1874. Minimize Product Sum of Two Arrays

Medium Greedy Array Sorting

Problem Description

The problem is asking for the minimum possible product sum of two equal-length arrays. The product sum is the sum of `a[i] * b[i]` for all possible `i`, given 0-indexed arrays `a` and `b`. The twist in the problem is that while one of the arrays (`nums2`) must remain in its original order, the other array (`nums1`) can be rearranged in any order to minimize the product sum.

To provide an example, let's assume we have `a = [1,2,3,4]` and `b = [5,2,3,1]`. If we were not allowed to rearrange `a`, then the product sum would be `1*5 + 2*2 + 3*3 + 4*1 = 22`. However, if we can rearrange `a`, we would try to pair the largest numbers in `b` with the smallest numbers in `a` to get the smallest possible product for each pair, thereby minimizing the total product sum.

Intuition

To find the minimum product sum, we should try to multiply the smallest numbers in `nums1` with the largest numbers in `nums2`, and vice versa. This is because a large number times a small number adds less to the product sum than a large number times another large number.

To implement this, both arrays `nums1` and `nums2` are sorted. For `nums1`, we sort it in non-decreasing order. This places the smallest elements at the start of the array. We do not need to sort `nums2` as its order cannot be changed, but for the purpose of the solution, we sort it in non-decreasing order as well.

We then iterate over the arrays, multiplying the `i`-th element of `nums1` with the `(n - i - 1)`-th element of `nums2` which corresponds to the `i`-th largest element in `nums2` because `nums2` is sorted in ascending order. The sum of these products gives the minimum product sum possible by this rearrangement.

This method works because sorting `nums1` and pairing its elements with the reverse-sorted elements of `nums2` ensures that each product added to the sum is as small as it can possibly be considering the constraints of the problem.

Solution Approach

The solution involves a few steps as described below:

1. Sort the `nums1` array in non-decreasing order, which will place the smallest elements at the beginning.
2. Sort the `nums2` array in non-decreasing order as well, though its original order does not need to be changed for the problem statement. This step is purely for ease of implementation to allow us to iterate from one end of `nums1` to the other while moving in the opposite direction in `nums2`.

By completing these sorting steps, we can now pair the smallest elements in `nums1` with the largest in `nums2`, which is the key to minimizing the product sum.

3. Initialize a variable `res` to store the cumulative product sum.
4. Iterate over the length of either `nums1` or `nums2` (since they are of equal length) using a for loop. Calculate the product by taking the `i`-th element from the sorted `nums1` and the element at index `(n - i - 1)` from the sorted `nums2` array. This effectively reverses the second array during the multiplication process.
5. Add this product to the variable `res` to keep a running total.
6. After the loop concludes, `res` contains the minimum product sum, as per the requirements of the problem statement.

The algorithm used is sorting, which under the hood, depending on the language and its implementation, can be a quicksort, mergesort, or similar `O(n log n)` sorting algorithm. The rest of the function simply iterates through the arrays, which is an `O(n)` operation.

Here is the code breakdown:

- `nums1.sort()`: Sorts the first list in non-decreasing order.
- `nums2.sort()`: Sorts the second list in non-decreasing order.
- `n`: Captures the length of the arrays.
- `res`: The variable that will accumulate the total product sum.
- `for i in range(n)`: Iterates through each index of the arrays.
- `res += nums1[i] * nums2[n - i - 1]`: Calculates the product of the smallest element in `nums1` with the largest remaining element in `nums2` and adds it to `res`.

After going through all elements, we simply return `res` which now holds the minimum product sum.

```
class Solution:
    def minProductSum(self, nums1: List[int], nums2: List[int]) -> int:
        nums1.sort()
        nums2.sort()
        n, res = len(nums1), 0
        for i in range(n):
            res += nums1[i] * nums2[n - i - 1]
        return res
```

This is a classic greedy approach that ensures the products of pairs are as small as possible to achieve the global minimum sum.

Example Walkthrough

Let's consider two sample arrays `nums1 = [4, 3, 2, 1]` and `nums2 = [1, 2, 3, 4]`. Our goal is to minimize the product sum of these two arrays, with the possibility of rearranging `nums1` but keeping the order of `nums2` fixed.

Following the solution approach:

1. Sort `nums1` in non-decreasing order to get `[1, 2, 3, 4]`.
2. Sort `nums2` in non-decreasing order for the purpose of calculation to get `[1, 2, 3, 4]`. Remember that the order of `nums2` isn't actually changed in the final answer—this step is simply to help visualize the process.
3. Initialize `res` to 0 which will hold the cumulative product sum.
4. Iterate over the arrays, calculating the product sum using the greedy approach:

   For `i = 0` (first pairing):

   - The product of `nums1[0]` and `nums2[4 - 0 - 1]` (the last element of `nums2`) is `1 * 4 = 4`.

   For `i = 1` (second pairing):

   - The product of `nums1[1]` and `nums2[4 - 1 - 1]` (the second last element of `nums2`) is `2 * 3 = 6`.

