2431. Maximize Total Tastiness of Purchased Fruits

Medium Array Dynamic Programming Leetcode Link

Problem Description

In this problem, we are dealing with fruit shopping with some constraints and aids in order to maximize the total tastiness of the fruits we can buy. We are provided with two arrays – one depicts the price of each fruit, and the other represents the tastiness of each fruit. The length of both arrays is the same, which indicates the number of different fruits available. Additionally, we are given two constraints in the form of integers, `maxAmount` and `maxCoupons`. Here's a summary of the conditions and objectives:

- We aim to choose fruits such that the sum of their tastiness values is maximized.
- The total price of the chosen fruits cannot exceed `maxAmount`.
- We can use `maxCoupons` coupons, where each coupon allows buying a fruit at half price (rounded down).
- Each fruit can only be purchased once, and we can apply at most one coupon to each fruit.

The key challenge is to make strategic decisions about which fruits to buy and on which ones to use the coupons, in order to achieve the maximum total tastiness without exceeding the budget constraints.

Intuition

The intuition behind the solution is to explore each possibility of buying a fruit with and without a coupon, and not buying it at all. This kind of problem hints at a recursive approach, likely exploring all options via depth-first search and applying memoization to optimize repeated subproblems — a common dynamic programming strategy.

To arrive at a solution:

1. We need to consider each fruit and decide one of three actions: not buying it, buying it at full price (if we have enough amount), or buying it at half price using a coupon (if we have enough amount and coupons left).
2. This creates a decision tree where each node represents these possibilities, and we traverse these nodes in depth-first order.
3. Since similar subproblems will be solved repeatedly (with overlapping decisions for subsets of fruits, remaining amount, and coupons), we can use memoization to store already computed solutions of subproblems. In Python, this can be achieved using the `@cache` decorator for our recursive function.
4. The result is the maximum tastiness we can achieve by considering the options for buying, not buying, or coupon-buying each fruit while keeping track of remaining amount and coupons.

The given solution leverages memoization for efficient computation and relies on the recursive call `dfs(i, j, k)` where `i` is the current index in the fruit arrays, `j` is the remaining amount, and `k` is the remaining coupons. At each step, it makes recursive calls to consider different scenarios and takes the maximum tastiness outcome.

Solution Approach

The problem is solved using a combination of recursive depth-first search (DFS) and dynamic programming (DP) with memoization. Here's a step-by-step explanation of the solution approach based on the provided Python code:

1. A recursive helper function `dfs(i, j, k)` is defined, which represents the state of the problem after considering the first `i` fruits, with `j` being the remaining budget (maxAmount subtracted by the cost of fruits bought so far), and `k` being the remaining coupons.

2. The base case is when `i` equals the length of the `price` array, meaning that we have considered all the fruits. In this case, we cannot increase tastiness anymore, so the function returns 0.

3. For each fruit `i`, the function computes the maximum tastiness by making a recursive call to itself to represent the scenario of not buying the current fruit: `dfs(i + 1, j, k)`.

4. It then checks if the current fruit can be bought without a coupon (provided the current budget `j` is at least the price of the fruit). If so, it calculates the tastiness of buying it at full price: `dfs(i + 1, j - price[i], k) + tastiness[i]` and updates the answer with the better option between not buying and buying at full price.

5. The function also checks if the current fruit can be bought with a coupon (provided there is at least one coupon and the current budget `j` is sufficient for the half price). In that case, it calculates the tastiness of buying it at half price: `dfs(i + 1, j - price[i] // 2, k - 1) + tastiness[i]` and updates the answer with the best option among not buying, buying at full price, and buying at half price.

6. Memoization is applied to the recursive function using the `@cache` decorator which stores the result of each unique state `(i, j, k)` so that repeated computations for the same state are avoided. This is where dynamic programming comes into play to optimize the solution.

7. Lastly, the `dfs(0, maxAmount, maxCoupons)` function call is made to kick off the decision process starting with the first fruit, the full budget, and all the coupons available. The returned value from this call will be the maximum possible total tastiness that can be achieved.

