2431. Maximize Total Tastiness of Purchased Fruits Medium Array **Dynamic Programming** 

### **Leetcode Link**

# In this problem, we are dealing with fruit shopping with some constraints and aids in order to maximize the total tastiness of the

**Problem Description** 

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.

- 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.

We can use maxCoupons coupons, where each coupon allows buying a fruit at half price (rounded down).

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.

cannot increase tastiness anymore, so the function returns 0.

with the better option between not buying and buying at full price.

- 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.
- 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

## not buying the current fruit: dfs(i + 1, j, k).

and buying at half price.

be achieved.

• tastiness: [5, 6, 5]

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

budget, and k is the number of coupons left.

maxAmount: 8

maxCoupons: 1

**Solution Approach** 

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

3. For each fruit i, the function computes the maximum tastiness by making a recursive call to itself to represent the scenario of

- 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,
- 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

In summary, this solution leverages dynamic programming with memoization to efficiently compute the maximum tastiness

without a coupon, or not to buy) and remembers solutions to subproblems to avoid redundant calculations.

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

**Example Walkthrough** Suppose we have the following small example: • price: [3, 5, 6]

achievable, considering the constraints of the budget and coupons. The algorithm tests all permutations of decisions (to buy with or

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:

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

Buying it at half price with a coupon: We spend 3 // 2 = 1 from our budget, keep the tastiness, and reduce the number of

• Not buying it: We move to the next fruit with the same budget and coupons. Recursive call: dfs(1, 8, 1).

coupons. Recursive call with additional tastiness: dfs(1, 5, 1) + 5.

these possible calls assuming we didn't buy the first fruit:

up again in a different branch of the decision tree.

Considering our example, an optimal solution would be to:

We don't have enough budget to buy the third fruit.

max\_amount: int, max\_coupons: int

if index == len(prices):

@lru cache(maxsize=None)

return 0

return max\_taste

40 # solution = Solution()

self, prices: List[int], tastiness\_values: List[int],

# to avoid recalculating results for the same state

# Base Case: If all items have been considered

def dfs(index, remaining\_amount, coupons\_left):

if remaining\_amount >= prices[index]:

max\_taste = max(max\_taste,

# Apply memoization to our DFS (Depth-First Search) function

# Decision 1: Skip the current item, no change to budget or coupons

# Decision 2: Buy the current item without a coupon if the price can be afforded

remaining\_amount - prices[index],

remaining\_amount - prices[index] // 2,

coupons\_left - 1) + tastiness\_values[index])

max\_taste = dfs(index + 1, remaining\_amount, coupons\_left)

dfs(index + 1,

dfs(index + 1,

# Start the DFS from the first item, given the max amount and coupons

private int dfs(int itemIndex, int remainingAmount, int remainingCoupons) {

if (memo[itemIndex][remainingAmount][remainingCoupons] != 0) {

return memo[itemIndex][remainingAmount][remainingCoupons];

int maxValue = dfs(itemIndex + 1, remainingAmount, remainingCoupons);

// Case 2: Buy the current item without a coupon if enough amount remains

if (remainingAmount >= prices[itemIndex] / 2 && remainingCoupons > 0) {

// Store the result in the memoization array before returning

memo[itemIndex][remainingAmount][remainingCoupons] = maxValue;

// Case 3: Buy the current item with a coupon if a coupon and enough amount remain

int maxTastiness(vector<int>& prices, vector<int>& tastinessValues, int maxAmount, int maxCoupons) {

// Return the stored result if this subproblem has already been computed

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

#### 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.

• Not buying it: dfs(2, 8, 1).

that option would be ignored.

by dfs(0, 8, 1).

Python Solution

) -> int:

9

10

13

14

15

16

17

18

19

20

21

23

25

31

32

33

34

35

36

37

38

13

16

17

18

19

20

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

42

43

45

44 }

 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

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

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.

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.

7. The memoization would remember outcomes of certain states such as dfs(1, 5, 1) and avoid recalculating them if they come

from functools import lru\_cache from typing import List class Solution: def maxTastiness(

The maximum total tastiness can be achieved with these decisions is 11, from the first and second fruit, and this is the value returned

26 coupons\_left) + tastiness\_values[index]) 27 28 # Decision 3: Buy the current item with a coupon if possible if remaining\_amount >= prices[index] // 2 and coupons\_left: 29 max\_taste = max(max\_taste, 30

return dfs(0, max\_amount, max\_coupons)

return dfs(0, maxAmount, maxCoupons);

if (itemIndex == itemCount) {

return 0;

return maxValue;

// Base case: when all items have been considered

// Case 1: Skip the current item and go to the next

if (remainingAmount >= prices[itemIndex]) -

41 # max\_taste = solution.maxTastiness([2, 5, 3], [4, 7, 5], 5, 1)

39 # Example of how to use the modified Solution class:

42 # print(max\_taste) # Should print the result of the maxTastiness computation 43 Java Solution 1 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 6 public int maxTastiness(int[] prices, int[] tastinessValues, int maxAmount, int maxCoupons) { itemCount = prices.length; this.prices = prices; 10 this.tastinessValues = tastinessValues; memo = new int[itemCount][maxAmount + 1][maxCoupons + 1]; 11 12 // Call the recursive function starting at the first item, with full budget, and all coupons available

maxValue = Math.max(maxValue, dfs(itemIndex + 1, remainingAmount - prices[itemIndex], remainingCoupons) + tastinessValues

maxValue = Math.max(maxValue, dfs(itemIndex + 1, remainingAmount - prices[itemIndex] / 2, remainingCoupons - 1) + tastine