   For `i = 2` (third pairing):

   - The product of `nums1[2]` and `nums2[4 - 2 - 1]` (the third last element of `nums2`) is `3 * 2 = 6`.

   For `i = 3` (fourth pairing):

   - The product of `nums1[3]` and `nums2[4 - 3 - 1]` (the first element of `nums2`) is `4 * 1 = 4`.
5. The cumulative sum of these products is `4 + 6 + 6 + 4 = 20`, which is stored in `res`.
6. After completing the iteration, `res` now contains the minimum product sum which, in this case, is `20`.

This demonstrates the solution approach effectively: by sorting `nums1` and pairing each element with the 'opposite' element from `nums2` (i.e., the element as far from it as possible in the sorted version of `nums2`), we achieve the minimum product sum. The implemented Python code will return `20` as the result for this example.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minProductSum(self, nums1: List[int], nums2: List[int]) -> int:
        # Sort the first list in non-decreasing order
        nums1.sort()
        # Sort the second list in non-decreasing order
        nums2.sort()

        # Initialize length of the list for iteration and result variable to store the sum
        length, result = len(nums1), 0

        # Iterate over the lists
        for i in range(length):
            # Multiply the i-th smallest element in nums1 with the i-th largest in nums2
            # and add it to the result. This maximizes the product sum of the min/max pairs.
            result += nums1[i] * nums2[length - i - 1]

        # Return the final product sum
        return result
```

Java

```java
class Solution {
    // Method to calculate the minimum product sum of two arrays
    public int minProductSum(int[] nums1, int[] nums2) {
        // Sort both arrays
        Arrays.sort(nums1);
        Arrays.sort(nums2);

        int n = nums1.length; // Number of elements in the array
        int result = 0;       // Variable to store the result of the minimum product sum

        // Iterate through the arrays to calculate the product sum
        // Multiply elements in a way that smallest of one array is paired with largest of the other
        for (int i = 0; i < n; i++) {
            result += nums1[i] * nums2[n - i - 1]; // Add to result by pairing elements
        }

        // Return the calculated minimum product sum
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    int minProductSum(vector<int>& nums1, vector<int>& nums2) {
        // Sort both vectors in non-decreasing order.
        sort(nums1.begin(), nums1.end());
        sort(nums2.begin(), nums2.end());

        int size = nums1.size(); // Store the size of the vectors.
        int result = 0; // Initialize result to accumulate the product sum.

        // Loop through every element of nums1.
        for (int i = 0; i < size; ++i) {
            // Multiply the current element in nums1 by the corresponding element from the end of nums2.
            // This ensures the smallest number in nums1 is multiplied with the largest in nums2 and so on.
            result += nums1[i] * nums2[size - i - 1];
        }

        // Return the final min product sum.
        return result;
    }
};
```

TypeScript

```typescript
function minProductSum(nums1: number[], nums2: number[]): number {
    // Sort both arrays in non-decreasing order.
    nums1.sort((a, b) => a - b);
    nums2.sort((a, b) => a - b);

    let size = nums1.length; // Store the length of the arrays.
    let result = 0; // Initialize result to accumulate the product sum.

    // Loop through every element of nums1
    for (let i = 0; i < size; ++i) {
        // Multiply the current element in nums1 by the corresponding element from the end of nums2
        // This ensures the smallest number in nums1 is multiplied with the largest in nums2, and so on
        result += nums1[i] * nums2[size - i - 1];
    }

    // Return the final min product sum
    return result;
}
```


```python
from typing import List

class Solution:
    def minProductSum(self, nums1: List[int], nums2: List[int]) -> int:
        # Sort the first list in non-decreasing order
        nums1.sort()
        # Sort the second list in non-decreasing order
        nums2.sort()

        # Initialize length of the list for iteration and result variable to store the sum
        length, result = len(nums1), 0

        # Iterate over the lists
        for i in range(length):
            # Multiply the i-th smallest element in nums1 with the i-th largest in nums2
            # and add it to the result. This maximizes the product sum of the min/max pairs.
            result += nums1[i] * nums2[length - i - 1]

        # Return the final product sum
        return result
```


Time and Space Complexity

Time Complexity

The primary operations in the code are the sorting of `nums1` and `nums2`, followed by a single pass through the arrays to calculate the product sum.

- Sorting: The `sort()` function in Python typically uses the Timsort algorithm, which has a time complexity of `O(n log n)`. As there are two lists being sorted, this operation is performed twice.
- Single Pass Calculation: After sorting, the code iterates through the lists once, with a single loop performing `n` iterations, where `n` is the length of `nums1` (and `nums2`). The operations inside the loop are constant time, so this is `O(n)`.

The overall time complexity is the sum of the two operations, but since `O(n log n)` dominates `O(n)`, the total time complexity of the algorithm is `O(n log n)`.

Space Complexity

The space complexity of the code is determined by the additional space required apart from the input:

- No extra space is used for storing intermediate results; only a fixed number of single-value variables (`n`, `res`) are utilized.
- The sorting algorithms may require `O(n)` space in the worst case for manipulating the data.

Therefore, the space complexity is `O(n)`.