2907. Maximum Profitable Triplets With Increasing Prices I

Medium Array Dynamic Programming Leetcode Link

Problem Description

In this problem, we're given two 0-indexed integer arrays: `prices` and `profits`. Both arrays have a length of `n`, and they represent items in a store where the `i`th item has a price of `prices[i]` and a potential profit of `profits[i]`.

The task is to select three different items to maximize our profit under a specific condition: we must pick items in a manner that their indices `i`, `j`, and `k` follow `i < j < k`, and their prices also follow an increasing order, `prices[i] < prices[j] < prices[k]`. In simpler terms, the second item's price and index should be higher than the first, and the third item's price and index should be higher than the second.

Our profit from such a selection would be the sum of the corresponding profits of these items: `profits[i] + profits[j] + profits[k]`.

The output should be the maximum achievable profit following these rules. However, if it's impossible to find such a triplet of items, the function should return `-1`.

Intuition

The intuitive approach to solve this problem is by trying to break it down and identify subproblems that make up the whole. We do this by picking one piece of the problem to focus on and then solving the remaining aspects around it.

In this case, we can streamline our search by fixing one of the elements as the pivotal middle element - that's the `j` index or our second item's position. The aim is to select this middle element such that we can find one `i` and one `k` where both the index and price restrictions are satisfied (`i < j < k` with `prices[i] < prices[j] < prices[k]`).

Once `j` is fixed, we look for the maximum profit on the left (`profits[i]`) and the maximum profit on the right (`profits[k]`) that fulfill our conditions related to both price and index ordering. By doing this for every possible middle element, we ensure that no possible profitable combination is missed.

The variables `left` and `right` in the solution help us keep track of the maximum profits found to the left and right of our fixed middle element. If we find valid `left` and `right` profits, we calculate the total profit for the current selection and compare it with the maximum found so far (`ans`). Repeating this process and updating `ans` when we find a greater total profit ensures that we arrive at the highest possible profit by the end of the function.

This strategy reduces the problem from a three-dimensional one (where we could naively check every possible triplet of items) to a one-dimensional problem focused around the chosen mid-point, which is much more efficient.

Solution Approach

The implementation of the solution begins by first understanding that the problem can be approached by iterating through the list of items and fixing the middle element of the triplet to simplify the process. This approach falls under the pattern of iteration combined with local optimization.

Here's how the solution works:

1. **Initialization**: The variable `ans` is initialized with `-1`, this will store the maximum profit or remain `-1` if no valid trio is found.
2. **Outer Loop**: We start by creating an outer loop to iterate over each potential middle item `j`:

   ```
   for j, x in enumerate(profits):
   ```

3. **Initialize Left and Right Profit**: For each `j`, initialize two variables `left` and `right` both to `0`. These will store the maximum profits to the left and to the right of `j` respectively.
4. **Find Left Maximum Profit**: We iterate from the start of the list to the element just before `j` to find the maximum profit (`profits[i]`) such that the price at `i` is less than the price at `j`:

   ```
   for i in range(j):
       if prices[i] < prices[j] and left < profits[i]:
           left = profits[i]
   ```

5. **Find Right Maximum Profit**: Similarly, we iterate from the element after `j` to the end of the list to find the maximum profit (`profits[k]`) such that the price at `j` is less than the price at `k`:

   ```
   for k in range(j + 1, n):
       if prices[j] < prices[k] and right < profits[k]:
           right = profits[k]
   ```

6. **Calculate and Compare Profit**: If we found valid `left` and `right` profits (meaning that `left` and `right` are not `0`), we calculate the total profit for this triplet, which is `left + x + right`, where `x` is `profits[j]`. We then check if this total profit is greater than the current maximum profit stored in `ans`:

   ```
   if left and right:
       ans = max(ans, left + x + right)
   ```

7. **Return Result**: After iterating through all possible middle elements, `ans` contains the maximum profit. The function finally returns `ans` which is either the maximum profit found or `-1` if no valid triplet was found.

The algorithm efficiently updates its best-known solution with each iteration, employing an iterative approach, while breaking the problem down into smaller sub-tasks (finding left and right maximums) that can be solved independently of each other.

This algorithm runs in `O(n^2)` time, because for each of the `n` elements chosen as a middle element, it performs potentially `O(n)` work to find `left` and `O(n)` work to find `right`. While not the most efficient for very large numbers of items, it's a reasonable approach that improves over a naive `O(n^3)` solution that would examine all possible triplets one by one.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have the following `prices` and `profits` arrays:

- `prices = [3, 1, 4, 2]`
- `profits = [4, 1, 5, 3]`

And we wish to maximize our profit by selecting three items such that their `prices` and indices are in strictly increasing order.

