Documentation

Problem Description

The "Create Maximum Number" problem involves constructing the largest possible number of length k by selecting digits from two given arrays, nums1 and nums2 while maintaining the relative order of digits within each variety. The challenge lies in combining these digits optimally to create the maximum number, given the constraint that $k \leq m + n$, where mm and n are the lengths of nums1 and nums2. This task has practical applications in areas requiring sequence optimization or maximizing numerical representations.

Approach Overview

The solution employs a combination of greedy algorithms and dynamic iteration. It involves selecting subsequences of specified lengths from both arrays and merging them to form candidate results. To find the optimal solution, we iterate over all possible splits of k, where a portion of digits is taken from one array and the rest from the other. For each split, we calculate the maximum subsequences from both arrays, merge them into a single sequence, and retain the best result.

Subsequence Selection

A key solution component is the max_subsequence function, which extracts the largest subsequence of a specified length while preserving the relative order of elements. This is achieved using a stack-based greedy algorithm that iterates over the array, discarding smaller elements when there is room to include larger ones. This ensures that the subsequence contains only the most promising digits. For example, given nums = [3, 4, 6, 5] and length 33, the resulting subsequence is [4, 6, 5].

Merging Subsequences

Once the subsequences are selected, they are merged into a single sequence using a merge function. This function compares the remaining elements in both subsequences and appends the larger one at each step, breaking ties by looking ahead at the subsequent elements. This ensures that the merged sequence is lexicographically the largest possible. For instance, merging [6, 7] and [6, 0, 4] produces [6, 7, 6, 0, 4].

Iterative Exploration

To find the final result, the algorithm iterates over all valid splits of k, where ii digits are taken from nums1 and k−ik-i digits are taken from nums2. For each split, the respective subsequences are generated, merged, and compared against the current maximum candidate. This exhaustive exploration ensures that no combination is overlooked, and the global maximum is achieved.

Complexity Analysis

The algorithm is efficient, with a time complexity of O (k·(m + n), where m and n are the lengths of nums1 and nums2, and k is the target length. This accounts for extracting subsequences and merging them for each split. The space complexity is O (k), as intermediate results are stored temporarily during computation. This design ensures the solution remains feasible for the input size constraints.

Edge Cases

The solution handles various edge cases, such as when all digits are taken from one array, overlapping or identical values in both arrays and situations where k approaches the sum of array lengths. For example, with nums1 = [6, 7], nums2 = [6, 0, 4], and k=5k = 5, the algorithm correctly produces [6, 7, 6, 0, 4].

Conclusion

By combining greedy selection, iterative exploration, and efficient merging, the algorithm provides a robust solution to the "Create Maximum Number" problem. It effectively balances performance and correctness, making it well-suited for both small and large inputs. The structured approach ensures clarity in implementation and reliability in achieving the desired results.