1035. Uncrossed Lines

Medium Array Dynamic Programming Leetcode Link

Problem Description

In this problem, we are given two arrays `nums1` and `nums2`, and we are asked to find the maximum number of "uncrossed lines" that can be drawn connecting equal elements in the two arrays such that each line represents a pair of equal numbers and does not cross over (or intersect) any other line. A line connecting `nums1[i]` and `nums2[j]` can only be drawn if `nums1[i] == nums2[j]` and it doesn't intersect with other connecting lines. A line cannot intersect with others even at the endpoints, which means each number is used once at most in the connections.

Essentially, this is a problem of finding the maximum number of pairings between two sequences without any overlaps. The problem can also be thought of as finding the longest sequence of matching numbers between `nums1` and `nums2` taking the order of numbers in each list into account.

Intuition

The intuition for solving this problem comes from a classic dynamic programming challenge — the Longest Common Subsequence (LCS) problem. In the LCS problem, we aim to find the longest subsequence common to two sequences which may be found in different orders within the two sequences. This problem is similar, as connecting lines between matching numbers creates a sequence of matches.

To solve the LCS problem, we can construct a two-dimensional `dp` (short for dynamic programming) array, where `dp[i][j]` represents the length of the longest common subsequence between `nums1[:i]` and `nums2[:j]` — that is, up to the `i-th` element in `nums1` and `j-th` element in `nums2`. We initialize a matrix with zeros, where the first row and first column represent the base case where one of the sequences is empty (no common elements).

The following relation is used to fill the `dp` matrix:

- If `nums1[i - 1] == nums2[j - 1]`, we have found a new common element. We take the value from the top-left diagonal `dp[i - 1][j - 1]` and add 1 to it because we have found a match, and add it to the current `dp[i][j]`.
- If `nums1[i - 1] != nums2[j - 1]`, no new match is found, so we take the maximum of the previous subproblem solutions. That is, we check whether the longest sequence is obtained by including one more element from `nums1` (use the value in `dp[i - 1][j]`) or by including one more element from `nums2` (use the value in `dp[i][j - 1]`).

The dynamic programming approach ensures that we do not recount any sequence and that we build our solution in a bottom-up manner, considering all possible matches from the simplest case (empty sequences) up to the full sequences. The final answer to the maximum number of uncrossed lines is given by `dp[m][n]`, which is the value for the full length of both `nums1` and `nums2`.

Solution Approach

The solution uses a dynamic programming approach to solve the problem efficiently. The key concept here is to build up the solution using previously computed results, avoiding redundant computations.

Here are the steps outlining the implementation details:

- We first initialize a two-dimensional array `dp` with dimensions `(m+1) x (n+1)`, where `m` is the length of `nums1` and `n` is the length of `nums2`. The array is initialized with zeros. This array will help us store the lengths of the longest common subsequences found up to each pair of indices `(i, j)`.

- The `dp` array is filled up row by row and column by column starting at index `1`, because the `0`-th row and column represent the base case where either of the input sequences is empty.

- For every pair of indices `i` from `1` to `m` and `j` from `1` to `n`, we check if the elements `nums1[i - 1]` and `nums2[j - 1]` match.

  - If `nums1[i - 1] == nums2[j - 1]`, this means that we can extend a common subsequence found up to `dp[i-1][j-1]`. We add `1` to the value of `dp[i-1][j-1]` and store it in `dp[i][j]`. This represents extending the length of the current longest common subsequence by `1`.

    - Mathematically, it is represented as:
      ```
      dp[i][j] = dp[i-1][j-1] + 1
      ```

  - If `nums1[i - 1] != nums2[j - 1]`, this means that the current elements do not match, so we cannot extend the subsequence by including both of these elements. Instead, we take the maximum value between `dp[i-1][j]` and `dp[i][j-1]`.

    - This means we are either including the `i-th` element from `nums1` and not the `j-th` element from `nums2`, or vice versa, depending on which option provides a longer subsequence so far.

