1913. Maximum Product Difference Between Two Pairs

Easy | Array | Sorting

Leetcode Link

Problem Description

The problem presents us with the concept of the product difference between two pairs of numbers `(a, b)` and `(c, d)`, which is calculated as `(a * b) - (c * d)`. The main objective is to find four distinct elements in an integer array `nums`, such that when you form two pairs with these elements and calculate their product difference, the result is as large as possible.

Consider an integer array `nums`. The task is to choose four distinct indices `w`, `x`, `y`, and `z` from this array such that the product difference calculated with elements at these indices `(nums[w] * nums[x]) - (nums[y] * nums[z])` is the maximum possible.

For example, if you have the numbers `{1,2,3,4}`, the maximum product difference would be calculated by using the highest two numbers for multiplication to form the first pair, and the lowest two numbers to form the second pair. The maximum product difference in this case would be `(4*3) - (1*2) = 10`.

Intuition

To maximize the product difference, we need the first product `(a * b)` to be as large as possible and the second product `(c * d)` to be as small as possible. A natural strategy to achieve this is to sort the array first.

When an array is sorted in ascending order:

- The two largest values, which will be at the end of the array, can be used to form the maximum product pair `(a * b)`.
- Conversely, the two smallest values, which will be at the beginning of the array, will form the minimum product pair `(c * d)`.

By selecting these pairs, the product difference `(a * b) - (c * d)` will be maximized. This is because multiplication is a commutative operation (meaning `a * b` is the same as `b * a`), and sorting the array gives us direct access to the largest and smallest values without additional comparisons.

Accordingly, by sorting `nums`, and then calculating the product of the last two elements and subtracting the product of the first two elements, we reach the solution.

The solution approach is direct and efficient, aligning with our intuition of maximizing the first product and minimizing the second.

Solution Approach

The method of solving this problem involves just a few straightforward steps but is grounded in understanding how sorting affects the arrangement of numbers and how it can be leveraged to find the maximum product difference. The approach follows these distinct steps:

1. **Sorting**: The input list `nums` is sorted. This is done with the `nums.sort()` method. Sorting rearranges the elements in the list from the smallest to the largest. In Python, this operation has an average and worst-case time complexity of `O(n log n)`, where `n` is the number of elements in the list.

2. **Selecting Elements**: After sorting, the two largest numbers will be placed at the end of the list. In Python, negative indices can be used to access elements from the end of a list. Thus `nums[-1]` and `nums[-2]` give us the largest and second-largest numbers, respectively.

3. **Calculating the Product of the Largest Pair**: The largest and second-largest numbers are then multiplied together to form the maximum possible product from the list. This is the first part of the product difference calculation, denoted as `(a * b)`.

4. **Calculating the Product of the Smallest Pair**: Similarly, the smallest numbers will now be at the very beginning of the list. By directly accessing `nums[0]` and `nums[1]`, the first two elements, we get the smallest and second-smallest numbers, which are multiplied together. This product forms the second part of the product difference, denoted as `(c * d)`.

5. **Calculating the Product Difference**: Finally, we calculate the product difference by subtracting the product of the smallest pair from the product of the largest pair, i.e., `(a * b) - (c * d)`.

6. **Return the Result**: The last step is simply to return the result of the product difference calculation.

The solution utilizes the built-in sorting algorithm and simple list indexing, which make the implementation quite efficient. The only data structure used is the list itself, and no additional space is required. The approach takes advantage of the sorted order of elements to quickly and directly access the values needed to maximize the product difference.

Here is the code snippet that encapsulates our solution approach:

```python
class Solution:
    def maxProductDifference(self, nums: List[int]) -> int:
        nums.sort()
        return nums[-1] * nums[-2] - nums[0] * nums[1]
```

This solution strategy provides a quick and elegant way to achieve the desired result, and it's an excellent example of how sorting can simplify certain types of problems.

Example Walkthrough

To help illustrate the solution approach, let's go through a small example using the array `nums = [4, 2, 5, 9, 7, 1]`.

Following the steps outlined in the solution approach:

1. **Sorting**: First, we sort the array `nums` to obtain `[1, 2, 4, 5, 7, 9]`. After sorting, the smallest numbers are at the start and the largest at the end.

2. **Selecting Elements**: The two largest numbers, which are now at the end of the array, are `9` and `7`. The two smallest numbers at the beginning of the array are `1` and `2`.

3. **Calculating the Product of the Largest Pair**: We multiply the largest two numbers to get the maximum product `(a * b) = 9 * 7 = 63`.