The process will be as follows:

1. Initialize `ans = -1`, as we have not found any triplet yet.
2. The outer loop iterates with `j` starting from 0 to `n-1`. In this case, `n` is 4. So `j` will take on values 0, 1, 2, and 3 one by one.
3. For each `j`, initialize `left = 0` and `right = 0`.
4. For `j = 0`, there is no element to the left of `profits[j]`, so we can skip and continue with `j = 1`.
5. For `j = 1`, we have:
   - `left = 0`: No profit picked yet.
   - Start `i` loop from 0 to (j-1), i.e., `i = 0`.
   - Check if `prices[0] < prices[1]`, which is `3 < 1`. This is false, so we skip.
6. Move to `j = 2`:
   - `left = 0`: No profit picked yet.
   - Start `i` loop:
     - `i = 0`: `prices[0] < prices[2]` is `3 < 4`, true, and `profits[0]` is 4, which is greater than `left`.
     - Update `left = 4`.
     - `i = 1`: `prices[1] < prices[2]` is `1 < 4`, true, but `profits[1]` is 1, which is not greater than the current `left`.
     - Keep `left = 4`.
   - Start `k` loop from (j+1) to `n`, i.e., `k = 3`.
   - `prices[2] < prices[3]` is `4 < 2`. This is false, so no right profit is updated.
7. Continue to `j = 3`:
   - `left = 0` again.
   - Start `i` loop:
     - `i = 0`: `prices[0] < prices[3]` is `3 < 2`. This is false, so continue.
     - `i = 1`: `prices[1] < prices[3]` is `1 < 2`. This is true and `profits[1]` is 1.
     - Update `left = 1`.
     - `i = 2`: `prices[2] < prices[3]`. This is false, so keep `left = 1`.
   - Since we are at the last element, there are no elements to the right for comparison; hence, no update to `right`.

Final step:

- The maximum profit calculated with `j=2` was `left + profits[j] + right`, but because we did not find a valid `right`, we do not update `ans`.
- Therefore, the output of the algorithm would be `-1` since we never updated our initial `ans` value from `-1` given that no valid triplet could satisfy the conditions of the problem with the provided example arrays.

Python Solution

```python
class Solution:
    def max_profit(self, prices: List[int], profits: List[int]) -> int:
        # Get the number of available prices/profits
        num_prices = len(prices)

        # Initialize the maximum profit to -1 to signify that initially, there is no profit
        max_profit_value = -1

        # Iterate through each price and associated profit
        for current_index, current_profit in enumerate(profits):
            # Initialize the max left and right profits to zero
            max_profit_left = max_profit_right = 0

            # Search to the left of the current index for a price that is lower than the current price
            # and also search for the maximum left profit that is associated with such a price
            for left_index in range(current_index):
                if prices[left_index] < prices[current_index] and max_profit_left < profits[left_index]:
                    max_profit_left = profits[left_index]

            # Search to the right of the current index for a price that is higher than the current price
            # and also search for the maximum right profit associated with such a price
            for right_index in range(current_index + 1, num_prices):
                if prices[current_index] < prices[right_index] and max_profit_right < profits[right_index]:
                    max_profit_right = profits[right_index]

            # If both left and right profits are found,
            # calculate the sum of the current profit and the left and right profits
            # and update the maximum profit if greater than the current maximum
            if max_profit_left and max_profit_right:
                total_profit = max_profit_left + current_profit + max_profit_right
                max_profit_value = max(max_profit_value, total_profit)

        # Return the maximum profit after examining all possible transactions
        return max_profit_value

