2706. Buy Two Chocolates

Easy Array Sorting

Leetcode Link

Problem Description

In this problem, you have an array `prices` that consists of the prices of chocolates available in a store. Additionally, you have an integer `money` that represents the amount of money you initially have. Your goal is to purchase exactly two chocolates. The two chocolates you choose to buy must cost less than or equal to the `money` you have. The objective is not just to find any two chocolates that you can afford but to select two such that you minimize the sum of their prices. By doing this, you maximize the amount of money you will have left after the purchase. The problem asks you to return the amount of money that will remain after buying these two chocolates. If it is not possible to buy two chocolates without spending more money than you have (that is, without going into debt), you need to return the original amount of money.

Intuition

The intuition behind the solution is to first sort the array of `prices` in ascending order. Sorting ensures that the chocolates with the lowest prices come first. To minimize the sum of the prices of the two chocolates, you simply need to choose the first two chocolates in the sorted array because they will be the cheapest.

After sorting, the algorithm checks the sum of the prices of the first two chocolates and compares it with the `money` you have. If the sum is less than or equal to `money`, it means you can afford these two chocolates and thus you should return the leftover amount, which is `money - cost` of the two chocolates. However, if the sum is greater than `money`, you cannot afford any pair of chocolates without going into debt, so the function should return the initial amount of money without any purchase, which is simply `money`.

In summary, the solution hinges on the fact that sorting the `prices` array helps identify the minimum cost for buying two chocolates. This turns the problem into a simple comparison and subtraction operation.

Solution Approach

The implementation of the solution can be dissected into few simple steps:

1. **Sorting**: First, we sort the `prices` array using the sort method. In Python, this is done using the `sort()` function. Sorting is an essential step here because it uses the built-in efficient sorting algorithms like Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort) in Python. Through sorting, we can ensure that we are considering the cheapest chocolates first.

```
prices.sort()
```

2. **Calculating Minimum Cost**: Once the array is sorted, the prices of the two cheapest chocolates can be accessed with `prices[0]` and `prices[1]`. The sum of these two prices gives us the minimum cost needed to purchase two chocolates.

```
cost = prices[0] + prices[1]
```

3. **Checking Affordability and Computing Leftover Money**: The if-else statement is used to check if you can afford to buy the two chocolates without going into debt. This is done by comparing the `money` with the `cost` of buying the two cheapest chocolates.

   - If `money` is less than `cost`, you cannot afford any two chocolates and should return the original amount of `money`.
   - If `money` is greater than or equal to `cost`, it means you can afford the chocolates, so you return the `money` left after the purchase, i.e., `money - cost`.

```
return money if money < cost else money - cost
```

This approach leverages the efficiency of Python's sorting function and simple arithmetic operations to achieve the solution. The time complexity of this solution is dominated by the sorting step, which is $O(n \log n)$ where n is the number of prices in the array. The space complexity is $O(1)$ as no additional space is used apart from the input and variables to store the cost and the final answer.

Example Walkthrough

Let's say we have an array `prices = [5, 3, 20, 8]` and `money = 10`. We want to buy exactly two chocolates within the `money` we have, and maximize the amount left after the purchase. Now let's go through the solution steps with this example:

1. **Sorting**: First, we sort the array `prices`. After sorting, the array becomes `prices = [3, 5, 8, 20]`.

```
prices.sort()  # prices becomes [3, 5, 8, 20]
```

2. **Calculating Minimum Cost**: We then calculate the cost of buying the two cheapest chocolates, which are now `prices[0]` and `prices[1]`, i.e., `3` and `5`.

```
cost = prices[0] + prices[1]  # cost becomes 3 + 5 = 8
```

3. **Checking Affordability and Computing Leftover Money**: We check if the `money` we have is enough to cover the `cost`.

   - We compare `money` (10) with `cost` (8). Since `10` is greater than `8`, we can afford the chocolates.
   - We then calculate the money left after the purchase, which is `money - cost`.

```
leftover_money = money - cost  # leftover_money becomes 10 - 8 = 2
```

Therefore, with `prices = [5, 3, 20, 8]` and `money = 10`, after purchasing the two chocolates costing `3` and `5`, we are left with `2` as the answer.

