718. Maximum Length of Repeated Subarray Binary Search Dynamic Programming Sliding Window

Problem Description

Medium

a contiguous part of an array. The challenge is to identify the longest sequence of elements that nums1 and nums2 have in common, wherever that sequence may appear within the arrays. Thus, the key point in this challenge is to compare elements at different positions across both arrays and keep track of the length

Rolling Hash

Hash Function

The problem is to find the maximum length of a subarray that appears in both given integer arrays nums1 and nums2. A subarray is

of the current matching subarrays, equipping ourselves to identify the maximum length out of these subarrays.

Intuition

Our approach leverages a classic technique in computer science known as dynamic programming. Specifically, we use a 2D array (let's call it f) where f[i][j] represents the length of the longest common subarray ending with nums1[i-1] and nums2[j-1].

Construct a 2D list f with dimensions $(m+1) \times (n+1)$, where m and n are the lengths of nums1 and nums2, respectively. Initialize

Here's the intuition broken down into steps:

- Loop through each element in nums1 (index i) and nums2 (index j). If we find a match (nums1[i - 1] == nums2[j - 1]), this extends a common subarray. Therefore, we set f[i][j] to be f[i -
- 1] [j 1] + 1, effectively saying, "the longest common subarray ending here is one longer than the longest at the previous

all elements to 0.

indices.

- Track the maximum length found during this process with a variable ans. After exploring all elements from both arrays, ans holds the length of the longest matching subarray found.
- Solution Approach The solution implements a <u>dynamic programming</u> approach to solve the problem efficiently by avoiding the recomputation of

Let's illustrate the solution approach with small example arrays nums1 and nums2.

Suppose nums1 = [1, 2, 8, 3] and nums2 = [5, 1, 8, 3, 9].

updated to f[2][2] + 1 and f[3][3] becomes 1.

computed values to build up a solution, avoiding unnecessary recomputation.

The table dimensions will be (length_nums1 + 1) x (length_nums2 + 1).

Check if the elements at the current indices are the same.

max_length = max(max_length, dp_table[i][j])

 $dp_table = [[0] * (length_nums2 + 1) for _ in range(length_nums1 + 1)]$

Variable to hold the length of the longest common subarray.

overlapping subproblems. Let's walk through the logical steps and explain how the solution is implemented:

respectively. Here, each element f[i][j] is meant to hold the length of the longest common subarray that ends with elements nums1[i-1] and nums2[j-1].

previously computed values.

optimally efficient solution.

Nested Loops for Comparison: Utilize two nested loops to iterate over both arrays. The outer loop runs through nums1 using index i, while the inner loop runs through nums2 using index j. Both indices start from 1 since the 0-th row and column of f

Initialization: Create a 2D list f of size $(m+1) \times (n+1)$ filled with zeros, where m and n are the lengths of nums1 and nums2,

will be used as a base for the <u>dynamic programming</u> algorithm and should remain zeroes. Matching Elements: Inside the inner loop, check if the elements nums1[i-1] and nums2[j-1] match. This is important because we are looking for a common subarray, so we are only interested in matching elements.

Updating f: If a match is found, update f[i][j] to f[i-1][j-1] + 1. This step carries over the length from the previous

matching subarray and adds one for the current match. It is the core of the dynamic programming approach as it builds upon

Tracking the Maximum Length: Keep updating a variable ans with the maximum value in the 2D list f as we go, by comparing

ans with f[i][j] at each step. Following this logic, the final value held in ans after the loops complete execution will be the length of the longest common

subarray between nums1 and nums2. This is because ans is updated each time we extend the length of a subarray, and it only

at an index can be found from longest subarrays ending at previous indices) and avoids redundant calculations, providing an

keeps the maximum length encountered. The <u>dynamic programming</u> pattern here exploits the "optimal substructure" property of the problem (the longest subarray ending

Example Walkthrough

Initialization: We create a 2D list f with dimensions (4+1) x (5+1) (as nums1 has length 4 and nums2 has length 5), so f will be

Nested Loops for Comparison: Begin with nested loops; with i iterating from 1 to 4 (for nums1) and j iterating from 1 to 5 (for

Matching Elements & Updating f: When i=1 and j=2, we find that nums1[0] == nums2[1] (which is 1). So, we update f[1][2] to f[0][1] + 1. Since f[0][1]

As loops continue, no more matches are found until i=3 and j=3, where nums1[2] == nums2[2] (which is 8); f[3][3] is

(the subarray being [8, 3]).

Python

from typing import List

class Solution:

is 0, f[1][2] becomes 1.

nums2).

Finally, at i=4 and j=4, nums1[3] == nums2[3] (which is 3). Since we had a match at the previous indices (nums1[2] == nums2[2] was 8), f[i][j] becomes f[3][3] + 1. f[3][3] was 1, so now f[4][4] is 2. This is the longest subarray we've

a 5×6 grid of zeros. This grid will store the lengths of the longest common subarrays found to our point.

encountered. Tracking the Maximum Length: ans is updated each time f[i][j] is bigger than the current ans. It starts at 0 and becomes 1 after the first match, and then 2 after the last match.

The loops conclude with ans holding the value 2, which is the maximum length of a subarray that appears in both nums1 and nums2

This walk-through has demonstrated how the dynamic programming approach efficiently solves the problem by using previously

Solution Implementation

def findLength(self, nums1: List[int], nums2: List[int]) -> int: # Get the lengths of the input arrays. length_nums1, length_nums2 = len(nums1), len(nums2) # Initialize the DP table with all values set to 0.