```

Java Solution

```java
class Solution {
    public int maxProfit(int[] prices, int[] profits) {
        // 'n' holds the total number of elements in the prices and profits arrays, as they are of the same length.
        int n = prices.length;

        // 'maxProfit' will keep track of the maximum profit found. Initialized to -1 as a default value.
        int maxProfit = -1;

        // Iterate through all the possible purchase indexes.
        for (int current = 0; current < n; ++current) {
            // 'maxLeftProfit' will keep track of the maximum profit to the left of the 'current' index.
            int maxLeftProfit = 0;

            // 'maxRightProfit' will keep track of the maximum profit to the right of the 'current' index.
            int maxRightProfit = 0;

            // Checking for the maximum profit that can be obtained before the current index.
            for (int i = 0; i < current; ++i) {
                // Only consider the profit if the price previously was less than the current price.
                if (prices[i] < prices[current]) {
                    maxLeftProfit = Math.max(maxLeftProfit, profits[i]);
                }
            }

            // Checking for the maximum profit that can be obtained after the current index.
            for (int k = current + 1; k < n; ++k) {
                // Only consider the profit if the current price is less than the future price.
                if (prices[current] < prices[k]) {
                    maxRightProfit = Math.max(maxRightProfit, profits[k]);
                }
            }

            // If there's a possible profit both to the left and right of the current index, check if this transaction sequence is th
            if (maxLeftProfit > 0 && maxRightProfit > 0) {
                maxProfit = Math.max(maxProfit, maxLeftProfit + profits[current] + maxRightProfit);
            }
        }

        // Return the maximum profit found. If no profitable sequence is found, return -1.
        return maxProfit;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For max function
using namespace std;

class Solution {
public:
    int maxProfit(vector<int>& prices, vector<int>& profits) {
        int numDays = prices.size(); // Number of days is the size of the prices vector
        int maxProfit = -1; // Initialize maxProfit to -1 (will signify no profit if it stays the same)

        // Iterate through each day to find the maximum profit
        for (int today = 0; today < numDays; ++today) {
            int bestProfitBeforeToday = 0; // Maximum profit from previous transactions before today
            int bestProfitAfterToday = 0; // Maximum profit from transactions after today

            // Calculate the best profit from transactions before today
            for (int prevDay = 0; prevDay < today; ++prevDay) {
                if (prices[prevDay] < prices[today]) {
                    bestProfitBeforeToday = max(bestProfitBeforeToday, profits[prevDay]);
                }
            }

            // Calculate the best profit from transactions after today
            for (int nextDay = today + 1; nextDay < numDays; ++nextDay) {
                if (prices[today] < prices[nextDay]) {
                    bestProfitAfterToday = max(bestProfitAfterToday, profits[nextDay]);
                }
            }

            // If there is a profit possible before and after today, update maxProfit
            if (bestProfitBeforeToday > 0 && bestProfitAfterToday > 0) {
                int totalProfit = bestProfitBeforeToday + profits[today] + bestProfitAfterToday;
                maxProfit = max(maxProfit, totalProfit);
            }
        }

        // Return the maximum profit that can be achieved
        return maxProfit;
    }
};

```

Typescript Solution

```typescript
function maxProfit(prices: number[], profits: number[]): number {
    // Get the length of the arrays.
    const numPrices = prices.length;

    // Initialize the answer to -1 to indicate no profit if not updated.
    let maximumProfit = -1;

    // Iterate over all prices.
    for (let current = 0; current < numPrices; ++current) {
        // Initialize the maximum profit before and after the current index.
        let maxProfitBefore = 0;
        let maxProfitAfter = 0;

        // Compute the maximum profit before the current index.
        for (let before = 0; before < current; ++before) {
            if (prices[before] < prices[current]) {
                maxProfitBefore = Math.max(maxProfitBefore, profits[before]);
            }
        }

        // Compute the maximum profit after the current index.
        for (let after = current + 1; after < numPrices; ++after) {
            if (prices[current] < prices[after]) {
                maxProfitAfter = Math.max(maxProfitAfter, profits[after]);
            }
        }

        // If there is a profit before and after the current index,
        // update the maximum profit if the sum of profits is greater than the current maximum.
        if (maxProfitBefore > 0 && maxProfitAfter > 0) {
            maximumProfit = Math.max(maximumProfit, maxProfitBefore + profits[current] + maxProfitAfter);
        }
    }

    // Return the maximum possible profit, or -1 if not found.
    return maximumProfit;
}

```

Time and Space Complexity

The time complexity of this function `maxProfit` is `O(n^2)`. This is because there are two nested loops. The outer loop runs `n` times, where `n` is the length of the input lists `prices` and `profits`. For each iteration of the outer loop, the inner loops (one for calculating `left` and another for calculating `right`) in total can iterate up to `n` times in the worst case (when `j` is at the center of the array). Since these inner loops are nested within the outer loop, the total number of operations can be up to `n * n`, which simplifies to `O(n^2)`.

The space complexity of this function is `O(1)`. This is because the extra space used by the function does not depend on the input size `n`. Only a fixed number of variables (`ans`, `left`, `right`, `n`, `j`, `x`, `i`, and `k`) are used to keep track of the maximum profit as well as loop counters and values. Therefore, the space used remains constant regardless of the input size.