Using this walkthrough as a guide, it's clear that the solution approach effectively uses sorting to minimize the purchase cost and an if-else logic to ensure we don't overspend. The result, `2` in this case, represents the optimal amount of money remaining after making a legitimate purchase of two chocolates.

Python Solution

```python
# The List type needs to be imported from typing to be used as a type hint.
from typing import List

class Solution:
    def buyChoco(self, prices: List[int], money: int) -> int:
        # Sort the prices in non-decreasing order to find the cheapest chocolates.
        prices.sort()

        # We check if there are at least two chocolates to buy,
        # as the customer should buy at least two chocolates to get the discount.
        if len(prices) < 2:
            # If we have less than two chocolates, we can't calculate a combined cost
            # so the customer can't spend the money on chocolates as intended.
            return 0

        # Calculate the combined cost of the two cheapest chocolates.
        cost = prices[0] + prices[1]

        # If the customer doesn't have enough money to buy the two cheapest chocolates,
        # they can't make a purchase, so return 0.
        if money < cost:
            return 0

        # Otherwise, deduct the combined cost from the customer's money
        # and return the remaining amount.
        return money - cost

```

Java Solution

```java
import java.util.Arrays; // Import Arrays class for sorting

class Solution {

    // Method to calculate how much money is left after buying two cheapest chocolates
    public int buyChoco(int[] prices, int money) {
        // Sort the array to get the prices in ascending order
        Arrays.sort(prices);

        // Calculate the cost of the two cheapest chocolates
        int costOfTwoCheapest = prices[0] + prices[1];

        // If the money is less than the cost of two chocolates, return the original amount of money
        // since you can't afford to buy them. Otherwise, return the remaining money after purchase.
        return money < costOfTwoCheapest ? money : money - costOfTwoCheapest;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for using the sort function

class Solution {
public:
    // This function calculates the remaining money after buying the two cheapest chocolates.
    int buyChoco(vector<int>& prices, int money) {
        // Sort the prices of the chocolates in ascending order
        sort(prices.begin(), prices.end());

        // Check if we have at least two chocolate prices
        if (prices.size() < 2) {
            // If not, we can't buy two chocolates, so return the original amount of money.
            return money;
        }

        // Calculate the total cost of buying the two cheapest chocolates
        int totalCost = prices[0] + prices[1];

        // If we have enough money for at least the two cheapest chocolates, return the remaining money
        // Otherwise, return the original amount as we can't buy those chocolates
        return (money >= totalCost) ? (money - totalCost) : money;
    }
};

```

Typescript Solution

```typescript
// Defines a function to determine how much money will be left after buying the two cheapest chocolates
// prices: Array of numbers representing the prices of different chocolates
// money: The total amount of money available to spend
function buyChoco(prices: number[], money: number): number {
    // Sort the prices array in ascending order to identify the two cheapest chocolates
    prices.sort((a, b) => a - b);

    // Calculate the total cost of the two cheapest chocolates
    const totalCostOfCheapestTwo: number = prices[0] + prices[1];

    // If the total money is less than the cost of the two cheapest chocolates,
    // return the original amount of money since no purchase can be made.
    // Otherwise, return the remaining money after the purchase is made.
    return money < totalCostOfCheapestTwo ? money : money - totalCostOfCheapestTwo;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function `buyChoco` is determined primarily by the sorting operation. The sorting function in Python, `.sort()`, typically uses Timsort, which has an average and worst-case time complexity of `O(n log n)` where `n` is the number of elements in the list `prices`.

Since the rest of the operations after sorting (accessing the first two elements and basic arithmetic operations) are constant time operations, i.e., `O(1)`, they don't significantly contribute to the overall time complexity.

Thus, the total time complexity of the function `buyChoco` is `O(n log n)`.

Space Complexity

The space complexity of the function `buyChoco` is determined by the additional space required for the operation of the code.

Since the sorting operation is done in place with the `.sort()` method, it does not require any additional space proportional to the input (it uses only a constant amount of extra space).

Therefore, the space complexity is `O(1)`, which is constant space complexity, since no additional space is allocated that scales with the input size.