Intuition

For this problem, There is nothing to try than trying all possible **combinations** of subsequences from the two input arrays.
I believe that will give as the **maximum** dot product but we can see that it is **inefficient**. 😔
How can we **optimize** it

Let's see next example together.

EX: nums1 = [2,1,-2,5], nums2 = [3,0,-6]

Let's try different subsequences:

[2, -2] & [3, -6] -> 18

[2, -2] & [0, -6] -> 12

[1, -2] & [3, -6] -> 15

[1, -2] & [0, -6] -> 13

Answer = 18

In all **previous** subsequences, we notice something **shared** between them. 🤯
Yes, The second half of all subsequences is **repeated** which mean there are repeated subproblems we can make use of. 🤔

We are here todya with our hero **Dynamic Programming**. 🦸
I think you are surprised like me 😂

For the **DP** solution, we will try all **combinations** of subsequences but we won't **calculate** all the combinations' results.
What does that mean? 🤔
Like we say in the previous example we have part of the subsequences is **repeated** so we will **cache** the optimal result for that part and whenever we encounter the same subproblem of that part again we will only return the optimal result. 🤪

EX: nums1 = [2,1,-2,5,-1,4], nums2 = [3,0,-6,5,2]

Let's try different subsequences:

[2, -2, 4] & [3, -6, 5] -> 38 = 18 + 20

[2, -2, 4] & [0, -6, 5] -> 32 = 12 + 20

[1, -2, 4] & [3, -6, 5] -> 35 = 15 + 20

[1, -2, 4] & [0, -6, 5] -> 33 = 13 + 20

Answer = 38

We can see here that we added new two numbers (new subarray) in each array from the previous example.
BUT we can notice that those subarrays have always the **best** answer which is 20 and won't change so we will **cache** it and whenever we want the best solution from them we will return simply 20.

- For the **DP** solution, we will start with **two pointers** one for each array and for the pointers we have **three** options:
    - **multiply** the numbers that the pointers point at then move the both pointers to next element indicating that we are **taking** them in the final subsequences
    - **move first** pointer to the next element indicating that we are **deleting** it from the final subsequences
    - **move second** pointer to the next element indicating that we are **deleting** it from the final subsequences

Then, there is only **one edge case** that we must handle its when one of the arrays is all **negative** numbers and the other is all **positive** numbers.
our fuction will return 0 as an answer since it won't take any number of the **negative** array which means **empty** subsequence which is **invalid**. 😔

EX: nums1 = [-2, -3, -1, -5], nums2 = [4, 2, 1, 5]

Answer -> -1 * 1 = -1

The answer is to get **maximum** element in **positive** array and **minimum** element in **negative** array then multiply them.

But what about both of the arrays are both negative numbers? 🤔
simply the dot product of any subsequence will be **positive** and we will have answer eventually unlike the previous case.

APPROACH

DP Recursive (Top-Down)

1. Initialize a dynamic programming array `dp` of size `505x505` and some variables for the **size** and **maximums** and **minimums**.
2. Find the **maximum** and **minimum** values in both arrays to handle **special** case where all elements are negative in any of the arryas.
3. Call the recursive function `calculateDotProduct` with initial indices `0` and `0` to compute the **maximum** dot product:
    - **Base Cases:** returns `0`, as the end of an array has been reached.
    - **Memoization:** Checks if the result for the current indices is **already** calculated and stored in `dp`.
    - **Calculates** the dot product with three options:
        - Multiply the elements at the current indices in `nums1` and `nums2`, and recursively calculate the dot product for the next indices (`idx1 + 1`, `idx2 + 1`).
    - Calculate the dot product by moving to the next index in `nums2` while keeping the index in `nums1` unchanged.
    - Calculate the dot product by moving to the next index in `nums1` while keeping the index in `nums2` unchanged.
    - **Stores** the **maximum** dot product among the three options in `dp` for the current indices (`idx1`, `idx2`) then return.
4. Return the computed **maximum dot product** taht you got from the function.

Complexity

- **Time complexity:** $O(N*M)$
  Since we are trying all possible options in Dynamic Programming, The complexity is the product of the sizes of the arrays which we are trying all possible combinations of then which is `O(N * M)`.
- **Space complexity:** $O(N*M)$
  Since we are storing the DP matrix and its size is `N * M` then complexity is `O(M * N)`. where `N` is maximum size of first array and `M` is maximum size of second array.