In summary, this solution leverages dynamic programming with memoization to efficiently compute the maximum tastiness achievable, considering the constraints of the budget and coupons. The algorithm tests all permutations of decisions (to buy with or without a coupon, or not to buy) and remembers solutions to subproblems to avoid redundant calculations.

Example Walkthrough

Suppose we have the following small example:

- `price`: [3, 5, 6]
- `tastiness`: [5, 6, 5]
- `maxAmount`: 8
- `maxCoupons`: 1

Let's walk through the solution approach for this example:

1. We will call our recursive function `dfs(i, j, k)` where `i` is the index of the current fruit we are considering, `j` is the remaining budget, and `k` is the number of coupons left.

2. Initially, we call `dfs(0, 8, 1)` because we start with the first fruit (index 0), we have the full budget of 8, and one coupon available.

3. The decision for the first fruit (price: 3, tastiness: 5) is as follows:

   - Not buying it: We move to the next fruit with the same budget and coupons. Recursive call: `dfs(1, 8, 1)`.
   - Buying it at full price: We spend 3 from our budget, so we move to the next fruit with a budget of 5 and the same number of coupons. Recursive call with additional tastiness: `dfs(1, 5, 1) + 5`.
   - Buying it at half price with a coupon: We spend `3 // 2 = 1` from our budget, keep the tastiness, and reduce the number of coupons by one. Recursive call with additional tastiness: `dfs(1, 7, 0) + 5`.
4. The helper function will compare these scenarios using the recursive calls and, with the aid of memoization, return the maximum tastiness of these choices.

5. This process continues for the next fruits as well. For example, for fruit with price 5 and tastiness 6 (index 1), we would have these possible calls assuming we didn't buy the first fruit:

   - Not buying it: `dfs(2, 8, 1)`.
   - Buying it at full price, if budget allows: `dfs(2, 3, 1) + 6`.
   - Buying it at half price with a coupon: `dfs(2, 6, 0) + 6`.
6. Note that for the second recursive call, the budget would not allow buying the fruit at the full price from the given example, thus that option would be ignored.

7. The memoization would remember outcomes of certain states such as `dfs(1, 5, 1)` and avoid recalculating them if they come up again in a different branch of the decision tree.

8. The recursion and decision-making continue until all possible combinations of buying/not buying fruits with and without the usage of coupons are exhaustively explored.

Considering our example, an optimal solution would be to:

- Use the coupon to buy the first fruit at a 50% discount, getting 5 tastiness and reducing our budget to 7.
- Buy the second fruit at full price with the remaining budget, getting an additional 6 tastiness.
- We don't have enough budget to buy the third fruit.

The maximum total tastiness can be achieved with these decisions is 11, from the first and second fruit, and this is the value returned by `dfs(0, 8, 1)`.

Python Solution

```python
from functools import lru_cache
from typing import List

class Solution:
    def maxTastiness(
        self, prices: List[int], tastiness_values: List[int],
        max_amount: int, max_coupons: int
    ) -> int:

        # Apply memoization to our DFS (Depth-First Search) function
        # to avoid recalculating results for the same state
        @lru_cache(maxsize=None)
        def dfs(index, remaining_amount, coupons_left):
            # Base Case: If all items have been considered
            if index == len(prices):
                return 0

            # Decision 1: Skip the current item, no change to budget or coupons
            max_taste = dfs(index + 1, remaining_amount, coupons_left)

            # Decision 2: Buy the current item without a coupon if the price can be afforded
            if remaining_amount >= prices[index]:
                max_taste = max(max_taste,
                                dfs(index + 1,
                                    remaining_amount - prices[index],
                                    coupons_left) + tastiness_values[index])

            # Decision 3: Buy the current item with a coupon if possible
            if remaining_amount >= prices[index] // 2 and coupons_left:
                max_taste = max(max_taste,
                                dfs(index + 1,
                                    remaining_amount - prices[index] // 2,
                                    coupons_left - 1) + tastiness_values[index])
            return max_taste

        # Start the DFS from the first item, given the max amount and coupons
        return dfs(0, max_amount, max_coupons)

# Example of how to use the modified Solution class:
# solution = Solution()
# max_taste = solution.maxTastiness([2, 5, 3], [4, 7, 5], 5, 1)
# print(max_taste) # Should print the result of the maxTastiness computation
```