4. **Calculating the Product of the Smallest Pair**: Next, we multiply the smallest two numbers to get the minimum product `(c * d) = 1 * 2 = 2`.

5. **Calculating the Product Difference**: The product difference `(a * b) - (c * d)` is thus `63 - 2 = 61`.

6. **Return the Result**: We return `61` as the maximum product difference we can get from the array by choosing four distinct elements to form two pairs.

By following these steps, we have maximized the product difference using the numbers `9`, `7` (the largest pair), and `1`, `2` (the smallest pair) from the given array `nums`. This approach can be applied to any input array to efficiently find the maximum product difference.

Python Solution

```python
# We are using the typing module for type hints
from typing import List

class Solution:
    def maxProductDifference(self, nums: List[int]) -> int:
        # First, we sort the list of numbers in ascending order.
        nums.sort()

        # The largest elements will be at the end of the sorted list, and
        # the smallest elements will be at the beginning.
        # Therefore, the product of the two largest elements can be found
        # by multiplying the last two elements in the sorted list.
        largest_product = nums[-1] * nums[-2]

        # Similarly, the product of the two smallest elements can be found
        # by multiplying the first two elements in the sorted list.
        smallest_product = nums[0] * nums[1]

        # The maximum product difference is the difference between the
        # largest product and the smallest product.
        max_product_difference = largest_product - smallest_product

        # Return the computed maximum product difference.
        return max_product_difference

```

Java Solution

```java
import java.util.Arrays; // Import Arrays class to use the sort method

public class Solution {
    // Method to find the maximum product difference between two pairs
    // The two pairs consist of the product of the two largest numbers
    // and the product of the two smallest numbers in the given array.
    public int maxProductDifference(int[] nums) {
        // Sort the array to easily find the smallest and largest elements
        Arrays.sort(nums);

        int length = nums.length; // Store the length of the array

        // Calculate the product difference:
        // maxProduct is the product of the two largest numbers
        int maxProduct = nums[length - 1] * nums[length - 2];

        // minProduct is the product of the two smallest numbers
        int minProduct = nums[0] * nums[1];

        // Return the difference between maxProduct and minProduct
        return maxProduct - minProduct;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for std::sort

class Solution {
public:
    // Function to calculate the maximum product difference between two pairs (a, b) and (c, d)
    // where a and b are elements from nums, and c and d are elements from nums.
    int maxProductDifference(std::vector<int>& nums) {
        // Sort the input vector in non-decreasing order.
        std::sort(nums.begin(), nums.end());

        // Get the size of the nums array.
        int size = nums.size();

        // The maximum product is the product of the last two elements in the sorted array,
        // and the minimum product is the product of the first two elements.
        // Calculate the product difference by subtracting the minimum product from the maximum product.
        int productDifference = nums[size - 1] * nums[size - 2] - nums[0] * nums[1];

        // Return the calculated product difference.
        return productDifference;
    }
};

```

Typescript Solution

```typescript
/**
 * Function to calculate the maximum product difference between two pairs of numbers
 * in an array. The product difference is defined as the product of the two largest
 * numbers minus the product of the two smallest numbers.
 *
 * @param {number[]} nums - The array of numbers
 * @return {number} - The maximum product difference
 */
function maxProductDifference(nums: number[]): number {
    // Sort the array in non-decreasing order
    nums.sort((a, b) => a - b);

    // Get the length of the array
    const length = nums.length;

    // Calculate the product of the two largest numbers
    const productOfLargest = nums[length - 1] * nums[length - 2];

    // Calculate the product of the two smallest numbers
    const productOfSmallest = nums[0] * nums[1];

    // Calculate the difference between the two products
    const answer = productOfLargest - productOfSmallest;

    // Return the maximum product difference
    return answer;
}

// Example usage:
// const result = maxProductDifference([5, 6, 2, 7, 4]);
// console.log(result); // Output will be the maximum product difference

```

Time and Space Complexity

The time complexity of the given code is `O(n log n)`, where `n` is the number of elements in the list `nums`. This time complexity arises from the sorting operation `nums.sort()`. Sorting is the most computationally intensive part of this code.

Once the list is sorted, accessing the four elements needed to compute the product difference (the first two elements and the last two elements) is done in constant time, `O(1)`.

The space complexity of the code is `O(1)` if we consider the sort to be in-place, which it is for Python's default sorting algorithm (Timsort). No additional space is proportional to the input size is utilized besides potential negligible temporary variables for holding products and the result of the subtraction.