2291. Maximum Profit From Trading Stocks

Medium Array Dynamic Programming Leetcode Link

Problem Description

The problem is essentially a variation of the classic 0/1 Knapsack problem. You are given two arrays `present` and `future` of the same length, which represent the current and future prices of stocks. With a given `budget`, your task is to determine the maximum profit that can be gained by buying stocks at present prices and selling them in the future considering that you can't spend more than the budget. You can buy each stock at most once. The maximum profit is calculated by finding the difference between the future price and the present price for the stocks you decide to buy and summing these differences up. The challenge is to select the stocks in such a way that the total cost does not exceed the budget while maximizing the profit.

Intuition

The intuition behind the solution is to use dynamic programming (DP) to solve the problem in a bottom-up manner. We define a DP array `f` with a length of `budget + 1`, where each index `j` represents the maximum profit that can be achieved with a budget `j`. We initialize the DP array with zero values because initially, with a budget of 0, we cannot achieve any profit.

We iterate over each stock, and for each stock, we iterate backward through the DP array starting from the current budget down to the price of the current stock. This reverse iteration is necessary to ensure that each stock is considered at most once. For each budget `j`, we update the DP value at `j` to be the maximum of its current value and the value at `j - present_stock_price` plus the profit that could be made by buying the current stock (`future_stock_price - present_stock_price`).

This way, `f[j]` represents the best decision we can make considering the first `i` stocks for a given budget `j`. When we reach `f[budget]`, it contains the maximum profit we can make given our total budget.

The reverse iteration of the budget within the loop makes sure that we do not reuse the same stock multiple times as it simulates buying stocks exactly once by ensuring the current stock's budget isn't used previously in the same iteration.

The problem is solved when we fill the DP array, at which point `f[-1]` (which is the same as `f[budget]`) contains the solution, i.e., the maximum possible profit.

Solution Approach

The solution uses dynamic programming as its core algorithmic approach. The idea is to build up a solution step-by-step for incremental values of the budget, ensuring that at each step, the maximum profit that can be obtained without exceeding that budget is accounted for.

The key data structure used here is a list `f` with a length of `budget + 1`. This list is initialized with zeros, as initially, no profit can be made without spending money (`budget = 0`). The list `f` is used to store the maximum profit possible for every budget from `0` to `budget`.

Let's delve deeper into the algorithm:

1. We iterate over each stock with a loop `for a, b in zip(present, future):`. The variables `a` and `b` represent the present and future price of each stock, respectively.
2. Within each iteration over the stocks, we iterate over the budgets from high to low `for j in range(budget, a - 1, -1):`. This is the reverse iteration that ensures we only consider buying a stock once.
3. At each budget `j`, we make a decision: Can buying this stock at its current price `a` increase our profit when selling it in the future for price `b`? To make this decision, we check if `f[j - a] + b - a` (which is our previous best profit at a reduced budget `j - a`, plus the profit from buying and selling this stock) is greater than the current best profit at `f[j]`.
4. If it is beneficial to buy the stock, we update `f[j]` with the new maximum profit `max(f[j], f[j - a] + b - a)`. Here `f[j - a] + b - a` represents the profit made by buying this stock (which is `future_price - present_price`) plus the profit that we could have made with the remaining budget after buying the stock (`f[j - a]`).
5. Finally, after considering all stocks for all possible budgets, we return `f[-1]`. Due to zero-indexing in Python, `f[-1]` refers to the last element of the list `f`, which corresponds to the maximum profit that can be achieved with the full budget.

The algorithm efficiently determines the maximum profit that can be achieved under the constraint of the budget by examining each possibility only once. This is made possible due to the nature of dynamic programming, which breaks down the problem into simpler subproblems and reuses their solutions.

Example Walkthrough

Let's use a simple example to illustrate the solution approach. Suppose you have the following input:

- `present`: [1, 2, 3]
- `future`: [2, 3, 5]
- `budget`: 5

Using these inputs, we want to find the maximum profit we can make by buying stocks at present prices and selling them at future prices without exceeding our budget.

Initialization:

- We initialize our DP list `f` of length `budget + 1`, which is `5 + 1 = 6`, with zeros, so `f` becomes `[0, 0, 0, 0, 0, 0]`.

Iteration over stocks:

1. First stock: `present` is 1 and `future` is 2.
   - We check budgets starting from 5 down to 1 (the price of the stock).
   - At `j = 5`, we consider buying this stock by comparing `f[5 - 1] + (2 - 1)` with `f[5]`. Since `f[4] + 1` is 1 and `f[5]` is 0, we update `f[5]` to 1.
   - We repeat this for budget `j = 4, 3, 2, 1`. The list `f` now becomes `[0, 1, 1, 1, 1, 1]`.
2. Second stock: `present` is 2 and `future` is 3.
   - This time we start from the budget of 5 and go down to 2.
   - At `j = 5`, we check if buying this stock is better than any previous decision by checking `f[5 - 2] + (3 - 2)` against `f[5]`. We calculate `f[3] + 1` is 2 and `f[5]` is 1, so we update `f[5]` to 2.
   - We do similar updates for `j = 4` and `j = 3`. Now, `f` is `[0, 1, 1, 2, 2, 2]`.
3. Third stock: `present` is 3 and `future` is 5.
   - We iterate from budget 5 to 3.
   - At `j = 5`, we compare `f[5 - 3] + (5 - 3)` with `f[5]`. Thus `f[2] + 2` is 3 and `f[5]` is 2, we update `f[5]` to 3.
   - At `j = 4`, `f[4]` remains 2 since we can't buy this stock with a budget of 1.
   - So `f` becomes `[0, 1, 1, 2, 2, 3]`.

