1458. Max Dot Product of Two Subsequences

```
Problem Description
```

<u>Array</u> <u>Dynamic Programming</u>

In this problem, we are given two integer arrays nums1 and nums2. We need to find the maximum dot product between non-empty subsequences of nums1 and nums2 that have the same length. A subsequence is derived from the original array by potentially removing some elements without changing the order of the remaining elements. The dot product of two sequences of the same length is the sum of the pairwise products of their corresponding elements.

product is a1*b1 + a2*b2 + a3*b3 + ... + ai*bi.

To illustrate, if we have a subsequence [a1, a2, a3, ..., ai] from nums1 and [b1, b2, b3, ..., bi] from nums2, the dot

Our goal is to find such subsequences from nums1 and nums2 that when we calculate their dot product, we get the maximum possible value.

Intuition

The intuition behind the solution comes from recognizing that this problem can be solved optimally by breaking it down into

simpler subproblems. This is a hint that <u>dynamic programming</u> might be a useful approach. More specifically, we can use a 2D

Hard

the first j elements of nums2. When we compute the entry dp[i][j], we consider the following possibilities to maximize our result: 1. dp[i - 1][j] — the maximum dot product without including the current element from nums1. 2. dp[i][j - 1] — the maximum dot product without including the current element from nums2.

dynamic programming table dp where dp[i][j] represents the maximum dot product between the first i elements of nums1 and

3. dp[i - 1][j - 1] + nums1[i - 1] * nums2[j - 1] — the maximum dot product including the current elements from both nums1 and nums2. 4. If dp[i - 1][j - 1] is less than 0, we only consider the product nums1[i - 1] * nums2[j - 1] because we would not want to diminish our

```
result by adding a negative dot product from previous elements.
```

By computing the maximum of these options at each entry of the dp table, we ensure that we have considered all possibilities and end up with the maximum dot product of subsequences of the same length from nums1 and nums2.

The solution to this problem involves using a 2D dynamic programming approach, which is a common pattern when dealing with problems of sequences and subproblems that depend on previous decisions.

We start by creating a 2D array dp with dimensions $(m + 1) \times (n + 1)$, where m is the length of nums1 and n is the length of nums2.

We use m + 1 and n + 1 because we want to have an extra row and column to handle the base cases where the subsequence

nums2.

class Solution:

return dp[-1][-1]

Solution Approach

length is zero from either of the arrays. We initialize all values in the dp array to negative infinity (-inf) to represent the minimum possible dot product.

The <u>dynamic programming</u> algorithm iteratively fills the dp array as follows: 1. We loop over each possible subsequence length for nums1 (denoted by i) and nums2 (denoted by j) starting from 1 because index 0 is the base case representing an empty subsequence. 2. For each i, j pair, we calculate the dot product of the last elements v by multiplying nums1[i - 1] * nums2[j - 1].

 dp[i][j - 1]: The max dot product without including nums2[j - 1]; o max(dp[i - 1][j - 1], 0) + v: The max dot product including the current elements of both arrays. We use max(dp[i - 1][j - 1], 0)

def maxDotProduct(self, nums1: List[int], nums2: List[int]) -> int:

o dp[i − 1][j]: The max dot product without including nums1[i − 1];

3. We then fill in dp[i][j] by taking the maximum of:

The reason we initialize with negative infinity and consider the $\max(dp[i-1][j-1], 0)$ in our recurrence is to ensure that we do not forcefully include negative products that would decrease the total maximum dot product. However, we need to include at

```
least one pair of elements from both subsequences, as the result cannot be an empty subsequence according to the problem
statement.
```

After filling the dp array, the last cell dp[m][n] contains the maximum dot product of non-empty subsequences of nums1 and

because if the previous dot product is negative, we would get a better result by just taking the current product v.

m, n = len(nums1), len(nums2) $dp = [[-inf] * (n + 1) for _ in range(m + 1)]$ for i in range(1, m + 1): for j in range(1, n + 1): v = nums1[i - 1] * nums2[j - 1]dp[i][j] = max(dp[i - 1][j], dp[i][j - 1], max(dp[i - 1][j - 1], 0) + v)

This code correctly solves the problem by leveraging the power of dynamic programming to optimize the process of finding the

maximum dot product.

