeValue * mapping[digit]; // Map the digit and add to the mapped number
                originalNum /= 10; // Remove the last digit from the original number
                placeValue *= 10; // Move to the next place value
            // Save the pair of mapped number and original index
            mappedAndIndexPairs[i] = {mappedNum, i};
       // Sort the pairs. The order is firstly by the mapped number, and then by the original index
       sort(mappedAndIndexPairs.begin(), mappedAndIndexPairs.end());
       // Extract the numbers from the sorted pairs, preserving the new order
       vector<int> sortedNums;
        for (auto& pair : mappedAndIndexPairs) {
            sortedNums.push_back(nums[pair.second]);
```

```
// Decompose the number and map its digits using the given mapping
for (; originalNum > 0; originalNum = Math.floor(originalNum / 10), positionMultiplier *= 10) {
    // Add the mapped digit multiplied by its positional value to 'mappedValue'
    mappedValue += mapping[originalNum % 10] * positionMultiplier;
```

class Solution:

**Time Complexity** 

term is O(N log N).

**Space Complexity** 

mapped with index = []

**}**;

**TypeScript** 

return sortedNums;

const numsLength = nums.length;

const mappedNums: number[][] = [];

let originalNum = nums[i];

for (let i = 0;  $i < numsLength; ++i) {$ 

mappedNums.push([mappedValue, i]);

// Loop through all numbers in the 'nums' array

// else initialize it to zero to build upon

```
# Iterate over the input numbers' list
       for index, num in enumerate(nums):
           # If the number is 0, get the mapped value for 0, else start with 0
           mapped_num = mapping[0] if num == 0 else 0
           power_of_ten = 1 # To keep track of the decimal place
           # Decompose the number into digits and map using the provided mapping
           while num:
               num, digit = divmod(num, 10)
               # Map the digit, adjust decimal place and add to the mapped number
               mapped_num = mapping[digit] * power_of_ten + mapped_num
               power_of_ten *= 10 # Increase the decimal place
           # Append the tuple of mapped number and original index to the list
           mapped_with_index.append((mapped_num, index))
       # Sort the list according to the mapped numbers, stable for identical values
       mapped_with_index.sort()
       # Reconstruct the sorted list using the original indices
       return [nums[i] for _, i in mapped_with_index]
Time and Space Complexity
```

// Sort the 'mappedNums' array based on the mapped values, and if those are equal, by the original indices

#### 1. Iterating over each element in nums: 0(N), where N is the number of elements in nums. 2. Within the loop, transforming each number based on the mapping. In the worst case, the number of digits D in a number is proportional to the logarithm of the number (log10(x)), resulting in a complexity of O(D). Since D is small compared to N for a reasonable range of integers, this can

3. Sorting the transformed array, which would take O(N log N) time. Thus, combining these factors, the overall time complexity is O(N \* log(M) + N log(N). If N is much larger than M, the dominant

be approximated to O(log(M)), where M is the maximum number in nums.

The time complexity of the code is determined by a few factors:

```
The space complexity of the code involves:
1. The additional array arr used to store the transformed tuples, which adds a space complexity of O(N).
```

There are no other significant uses of additional space, so the total space complexity is O(N).