Java Solution

```java
class Solution {
    private int[][][] memo; // 3D memoization array to store the results of subproblems
    private int[] prices;   // Array to store prices of items
    private int[] tastinessValues; // Array to store tastiness values of items
    private int itemCount;  // The number of items

    public int maxTastiness(int[] prices, int[] tastinessValues, int maxAmount, int maxCoupons) {
        itemCount = prices.length;
        this.prices = prices;
        this.tastinessValues = tastinessValues;
        memo = new int[itemCount][maxAmount + 1][maxCoupons + 1];
        // Call the recursive function starting at the first item, with full budget, and all coupons available
        return dfs(0, maxAmount, maxCoupons);
    }

    private int dfs(int itemIndex, int remainingAmount, int remainingCoupons) {
        // Base case: when all items have been considered
        if (itemIndex == itemCount) {
            return 0;
        }

        // Return the stored result if this subproblem has already been computed
        if (memo[itemIndex][remainingAmount][remainingCoupons] != 0) {
            return memo[itemIndex][remainingAmount][remainingCoupons];
        }

        // Case 1: Skip the current item and go to the next
        int maxValue = dfs(itemIndex + 1, remainingAmount, remainingCoupons);

        // Case 2: Buy the current item without a coupon if enough amount remains
        if (remainingAmount >= prices[itemIndex]) {
            maxValue = Math.max(maxValue, dfs(itemIndex + 1, remainingAmount - prices[itemIndex], remainingCoupons) + tastinessValues
        }

        // Case 3: Buy the current item with a coupon if a coupon and enough amount remain
        if (remainingAmount >= prices[itemIndex] / 2 && remainingCoupons > 0) {
            maxValue = Math.max(maxValue, dfs(itemIndex + 1, remainingAmount - prices[itemIndex] / 2, remainingCoupons - 1) + tastine
        }

        // Store the result in the memoization array before returning
        memo[itemIndex][remainingAmount][remainingCoupons] = maxValue;
        return maxValue;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <functional>

class Solution {
public:
    int maxTastiness(vector<int>& prices, vector<int>& tastinessValues, int maxAmount, int maxCoupons) {
        int n = prices.size();

        // Initialize the memoization array with 0 values.
        memset(memo, 0, sizeof memo);

        // Define the dfs (Depth-First Search) as a lambda function that can call itself.
        // i: current item index
        // currentAmount: current amount of money left
        // currentCoupons: current number of coupons left
        function<int(int, int, int)> dfs = [&](int i, int currentAmount, int currentCoupons) {
            // Base case: All items have been considered.
            if (i == n) return 0;

            // If this state has already been calculated, return the stored value.
            if (memo[i][currentAmount][currentCoupons]) return memo[i][currentAmount][currentCoupons];

            // Option 1: Do not buy the current item.
            int maxTaste = dfs(i + 1, currentAmount, currentCoupons);

            // Option 2: Buy the current item without using a coupon.
            if (currentAmount >= prices[i]) {
                maxTaste = max(maxTaste, dfs(i + 1, currentAmount - prices[i], currentCoupons) +
                                         tastinessValues[i]);
            }

            // Option 3: Buy the current item using a coupon, if available.
            if (currentAmount >= prices[i] / 2 && currentCoupons) {
                maxTaste = max(maxTaste, dfs(i + 1, currentAmount - prices[i] / 2, currentCoupons - 1) +
                                         tastinessValues[i]);
            }

            // Store the calculated value and return it.
            memo[i][currentAmount][currentCoupons] = maxTaste;
            return maxTaste;
        };

        // Start the dfs from the first item, with the maximum amount and coupons available.
        return dfs(0, maxAmount, maxCoupons);
    }

private:
    // Memoization array to store previously computed states:
    // f[item index][current amount][current coupons left]
    int memo[101][1001][6];
};
```