Returning the result:

- The last element in list `f` is 3, so we return `f[-1]` which is the maximum profit possible with the initial budget, which is 3.

This walkthrough shows that by using dynamic programming, we can efficiently calculate the maximum profit without exceeding our budget by making smart decisions at each step.

Python Solution

```python
class Solution:
    def maximumProfit(self, present_values: List[int], future_values: List[int], budget: int) -> int:
        # Create a table to store the maximum profit that can be achieved with each budget
        profit_table = [0] * (budget + 1)

        # Iterate over each item pair of present and future values
        for present_value, future_value in zip(present_values, future_values):
            # Update the profit table in reverse to avoid using updated values in the same iteration
            for current_budget in range(budget, present_value - 1, -1):
                # Calculate the profit by buying an item at present value and selling at future value
                profit = future_value - present_value

                # Update the profit table with the maximum profit so far
                # Either keep the current profit for the budget or
                # Take the profit by investing present_value and adding to the profit found with the remaining budget
                profit_table[current_budget] = max(profit_table[current_budget],
                                                   profit_table[current_budget - present_value] + profit)

        # The last element of the profit table holds the maximum profit achievable with the given budget
        return profit_table[-1]

```

Java Solution

```java
class Solution {
    public int maximumProfit(int[] presentValues, int[] futureValues, int budget) {
        int numberOfItems = presentValues.length; // Total number of items

        // dp array to store the maximum profit for each budget upto the given budget
        int[] dp = new int[budget + 1];

        // Iterate over each item
        for (int i = 0; i < numberOfItems; ++i) {
            int presentValue = presentValues[i];
            int futureValue = futureValues[i];

            // Iterate over all possible remaining budgets in reverse
            // to avoid using same item's profit more than once
            for (int currentBudget = budget; currentBudget >= presentValue; --currentBudget) {
                // Update dp array with the maximum profit achievable with the current budget
                dp[currentBudget] = Math.max(dp[currentBudget], dp[currentBudget - presentValue] + futureValue - presentValue);
            }
        }

        // Return the maximum profit that can be achieved with the given budget
        return dp[budget];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Solution {
public:
    // This function calculates the maximum profit given the present and future prices of gifts
    // within a specified budget constraint.
    int maximumProfit(vector<int>& presentPrices, vector<int>& futurePrices, int budget) {
        int n = presentPrices.size(); // Number of gifts
        vector<int> dp(budget + 1, 0); // Initialize DP array to store max profits with 0 as default

        // Iterate over all the gifts
        for (int i = 0; i < n; ++i) {
            int buyPrice = presentPrices[i];  // Buy price of the current gift
            int sellPrice = futurePrices[i];  // Future (sell) price of the current gift

            // Traverse the budget from back to front to avoid using updated states
            for (int j = budget; j >= buyPrice; --j) {
                // Update the dp array for the current budget 'j' with the greater value between
                // the previous dp value for 'j' and the dp value for 'j - buyPrice'
                // added with the profit of selling the current gift (sellPrice - buyPrice).
                dp[j] = max(dp[j], dp[j - buyPrice] + sellPrice - buyPrice);
            }
        }
        return dp[budget]; // Return the maximum profit for the given budget
    }
};

```

Typescript Solution

```typescript
function maximumProfit(currentPrices: number[], futurePrices: number[], budget: number): number {
    // Initialize a dynamic programming array of length 'budget + 1' with zeros.
    const dp = new Array<number>(budget + 1).fill(0);

    // Iterate over each item (both present and future prices).
    for (let i = 0; i < currentPrices.length; ++i) {
        // Capture the current and future price of the item.
        const currentPrice = currentPrices[i];
        const futurePrice = futurePrices[i];

        // Iterate over the budget from high to low to avoid recomputation.
        for (let currentBudget = budget; currentBudget >= currentPrice; --currentBudget) {
            // Calculate the maximum profit by comparing whether to buy this item or not.
            dp[currentBudget] = Math.max(
                dp[currentBudget],          // Not buying the item
                dp[currentBudget - currentPrice] + futurePrice - currentPrice // Buying the item
            );
        }
    }

    // The last element in the dynamic programming array contains the maximum profit.
    return dp[budget];
}

```

Time and Space Complexity

The given Python function `maximumProfit` aims to determine the maximum profit that can be made given a budget constraint and lists of present and future values of items.

Time Complexity

The time complexity of the function is determined by the nested loops present in the code. The outer loop runs once for each pair of present and future values, while the inner loop runs for each value of the budget from the current item's present value to 1, decrementing on each iteration.

Given that `n` is the total number of items (length of the `present` or `future` list), and `budget` is the maximum budget available, the outer loop runs `n` times, and the inner loop runs at most `budget` times for each iteration of the outer loop.

Therefore, the overall time complexity is `O(n * budget)`, where `n` is the length of the lists.

Space Complexity

The space complexity is determined by the storage used by the algorithm outside of the input storage. In this case, a list `f` of size `budget + 1` is created to store temporary values computed during the execution of the function.

This results in a space complexity of `O(budget)`, since the list `f` grows linearly with respect to the `budget` parameter.