1770. Maximum Score from Performing Multiplication Operations

Hard · Array · Dynamic Programming — Leetcode Link

Problem Description

You are given two arrays `nums` and `multipliers`. Array `nums` is of size `n`, and `multipliers` is of size `m`, with the condition that `n >= m`. You start with a score of `0` and intend to perform exactly `m` operations to maximize your score.

Each operation consists of the following steps:

1. Select an integer `x` from either the start or end of the array `nums`.
2. Multiply `x` by the corresponding `multipliers[i]` (where `i` is the index of the current operation, starting at `0`) and add the result to your score.
3. Remove `x` from `nums`.

The challenge is to figure out the sequence of operations that leads to the maximum possible score after performing `m` operations, and return this maximum score.

Intuition

To solve this problem, we use dynamic programming to keep track of the highest score we can achieve after a certain number of operations. Specifically, we create a 2D array `f` where `f[i][j]` represents the maximum score we can achieve by selecting `i` elements from the start and `j` elements from the end of `nums`. The value of `k` in the code represents the current operation we're on, and `ans` will store our final answer.

Here's the intuition behind the solution step by step:

1. We prepare a 2D DP array `f` with dimensions `(m + 1) x (m + 1)` where each cell is initially filled with negative infinity (`-inf`) because we want to calculate the maximum. The `+1` ensures we have space for zero selections from both ends.
2. Set `f[0][0]` to `0` as the base case, which represents that no operation has been performed yet and the score is zero.
3. For each possible set of selections from the start and end (represented by `i` and `j` respectively), calculate the score for that state and store it in `f[i][j]`.
4. For the state represented by `f[i][j]`, we consider two possibilities:
   - `f[i - 1][j] + multipliers[k] * nums[i - 1]`: This corresponds to selecting the element `x` from the start of the array.
   - `f[i][j - 1] + multipliers[k] * nums[n - j]`: This corresponds to selecting the element `x` from the end of the array.
5. Take the maximum of these two values to find the optimal score for that particular operation.
6. If the sum of `i` and `j` is equal to `m` (meaning all operations have been used), update `ans` to track the highest score found up to that point.
7. After evaluating all possible combinations of selections from `nums`, return the maximum score `ans`.

This approach ensures that at every step, we are considering all possible choices and computing the optimal score that can be achieved given the previous decisions, thereby solving the problem optimally.

Solution Approach

The solution approach uses dynamic programming, a method for solving a complex problem by breaking it down into simpler subproblems. It is applicable here because the decision at each step depends on the results of previous steps, and there are overlapping subproblems.

The implementation details are as follows:

- **Initialization**: Create a 2D list `f` of size `(m + 1) x (m + 1)`, where `m` is the length of the `multipliers` array, and set all elements to `-inf`. This represents the maximum possible scores. Additionally, set `f[0][0]` to `0` as a starting point since no operations mean no score.
- **Nested Loops**: Two nested loops iterate over all possibilities where `i` represents the number of elements selected from the start, and `j` represents the number of elements selected from the end. Since the problem requires exactly `m` operations, the outer loop ranges from `0` to `m`, inclusive, ensuring `i + j <= m`.
- **State Update**: For each `(i, j)` pair:
  - Calculate the index `k = i + j - 1`, which represents the operation number.
  - Update `f[i][j]` according to the dynamic programming state transition equation:
    - If `i > 0`, meaning we can select from the start, update `f[i][j]` with the maximum of its current value or the value from choosing the start (`f[i - 1][j] + multipliers[k] * nums[i - 1]`).
    - If `j > 0`, meaning we can select from the end, update `f[i][j]` with the maximum of its current value or the value from choosing the end (`f[i][j - 1] + multipliers[k] * nums[n - j]`).
  - Check if `i + j` equals `m`, that is, if we have performed `m` operations. If so, update `ans` with the maximum of its current value and `f[i][j]`.
- **Answer Calculation**: After populating the `f` table with all possible scores, `ans`, which has been tracking the maximum, will contain the maximum score possible. Hence, return `ans`.

By storing and updating the maximum score at each state, the solution efficiently computes the maximum score possible after `m` operations. This dynamic programming table eliminates the need to recompute overlapping subproblems and ensures each subproblem is solved only once. The final answer is found by considering all possible ways to perform the `m` operations.

Example Walkthrough

Let's illustrate the dynamic programming solution approach with a small example.

Suppose we have:

```
nums = [1, 2, 3]
multipliers = [3, 2]
```

Here, `n = 3` (size of `nums`) and `m = 2` (size of `multipliers`). As per the problem, we have to perform `m` operations, so in this case, we need to perform `2` operations.