If they are, update the DP table by adding 1 to the value # from the previous indices in both nums1 and nums2. dp_table[i][j] = dp_table[i - 1][j - 1] + 1 # Update the max_length if a longer common subarray is found.

C++

#include <vector>

using std::vector;

using std::max;

class Solution {

int maxLength = 0;

return max_length

max_length = 0

Loop through each element in nums1.

for i in range(1, length_nums1 + 1):

Loop through each element in nums2.

if nums1[i - 1] == nums2[j - 1]:

Return the length of the longest common subarray.

for j in range(1, length_nums2 + 1):

```
Java
// Class name Solution indicates that this is a solution to a problem.
class Solution {
   // Method findLength returns the length of the longest common subarray between two arrays.
    public int findLength(int[] nums1, int[] nums2) {
       // m and n store the lengths of the two input arrays nums1 and nums2 respectively.
        int m = nums1.length;
        int n = nums2.length;
       // Create a 2D array 'dp' to store the lengths of common subarrays.
       int[][] dp = new int[m + 1][n + 1];
       // Variable 'maxLen' keeps track of the maximum length of common subarrays found so far.
       int maxLen = 0;
       // Iterate over the elements of nums1 and nums2.
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j \le n; ++j) {
                // Check if elements from both arrays match.
                if (nums1[i - 1] == nums2[j - 1]) {
                    // If they match, increment the value from the previous diagonal element by 1.
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                    // Update 'maxLen' if the current length of the common subarray is greater.
                    maxLen = Math.max(maxLen, dp[i][j]);
                // If elements do not match, the length of common subarray is 0 (by default in Java).
       // Return the maximum length of common subarray found.
       return maxLen;
```

```
public:
   int findLength(vector<int>& nums1, vector<int>& nums2) {
       // Size of the input vectors
       int sizeNums1 = nums1.size();
       int sizeNums2 = nums2.size();
```

// Iterate over nums1 and nums2 vectors

for (int i = 1; i <= sizeNums1; ++i) {</pre>

#include <algorithm> // Include library for std::max

```
for (int j = 1; j <= sizeNums2; ++j) {</pre>
                // If elements match, extend the length of the common subarray
                if (nums1[i - 1] == nums2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                    // Update maxLength with the largest length found
                    maxLength = max(maxLength, dp[i][j]);
                // No need to handle the else case explicitly, as the dp array is initialized to 0s
        // Return the maximum length of common subarray found
        return maxLength;
};
TypeScript
function findLength(nums1: number[], nums2: number[]): number {
    // Get the lengths of both input arrays.
    const lengthNums1 = nums1.length;
    const lengthNums2 = nums2.length;
    // Initialize a 2D array 'dp' (dynamic programming table) with zeros.
    // The table will store lengths of common subarrays.
    const dp: number[][] = Array.from({ length: lengthNums1 + 1 }, () => new Array(lengthNums2 + 1).fill(0));
    // Variable to keep track of the maximum length of common subarray found so far.
    let maxLength = 0;
    // Iterate over both arrays to fill the dynamic programming table.
    for (let i = 1; i <= lengthNums1; ++i) {</pre>
        for (let j = 1; j <= lengthNums2; ++j) {</pre>
            // When elements at the current position in both arrays match,
            // increment the value by 1 from the diagonally previous.
```

// Create a 2D vector to store the length of longest common subarray ending at i-1 and j-1

// Initialize answer to keep track of the max length of common subarray found so far

vector<vector<int>> dp(sizeNums1 + 1, vector<int>(sizeNums2 + 1));

```
return maxLength;
from typing import List
class Solution:
   def findLength(self, nums1: List[int], nums2: List[int]) -> int:
       # Get the lengths of the input arrays.
        length_nums1, length_nums2 = len(nums1), len(nums2)
       # Initialize the DP table with all values set to 0.
       # The table dimensions will be (length_nums1 + 1) x (length_nums2 + 1).
       dp_table = [[0] * (length_nums2 + 1) for _ in range(length_nums1 + 1)]
```

Variable to hold the length of the longest common subarray.

Check if the elements at the current indices are the same.

from the previous indices in both nums1 and nums2.

dp_table[i][j] = dp_table[i - 1][j - 1] + 1

max_length = max(max_length, dp_table[i][j])

If they are, update the DP table by adding 1 to the value

Update the max_length if a longer common subarray is found.

// Update maxLength if a longer common subarray is found.

if (nums1[i - 1] === nums2[j - 1]) {

// Return the maximum length of the common subarray.

dp[i][j] = dp[i - 1][j - 1] + 1;

maxLength = Math.max(maxLength, dp[i][j]);

```
Time and Space Complexity
  The given code implements a dynamic programming approach to find the length of the longest common subsequence between
  two arrays nums1 and nums2.
```

a constant amount of work.

Time Complexity

 $max_length = 0$

return max length

Loop through each element in nums1.

Loop through each element in nums2.

if nums1[i - 1] == nums2[j - 1]:

Return the length of the longest common subarray.

for j in range(1, length_nums2 + 1):

for i in range(1, length_nums1 + 1):

Space Complexity The space complexity of the code is also 0(m * n) because it creates a 2D list f of size (m + 1) * (n + 1) to store the lengths of

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

uses two nested loops, each iterating up to the length of the respective arrays. For each pair of indices (i, j) the code performs

```
common subsequences for each index pair (i, j). This 2D list is required to remember the results for all subproblems, which is a
```

Space Complexity: 0(m * n)

typical requirement of dynamic programming approaches. In summary: • Time Complexity: 0(m * n)