    - Mathematically, it is represented as:

      ```
      dp[i][j] = max(dp[i-1][j], dp[i][j-1])
      ```

- After completing the filling of the `dp` matrix, the value of `dp[m][n]` will give us the maximum number of connecting lines we can draw. This represents the length of the longest common subsequence between `nums1` and `nums2`.

By using dynamic programming, the algorithm ensures that we are only computing the necessary parts of the solution space and building upon them to find the optimal solution to the original problem.

Example Walkthrough

Let's illustrate the solution approach with small example arrays `nums1 = [2, 5, 1]` and `nums2 = [5, 2, 1, 4]`.

1. Initialize the `dp` array with dimensions `(4) x (5)`, since `len(nums1) = 3` (we add 1 for the base case) and `len(nums2) = 4` (again, we add 1 for the base case), and fill it with zeros.

```
    dp matrix:
       0  0  0  0  0
       0  0  0  0  0
       0  0  0  0  0
       0  0  0  0  0
```

2. Begin iterating through the `dp` array starting with `i = 1` (for `nums1`) and `j = 1` (for `nums2`).

3. For `i = 1` and `j = 1`, compare `nums1[0]` (which is `2`) and `nums2[0]` (which is `5`). They are not equal, so we set `dp[1][1]` to `max(dp[0][1], dp[1][0])` which is `0`.

```
    dp matrix:
       0  0  0  0  0
       0  0  0  0  0
       0  0  0  0  0
       0  0  0  0  0
```

4. Continue this for the combination of the indices, when you reach `i = 1` and `j = 2`, `nums1[0]` is `2` and `nums2[1]` is 2. They match, so we set `dp[1][2]` to `dp[0][1] + 1` which is `1`.

```
    dp matrix:
       0  0  0  0  0
       0  0  1  0  0
       0  0  0  0  0
       0  0  0  0  0
```

5. As we proceed, we come to a case where `i = 2` and `j = 1`, `nums1[1]` is `5` and `nums2[0]` is `5`. They do match, so `dp[2][1] = dp[1][0] + 1`, hence the value is `1`.

```
    dp matrix:
       0  0  0  0  0
       0  0  1  0  0
       0  1  0  0  0
       0  0  0  0  0
```

6. Continue filling the matrix using the rules described above. Finally, the `dp` matrix looks like:

```
    dp matrix:
       0  0  0  0  0
       0  0  1  1  1
       0  1  1  1  1
       0  1  1  2  2
```

7. The bottom-right value `dp[3][4]` is `2`, indicating the maximum number of "uncrossed lines" we can draw between `nums1` and `nums2` is `2`. These lines connect the pairs `(2, 2)` and `(1, 1)` from `nums1` and `nums2`, respectively.

By following these steps, the dynamic programming algorithm efficiently computes the maximum number of uncrossed lines that can be drawn between the two arrays.

Python Solution

```python
class Solution:
    def max_uncrossed_lines(self, nums1: List[int], nums2: List[int]) -> int:
        # Get the lengths of the two input lists
        len_nums1, len_nums2 = len(nums1), len(nums2)

        # Create a 2D array with dimensions (len_nums1+1) x (len_nums2+1)
        # Initially, fill it with zeros
        dp = [[0] * (len_nums2 + 1) for _ in range(len_nums1 + 1)]

        # Iterate through each element in nums1
        for i in range(1, len_nums1 + 1):
            # Iterate through each element in nums2
            for j in range(1, len_nums2 + 1):
                # If the elements in nums1 and nums2 are equal
                if nums1[i - 1] == nums2[j - 1]:
                    # Take the value from the diagonal and add 1
                    dp[i][j] = dp[i - 1][j - 1] + 1
                else:
                    # Otherwise, take the max value from either
                    # the left or the top cell
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

        # Return the value in the bottom-right corner of the matrix
        # which holds the count of the maximum number of uncrossed lines
        return dp[len_nums1][len_nums2]