Here is the code snippet again that reflects this approach:

```
Example Walkthrough
  Let's consider two small arrays for simplicity:
 • nums1 = [2, 3]
 • nums2 = [1, 2]
```

(since nums1 and nums2 both have lengths 2, and we include an extra row and column for the base case):

The entries dp[i - 1][j], dp[i][j - 1], and dp[i - 1][j - 1] are all 0 as they refer to base cases.

○ We consider the maximum of dp[1][1], dp[1][0], and dp[0][1] + 4 which are 2, 0, and 4, respectively.

○ We take the maximum of dp[2][0], dp[1][1], and 0 + 3 (since dp[i - 1][j - 1] is 0), which are 0, 2, and 3.

dp array initialization (values are -inf except dp[0][*] and dp[*][0] which could be set as 0 for base case): 0, (-inf), (-inf)] [(-inf), (-inf), (-inf)] [(-inf), (-inf), (-inf)]

Using the dynamic programming approach described in the problem solution, we will create a 2D dp array with dimensions 3×3

```
○ We select the maximum: max(0, 0, 0 + 2) which is 2.
Now, dp[1] [1] is updated to 2.
At i = 1, j = 2:

    nums1[i - 1] * nums2[j - 1] is 2 * 2 which equals 4.
```

```
• The maximum value is 3, so dp[2][1] is updated to 3.
At i = 2, j = 2:
```

We start populating the dp array at dp[1][1]:

nums1[i - 1] * nums2[j - 1] is 2 * 1 which equals 2.

• The maximum value is 4, so we update dp[1][2] to 4.

 \circ nums1[i - 1] * nums2[j - 1] is 3 * 1 which equals 3.

 \circ nums1[i - 1] * nums2[j - 1] is 3 * 2 which equals 6.

After finishing the iteration, the final dp array looks like this:

At i = 1, j = 1:

At i = 2, j = 1:

0, 0] 2, 4] 3, 8]

Solution Implementation

from typing import List

return dp[-1][-1]

class Solution:

Python

```
    Now, we consider max(dp[2][1], dp[1][2], dp[1][1] + 6) which are 3, 4, and 8.

• The maximum value is 8, so we update dp[2][2] to 8.
```

```
• The value dp[2][2] which is 8 represents the maximum dot product of non-empty subsequences of nums1 and nums2.
Thus, the maximum dot product obtained from the subsequences [2, 3] from nums1 and [1, 2] from nums2 is 8, which is the dot
```

this would indeed be the maximum dot product in this simple example.

def max_dot_product(self, nums1: List[int], nums2: List[int]) -> int:

Calculate the dot product of the current elements

3. The previous row and column plus the current dot_product,

Return the last element of the DP array which contains the maximum dot product

dpTable[i][j] = Math.max(dpTable[i - 1][j], dpTable[i][j - 1]);

// Return the result from the DP table which contains the maximum dot product

ensuring that if the previous value is negative, zero is used instead

 $dp[i][j] = max(dp[i - 1][j], dp[i][j - 1], max(dp[i - 1][j - 1], 0) + dot_product)$

 $dot_product = nums1[i - 1] * nums2[j - 1]$

1. The previous row at the same column

2. The same row at the previous column

Update the DP table by considering:

// Iterate over the arrays to populate the DP table

for (int i = 1; i <= length1; ++i) {</pre>

return dpTable[length1][length2];

return dp[numRows][numCols];

for i in range(1, len_nums1 + 1):

return dp[-1][-1]

for j in range(1, len_nums2 + 1):

Calculate the dot product of the current elements

3. The previous row and column plus the current dot_product,

Return the last element of the DP array which contains the maximum dot product

ensuring that if the previous value is negative, zero is used instead

 $dp[i][j] = max(dp[i - 1][j], dp[i][j - 1], max(dp[i - 1][j - 1], 0) + dot_product)$

dot_product = nums1[i - 1] * nums2[j - 1]

1. The previous row at the same column

2. The same row at the previous column

Update the DP table by considering:

function maxDotProduct(nums1: number[], nums2: number[]): number {

};

TypeScript

for (int j = 1; j <= length2; ++j) {</pre>

Initialize the lengths of the two input lists

len_nums1, len_nums2 = len(nums1), len(nums2)

for j in range(1, len_nums2 + 1):