Following the solution approach:

1. **Initialization**: We create a 2D list `f` with dimensions `(3 x 3)` (since `m + 1 = 2 + 1 = 3`), initializing all elements to `-inf`. Then, set `f[0][0]` to `0`. The table looks like this:

```
f = [
  [0, -inf, -inf],
  [-inf, -inf, -inf],
  [-inf, -inf, -inf]
]
```

2. **Nested Loops**: We iterate through all combinations of `i` (selections from the start) and `j` (selections from the end). Our loops will cover `(i, j)` pairs such as `(0, 0)`, `(1, 0)`, `(0, 1)`, and so on, but will not exceed `i + j = m`:

```
For i = 0 to 2
    For j = 0 to 2
        If i + j <= 2 // This is our operation limit `m`
            Perform state updates
```

3. **State Update**: We update `f[i][j]` for each `(i, j)` pair:
   - When `i = 1` and `j = 0`, `k = 0`:
     - We can take from the start: `f[1][0]` gets updated to `max(-inf, 0 + 3 * nums[0])` which is `3`.
   - When `i = 0` and `j = 1`, `k = 0`:
     - We can take from the end: `f[0][1]` gets updated to `max(-inf, 0 + 3 * nums[2])` which is `9`.

We continue doing this to fill out our table `f`. After the first set of operations, part of our table looks like this:

```
f = [
  [0, 9, -inf],
  [3, -inf, -inf],
  [-inf, -inf, -inf]
]
```

4. **Answer Calculation**: We check `f[i][j]` values where `i + j = m` to update `ans`, the maximum score so far. Since `m = 2`, we will check `f[2][0]`, `f[1][1]`, and `f[0][2]` to find our answer.

Continuing the update process with `i` and `j` values:

- When `i = 1` and `j = 1`, `k = 1`:
  - Take from the start: `f[1][1]` can become `max(-inf, f[0][1] + 2 * nums[0])` which is `9 + 2 * 1 = 11`.
  - Take from the end: `f[1][1]` can become `max(-inf, f[1][0] + 2 * nums[2])` which is `3 + 2 * 3 = 9`. Since `11` is greater, `f[1][1]` becomes `11`.

After updating all values, our table `f` looks like this:

```
f = [
  [0, 9, -inf],
  [3, 11, -inf],
  [-inf, -inf, -inf]
]
```

Our maximum score `ans` is the maximum value in `f` where `i + j = m`, which in this case is `f[1][1] = 11`.

Thus, the final answer is `11`; it's the maximum score we can get by performing `2` operations using the given `nums` and `multipliers`.

Python Solution

```python
from typing import List

class Solution:
    def maximumScore(self, nums: List[int], multipliers: List[int]) -> int:
        # The length of the nums and multipliers arrays
        num_length, multipliers_length = len(nums), len(multipliers)

        # Initialize the dp (Dynamic Programming) array with negative infinity
        # To accommodate for potential negative scores
        dp = [[float('-inf')] * (multipliers_length + 1) for _ in range(multipliers_length + 1)]

        # The starting score is 0 when no multipliers have been applied
        dp[0][0] = 0

        # Initialize the answer with negative infinity
        max_score = float('-inf')

        # Loop through all the possible combinations of left and right operations
        for left_count in range(multipliers_length + 1):
            for right_count in range(multipliers_length - left_count + 1):
                # The index of the current multiplier
                current_multiplier_index = left_count + right_count - 1

                # Calculate the score if we take the number from the left
                if left_count > 0:
                    dp[left_count][right_count] = max(dp[left_count][right_count],
                                                      dp[left_count - 1][right_count] + multipliers[current_multiplier_index] * num

                # Calculate the score if we take the number from the right
                if right_count > 0:
                    dp[left_count][right_count] = max(dp[left_count][right_count],
                                                      dp[left_count][right_count - 1] + multipliers[current_multiplier_index] * num

                # If we have used all multipliers, update the maximum score
                if left_count + right_count == multipliers_length:
                    max_score = max(max_score, dp[left_count][right_count])

        # Return the maximum score after using all multipliers
        return max_score