Typescript Solution

```typescript
// Define a type for the memoization array.
type MemoizationArray = number[][][];

// Define global variables for memoization, prices, tastinessValues, and the dimensions of our memo array.
let memo: MemoizationArray;
let prices: number[];
let tastinessValues: number[];
let maxAmount: number;
let maxCoupons: number;

// Initialize a 3D memo array with all values set to -1 (unvisited state).
function initializeMemo(n: number, maxAmount: number, maxCoupons: number): void {
    memo = Array.from({ length: n }, () =>
        Array.from({ length: maxAmount + 1 }, () =>
            Array(maxCoupons + 1).fill(-1)));
}

// Recursive function to calculate the max tastiness given the current state.
function dfs(i: number, currentAmount: number, currentCoupons: number): number {
    // Base case: All items have been considered.
    if (i === prices.length) return 0;

    // If this state has already been calculated, return the stored value.
    if (memo[i][currentAmount][currentCoupons] !== -1) {
        return memo[i][currentAmount][currentCoupons];
    }

    // Option 1: Do not buy the current item.
    let maxTaste = dfs(i + 1, currentAmount, currentCoupons);

    // Option 2: Buy the current item without using a coupon.
    if (currentAmount >= prices[i]) {
        maxTaste = Math.max(maxTaste, dfs(i + 1, currentAmount - prices[i], currentCoupons) + tastinessValues[i]);
    }

    // Option 3: Buy the current item using a coupon, if available.
    if (currentAmount >= Math.floor(prices[i] / 2) && currentCoupons > 0) {
        maxTaste = Math.max(maxTaste, dfs(i + 1, currentAmount - Math.floor(prices[i] / 2), currentCoupons - 1) + tastinessValues[i
    }

    // Store the calculated value in the memo array and return it.
    memo[i][currentAmount][currentCoupons] = maxTaste;
    return maxTaste;
}

// Wrapper function to start the recursive calculation.
function maxTastiness(inputPrices: number[], inputTastinessValues: number[], inputMaxAmount: number, inputMaxCoupons: number): numb
    prices = inputPrices;
    tastinessValues = inputTastinessValues;
    maxAmount = inputMaxAmount;
    maxCoupons = inputMaxCoupons;

    // Initialize the memo array with proper dimensions.
    initializeMemo(prices.length, maxAmount, maxCoupons);

    // Start the depth-first search from the first item, with the maximum amount and coupons available.
    return dfs(0, maxAmount, maxCoupons);
}
```

Time and Space Complexity

The given Python code defines a recursive function with memoization to solve a variation of the knapsack problem, where we are trying to maximize the tastiness of items chosen under certain constraints on the price and available coupons.

Time Complexity

The time complexity of the code is dictated by the number of states that need to be computed, which is determined by the number of decisions for each item, the range of `maxAmount` (denoted as `M`), and the number of coupons `maxCoupons` (denoted as `C`). The function `dfs` is called with different states represented by a combination of current item index `i`, remaining amount `j`, and remaining coupons `k`.

For each item, there are three choices: skip the item, take the item without a coupon, or take the item with a coupon (if available). Since we only move to the next item in each recursive call, there are `N` levels of recursion, where `N` is the total number of items.

There is a unique state for each combination of `(i, j, k)`. Since `i` can be in the range `[0, N]`, `j` can take on values from `[0, M]`, and `k` from `[0, C]`, the number of possible states is roughly `N * M * C`.

The recursion is memoized to ensure that each state is computed at most once. Therefore, the time complexity is `O(N*M*C)`.

Space Complexity

The space complexity of the code is governed by the storage required for:

1. The memoization cache, which needs to store the result for each unique state `(i, j, k)`, thus requiring a space complexity of `O(N*M*C)`.
2. The call stack for the recursion, which at maximum depth will be `O(N)`, as we have `N` levels of recursion in the worst case.

Thus, the overall space complexity of the code is also `O(N*M*C)` due to the cache size dominating the recursive call stack.