Initialize a 2D DP array filled with negative infinity dp = [[float('-inf')] * (len_nums2 + 1) for _ in range(len_nums1 + 1)] # Build the DP table row by row, column by column for i in range(1, len_nums1 + 1):

product of the two arrays. Since these arrays are both the full length of the originals and we are looking for any subsequences,

```
Java
class Solution {
   // Method to calculate the maximum dot product between two arrays
   public int maxDotProduct(int[] nums1, int[] nums2) {
       // Lengths of the input arrays
       int length1 = nums1.length, length2 = nums2.length;
       // Create a DP table with an extra row and column for the base case
       int[][] dpTable = new int[length1 + 1][length2 + 1];
       // Initialize the DP table with the minimum integer values
        for (int[] row : dpTable) {
           Arrays.fill(row, Integer.MIN_VALUE);
```

// Taking the maximum between the current value, ignoring current elements of nums1 or nums2

// Determine the maximum value by considering the current elements and previous subsequence's result

dpTable[i][j] = Math.max(dpTable[i][j], Math.max(0, dpTable[i - 1][j - 1]) + nums1[i - 1] * nums2[j - 1]);

```
C++
#include <vector>
#include <algorithm>
#include <climits>
class Solution {
public:
    // Function to calculate the maximum dot product between two sequences.
    int maxDotProduct(vector<int>& nums1, vector<int>& nums2) {
        int numRows = nums1.size(); // Size of the first sequence.
        int numCols = nums2.size(); // Size of the second sequence.
       // Create a 2D DP (Dynamic Programming) table with all elements initialized to INT_MIN.
        vector<vector<int>> dp(numRows + 1, vector<int>(numCols + 1, INT_MIN));
        // Building the DP table by considering each possible pair of elements from nums1 and nums2.
        for (int i = 1; i <= numRows; ++i) {</pre>
            for (int j = 1; j <= numCols; ++j) {</pre>
                // Current dot product value.
                int currentDotProduct = nums1[i - 1] * nums2[j - 1];
                // Choosing the maximum between not taking the current pair, or taking the current pair.
                // First, consider the maximum value from ignoring the current pair (up or left in DP table).
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                // Then, consider the maximum value from taking the current pair, which is the current dot product
                // plus the maximum dot product without both elements (up-left diagonally in DP table), unless negative,
                // in which case use zero (as dot products with negative results don't contribute to the maximum).
                dp[i][j] = max(dp[i][j], max(0, dp[i - 1][j - 1]) + currentDotProduct);
```

```
const numRows = nums1.length; // Size of the first sequence
const numCols = nums2.length; // Size of the second sequence
// Create a 2D DP (Dynamic Programming) table initialized to negative infinity
const dp: number[][] = Array.from({ length: numRows + 1 }, () =>
                                  Array(numCols + 1).fill(-Infinity));
// Building the DP table by considering each possible pair of elements from nums1 and nums2
for (let i = 1; i <= numRows; i++) {</pre>
    for (let j = 1; j <= numCols; j++) {</pre>
       // Current dot product value
        const currentDotProduct = nums1[i - 1] * nums2[j - 1];
        // Choose the maximum between not taking the current pair, or taking the current pair
       // First, consider the maximum value from ignoring the current pair (above or to the left in DP table)
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
        // Then, consider the maximum value from taking the current pair, which includes the current dot product
        // plus the maximum dot product without both elements (diagonally above to the left in DP table); if this value
        // is negative, use zero instead, as dot products with negative results don't contribute to the maximum
```

// The maximum dot product for the sequences will be in the bottom—right corner of the DP table.

```
// The maximum dot product for the sequences will be in the bottom-right corner of the DP table
      return dp[numRows][numCols];
from typing import List
class Solution:
   def max_dot_product(self, nums1: List[int], nums2: List[int]) -> int:
       # Initialize the lengths of the two input lists
        len_nums1, len_nums2 = len(nums1), len(nums2)
       # Initialize a 2D DP array filled with negative infinity
        dp = [[float('-inf')] * (len_nums2 + 1) for _ in range(len_nums1 + 1)]
       # Build the DP table row by row, column by column
```

dp[i][j] = Math.max(dp[i][j], Math.max(0, dp[i - 1][j - 1]) + currentDotProduct);

Time and Space Complexity The time complexity of the given code is 0(m * n), where m is the length of nums1 and n is the length of nums2. This is because

```
there are two nested loops, each iterating through the elements of nums1 and nums2 respectively.
The space complexity of the code is also 0(m * n). This is due to the allocation of a 2D array dp of size (m + 1) * (n + 1) to
store the intermediate results for each pair of indices (i, j).
```