C++ Solution

#include <vector>

2 #include <cstring>

class Solution {

public:

3 #include <functional>

```
int n = prices.size();
  9
 10
             // Initialize the memoization array with 0 values.
 11
             memset(memo, 0, sizeof memo);
 12
 13
             // Define the dfs (Depth-First Search) as a lambda function that can call itself.
 14
             // i: current item index
             // currentAmount: current amount of money left
 15
             // currentCoupons: current number of coupons left
 16
             function<int(int, int, int)> dfs = [&](int i, int currentAmount, int currentCoupons) {
 17
                 // Base case: All items have been considered.
 18
 19
                 if (i == n) return 0;
 20
                 // If this state has already been calculated, return the stored value.
 21
 22
                 if (memo[i][currentAmount][currentCoupons]) return memo[i][currentAmount][currentCoupons];
 23
 24
                 // Option 1: Do not buy the current item.
                 int maxTaste = dfs(i + 1, currentAmount, currentCoupons);
 25
 26
 27
                 // Option 2: Buy the current item without using a coupon.
 28
                 if (currentAmount >= prices[i]) {
 29
                     maxTaste = max(maxTaste, dfs(i + 1, currentAmount - prices[i], currentCoupons) +
 30
                                                  tastinessValues[i]);
 31
 32
                 // Option 3: Buy the current item using a coupon, if available.
 33
                 if (currentAmount >= prices[i] / 2 && currentCoupons) {
 34
 35
                     maxTaste = max(maxTaste, dfs(i + 1, currentAmount - prices[i] / 2, currentCoupons - 1) +
 36
                                                  tastinessValues[i]);
 37
 38
 39
                 // Store the calculated value and return it.
                 memo[i][currentAmount][currentCoupons] = maxTaste;
 40
 41
                 return maxTaste;
 42
             };
 43
             // Start the dfs from the first item, with the maximum amount and coupons available.
 45
             return dfs(0, maxAmount, maxCoupons);
 46
 47
    private:
        // Memoization array to store previously computed states:
 49
 50
        // f[item index][current amount][current coupons left]
 51
         int memo[101][1001][6];
 52 };
 53
Typescript Solution
  1 // Define a type for the memoization array.
  2 type MemoizationArray = number[][][];
  4 // Define global variables for memoization, prices, tastinessValues, and the dimensions of our memo array.
```

#### 34 35 36 // Option 3: Buy the current item using a coupon, if available. 37 if (currentAmount >= Math.floor(prices[i] / 2) && currentCoupons > 0) { 38 maxTaste = Math.max(maxTaste, dfs(i + 1, currentAmount - Math.floor(prices[i] / 2), currentCoupons - 1) + tastinessValues[i 39

5 let memo: MemoizationArray;

7 let tastinessValues: number[];

memo = Array.from({ length: n }, () =>

if (i === prices.length) return 0;

if (currentAmount >= prices[i]) {

11 // Initialize a 3D memo array with all values set to -1 (unvisited state).

Array.from({ length: maxAmount + 1 }, () =>

if (memo[i][currentAmount][currentCoupons] !== -1) {

return memo[i][currentAmount][currentCoupons];

let maxTaste = dfs(i + 1, currentAmount, currentCoupons);

// Option 2: Buy the current item without using a coupon.

Array(maxCoupons + 1).fill(-1)));

// Base case: All items have been considered.

// Option 1: Do not buy the current item.

function initializeMemo(n: number, maxAmount: number, maxCoupons: number): void {

// Recursive function to calculate the max tastiness given the current state.

function dfs(i: number, currentAmount: number, currentCoupons: number): number {

// If this state has already been calculated, return the stored value.

6 let prices: number[];

8 let maxAmount: number;

10

13

14

15

16

17

20

21

22

23

24

25

26

27

28

29

30

31

32

33

40

let maxCoupons: number;

```
41
         // Store the calculated value in the memo array and return it.
 42
        memo[i][currentAmount][currentCoupons] = maxTaste;
 43
         return maxTaste;
 44 }
 45
    // Wrapper function to start the recursive calculation.
     function maxTastiness(inputPrices: number[], inputTastinessValues: number[], inputMaxAmount: number, inputMaxCoupons: number): numb
         prices = inputPrices;
 48
         tastinessValues = inputTastinessValues;
 49
 50
         maxAmount = inputMaxAmount;
 51
        maxCoupons = inputMaxCoupons;
 52
 53
        // Initialize the memo array with proper dimensions.
 54
         initializeMemo(prices.length, maxAmount, maxCoupons);
 55
 56
         // Start the depth-first search from the first item, with the maximum amount and coupons available.
 57
         return dfs(0, maxAmount, maxCoupons);
 58 }
 59
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
```

maxTaste = Math.max(maxTaste, dfs(i + 1, currentAmount - prices[i], currentCoupons) + tastinessValues[i]);

## The recursion is memoized to ensure that each state is computed at most once. Therefore, the time complexity is 0(N\*M\*C). **Space Complexity**

coupons k.

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

k from [0, C], the number of possible states is roughly N \* M \* C.

1. The memoization cache, which needs to store the result for each unique state (i, j, k), thus requiring a space complexity of 0(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.

function dfs is called with different states represented by a combination of current item index i, remaining amount j, and remaining

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).

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

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.

Thus, the overall space complexity of the code is also 0(N\*M\*C) due to the cache size dominating the recursive call stack.