```

Java Solution

```java
class Solution {
    public int maxUncrossedLines(int[] A, int[] B) {
        // Lengths of the input arrays.
        int lengthA = A.length;
        int lengthB = B.length;

        // Create a 2D array for dynamic programming.
        int[][] dpArray = new int[lengthA + 1][lengthB + 1];

        // Iterate through each element in both arrays.
        for (int indexA = 1; indexA <= lengthA; indexA++) {
            for (int indexB = 1; indexB <= lengthB; indexB++) {
                // If the current elements in both arrays match,
                // take the value from the previous indices in both arrays and add one.
                if (A[indexA - 1] == B[indexB - 1]) {
                    dpArray[indexA][indexB] = dpArray[indexA - 1][indexB - 1] + 1;
                } else {
                    // If they do not match, take the maximum value from the two possibilities:
                    // (1) one index back in the first array (indexA - 1),
                    // (2) one index back in the second array (indexB - 1).
                    dpArray[indexA][indexB] = Math.max(dpArray[indexA - 1][indexB], dpArray[indexA][indexB - 1]);
                }
            }
        }

        // The bottom-right value in the dpArray will have the count of maximum uncrossed lines.
        return dpArray[lengthA][lengthB];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxUncrossedLines(vector<int>& A, vector<int>& B) {
        // Find the size of both input vectors
        int sizeA = A.size(), sizeB = B.size();

        // Initialize a 2D vector dp to hold the max uncrossed lines count for subproblems
        vector<vector<int>> dp(sizeA + 1, vector<int>(sizeB + 1));

        // Iterate over each element in vector A (1-indexed for dp purposes)
        for (int i = 1; i <= sizeA; ++i) {
            // Iterate over each element in vector B (1-indexed for dp purposes)
            for (int j = 1; j <= sizeB; ++j) {
                // If the current elements in A and B are equal
                if (A[i - 1] == B[j - 1]) {
                    // The number of uncrossed lines is 1 plus the number for the previous elements
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    // Otherwise, take the maximum from either excluding current element in A or B
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        // Return the max uncrossed lines for the whole sequences
        return dp[sizeA][sizeB];
    }
};

```

Typescript Solution

```typescript
function maxUncrossedLines(nums1: number[], nums2: number[]): number {
    // Get the lengths of both input arrays
    const lengthNums1 = nums1.length;
    const lengthNums2 = nums2.length;
    // Initialize a 2D array for dynamic programming with all values set to 0
    const dp: number[][] = Array.from({ length: lengthNums1 + 1 }, () => new Array(lengthNums2 + 1).fill(0));

    // Iterate over both arrays starting from index 1 since dp array is 1-indexed
    for (let i = 1; i <= lengthNums1; ++i) {
        for (let j = 1; j <= lengthNums2; ++j) {
            // For each cell, set the value to the maximum of the cell to the left and the cell above
            dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            // If the current elements in nums1 and nums2 match, set dp[i][j] to the value of
            // the diagonal cell dp[i - 1][j - 1] plus 1 (representing this matching pair)
            if (nums1[i - 1] === nums2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            }
        }
    }

    // Return the bottom-right cell of the dp array, which represents the maximum number
    // of uncrossed lines (matches) between nums1 and nums2
    return dp[lengthNums1][lengthNums2];
}

```

Time and Space Complexity

The given code implements a dynamic programming solution to find the maximum number of uncrossed lines which can be drawn between two arrays. Here's the analysis of its complexities:

- **Time complexity:** The code has two nested loops, each iterating over the elements of `nums1` (of size `m`) and `nums2` (of size `n`), respectively. Therefore, the time complexity is `O(m * n)`, since every cell in the DP table `dp` of size `m+1` by `n+1` is filled exactly once.

- **Space complexity:** The space complexity is determined by the size of the table `dp`, which has `(m + 1) * (n + 1)` cells. Hence, the space complexity is `O(m * n)` as we are using an auxiliary 2D array to store the solutions to subproblems.