```

Java Solution

```java
import java.util.Arrays;

class Solution {

    // This method calculates the maximum score from performing operations on the array `nums` using the `multipliers`.
    public int maximumScore(int[] nums, int[] multipliers) {
        int n = nums.length; // The length of the nums array
        int m = multipliers.length; // The length of the multipliers array
        int[][] dp = new int[m + 1][m + 1]; // DP array to store the intermediate solutions

        // Initialize all values in the DP array to a very small number to ensure there's no overcount
        for (int i = 0; i <= m; i++) {
            Arrays.fill(dp[i], Integer.MIN_VALUE);
        }
        dp[0][0] = 0; // Base case: no operation gives score of 0

        int maxScore = Integer.MIN_VALUE; // Variable to keep track of the maximum score

        // Outer loop goes through all possible counts of operations
        for (int i = 0; i <= m; ++i) {
            // Inner loop considers different splits between left and right operations
            for (int left = 0; left <= m - i; ++left) {
                int right = i + left - 1;
                // If we can take a left operation, update the DP value considering the left pick
                if (left > 0) {
                    dp[left][i - left] = Math.max(dp[left][i - left], dp[left - 1][i - left] + multipliers[right] * nums[left - 1]);
                }
                // If we can take a right operation, update the DP value considering the right pick
                if (i - left > 0) {
                    dp[left][i - left] = Math.max(dp[left][i - left], dp[left][i - left - 1] + multipliers[right] * nums[n - (i - lef
                }
                // If we have used all multipliers, update the maximum score if the current one is higher
                if (i == m) {
                    maxScore = Math.max(maxScore, dp[left][i - left]);
                }
            }
        }
        return maxScore; // Return the maximum score found
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maximumScore(vector<int>& nums, vector<int>& multipliers) {
        // No need for such a large negative initial value, as scores can be negative.
        // Let's use the minimum possible value for integers.
        const int minInt = numeric_limits<int>::min();

        // Size of the nums and multipliers arrays.
        int numSize = nums.size(), multiplierSize = multipliers.size();

        // Initializing a DP table where f[i][j] represents the maximum score possible
        // using the first i elements from the beginning and the first j elements
        // from the end of nums, after applying i + j multipliers.
        vector<vector<int>> dp(multiplierSize + 1, vector<int>(multiplierSize + 1, minInt));

        // Base case: no elements picked and no multipliers applied.
        dp[0][0] = 0;

        // Variable to store the final maximum score.
        int maxScore = minInt;

        // Loop through all valid combinations of picks.
        for (int i = 0; i <= multiplierSize; ++i) {
            for (int j = 0; j <= multiplierSize - i; ++j) {
                // The kth multiplier to apply.
                int k = i + j - 1;

                // Update the dp value for picking from the beginning of nums.
                if (i > 0) {
                    dp[i][j] = max(dp[i][j], dp[i - 1][j] + multipliers[k] * nums[i - 1]);
                }

                // Update the dp value for picking from the end of nums.
                if (j > 0) {
                    dp[i][j] = max(dp[i][j], dp[i][j - 1] + multipliers[k] * nums[numSize - j]);
                }

                // If we've used all multipliers, compare with maxScore.
                if (i + j == multiplierSize) {
                    maxScore = max(maxScore, dp[i][j]);
                }
            }
        }

        // Return the maximum score found.
        return maxScore;
    }
};

```

Typescript Solution

```typescript
function maximumScore(nums: number[], multipliers: number[]): number {
    // Initialize a large negative number that will never be reached in the problem.
    const negativeInfinity = Number.MIN_SAFE_INTEGER;

    // Retrieve the lengths of the input arrays.
    const numLength = nums.length;
    const multipliersLength = multipliers.length;

    // Initialize a DP table with dimensions of multipliersLength + 1 and fill with negative infinity.
    const dp = new Array(multipliersLength + 1)
        .fill(0)
        .map(() => new Array(multipliersLength + 1).fill(negativeInfinity));

    // Base case: if no numbers are picked, score is 0.
    dp[0][0] = 0;

    // This will hold the answer to the problem.
    let maxScore = negativeInfinity;

    // Iterate through all possible counts of elements picked from the beginning and end.
    for (let i = 0; i <= multipliersLength; ++i) {
        for (let j = 0; j <= multipliersLength - i; ++j) {
            // Calculate the index for the current multiplier based on i and j.
            const multiplierIndex = i + j - 1;

            // If an element from the front is picked, update the DP table accordingly.
            if (i > 0) {
                dp[i][j] = Math.max(
                    dp[i][j],
                    dp[i - 1][j] + nums[i - 1] * multipliers[multiplierIndex]
                );
            }

            // If an element from the back is picked, update the DP table accordingly.
            if (j > 0) {
                dp[i][j] = Math.max(
                    dp[i][j],
                    dp[i][j - 1] + nums[numLength - j] * multipliers[multiplierIndex]
                );
            }

            // If all multipliers have been used, consider the result as a possible max score.
            if (i + j === multipliersLength) {
                maxScore = Math.max(maxScore, dp[i][j]);
            }
        }
    }

    // Return the calculated maxScore after considering all possibilities.
    return maxScore;
}

```

Time and Space Complexity

The provided Python code aims to find the maximum score from multiplying elements of `nums` with `multipliers` based on specific rules. The function `maximumScore` utilizes dynamic programming with a 2D array `f` to store intermediate results.

Time Complexity:

The time complexity depends on two loops that iterate over the range `(m + 1)`. The outer loop variable `i` runs from `0` to `m`, and for each value of `i`, the inner loop variable `j` runs up to `(m - i)`. This essentially results in a total of roughly `m/2 * m/2` iterations, which can be approximated to `O(m^2)` where `m` is the length of the `multipliers` list.

Moreover, the updates within the if conditions inside the nested loops also operate in constant time. Therefore, no additional complexity is added there.

Hence, the overall time complexity of this code is `O(m^2)`.

Space Complexity:

The space complexity is determined by the size of the 2D array `f`, which has dimensions `(m + 1)` by `(m + 1)`. So the total space used is `(m + 1) * (m + 1)`, resulting in a space complexity of `O(m^2)` as well.

Thus, the overall space complexity of the algorithm is `O(m^2)`.