1833. Maximum Ice Cream Bars

Medium Greedy Array Sorting

Problem Description

On a hot summer day, a boy is looking to buy some ice cream bars from a store. The store has `n` different types of ice cream bars, each with its own price. The prices are given in an array `costs` with `n` elements, where `costs[i]` represents the price of the `i-th` ice cream bar in coins. The boy has a limited amount of money, specifically `coins` coins, which he will use to purchase as many ice cream bars as he can.

The key point here is that the boy is allowed to buy the ice cream bars in any order he prefers. The main challenge is to determine the maximum number of ice cream bars the boy can buy without exceeding the number of coins he has.

To solve the problem, we need to formulate an approach that ensures we can get the maximum number of bars under the given constraints.

Intuition

Considering that we aim to maximize the number of ice cream bars the boy can buy, it makes sense to start by purchasing the cheapest available options. This strategy ensures that we get the most out of the coins available to us.

We follow a Greedy approach to solve this problem. The Greedy algorithm is a straightforward, intuitive method where we always choose the next most favorable option, hoping that this will lead to an optimal overall solution. In this case, the most favorable option is to buy the cheapest ice cream bar first.

To implement this Greedy approach efficiently, we sort the `costs` array in ascending order. Then, we iterate through this sorted list and keep subtracting the cost of each ice cream bar from the total coins we have until we can't afford to buy the next bar.

At each step, we're making the optimal local decision to buy the cheapest available ice cream bar, and as a result, we end up with the maximum number of bars that can be bought with the coins available.

Solution Approach

The solution follows a Greedy algorithm, which is perfect for this scenario since we want to buy as many items as possible given a constraint (the coins).

Here's a breakdown of how the algorithm is implemented, referring to the given solution approach:

1. **Sort the `costs` Array**: The first step is to sort the array of ice cream costs. This is essential because it allows us to start with the least expensive options, ensuring we can buy as many as possible.

   ```
   costs.sort()
   ```

   This operation uses an `O(n log n)` sorting algorithm where `n` is the number of ice cream bars (`n` being the length of `costs`).

2. **Iterate through the Sorted `costs`**: Once we have the `costs` array sorted, we iterate through it to simulate purchasing the ice cream bars from the cheapest to the most expensive.

   ```
   for i, c in enumerate(costs):
   ```

   We use a `for` loop combined with `enumerate` to go through each price (`c`) along with its index (`i`) in the sorted `costs` array.

3. **Check if the Next Ice Cream Can Be Bought**: For each price in the array, we check if we have enough coins to buy the ice cream at the current price. If we do not have enough, we simply return the number of ice cream bars bought so far, which would correspond to the index `i`.

   ```
   if coins < c:
       return i
   ```

   If the ice cream bar can be purchased, we subtract its cost from our total coins.

   ```
   coins -= c
   ```

   We repeat this step until we run out of coins or have bought all the ice cream bars.

4. **Return the Total Bars Bought**: If we break out of the loop because we've run out of coins, we will have already returned the number of ice cream bars bought. If we complete the loop without running out of coins, it means we could buy all available ice cream bars. Therefore, we return the total number of ice cream bars (`len(costs)`).

   ```
   return len(costs)
   ```

The use of sorting, followed by a simple linear scan of the array, allows this solution to be effective and to the point. This algorithm's efficiency relies heavily on the sorting step, which is why sorting algorithms are such a critical part of most Greedy approaches to problem-solving.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have an array of ice cream costs `costs = [1, 3, 2, 4, 1]` and the boy has `coins = 7`.

Here's how we apply the Greedy algorithm to maximize the number of ice cream bars the boy can buy:

1. **Sort the `costs` Array**: First, we sort the `costs` array. After sorting, we get `[1, 1, 2, 3, 4]`. This sorting will allow us to consider the cheapest ice cream bars first.
2. **Iterate through the Sorted `costs`**: We start iterating from the first element of the sorted array.

   For `i = 0` (first ice cream bar), `c = 1`. We check if the boy can afford to buy this ice cream bar. Since `coins = 7` and `c = 1`, he can afford it. Thus, we subtract the cost of this ice cream bar from his coins: `coins = 7 - 1 = 6`.

   For `i = 1` (second ice cream bar), `c = 1`. He can still afford it, so again we subtract the cost: `coins = 6 - 1 = 5`.

   For `i = 2` (third ice cream bar), `c = 2`. He can afford it, so we subtract the cost: `coins = 5 - 2 = 3`.

   For `i = 3` (fourth ice cream bar), `c = 3`. He can afford it, so we subtract the cost: `coins = 3 - 3 = 0`.

   For `i = 4` (fifth ice cream bar), `c = 4`. Now the boy has `coins = 0` and cannot afford to buy this ice cream bar.
3. **Check if Next Ice Cream Can Be Bought & Return Total Bars Bought**: As soon as the boy can no longer afford an ice cream bar, we return the number he has bought so far. In this case, after buying the first four ice cream bars, he's left with zero coins and is unable to buy the fifth one that costs 4 coins.

Therefore, the maximum number of ice cream bars the boy can buy is `4`.

Solution Implementation

Python

```python
class Solution:
    def maxIceCream(self, ice_cream_costs: List[int], total_coins: int) -> int:
        # Sort the ice cream costs in non-decreasing order
        ice_cream_costs.sort()

        # Initialize a variable to count the number of ice creams bought
        ice_creams_bought = 0

        # Iterate over the sorted ice cream costs
        for cost in ice_cream_costs:
            # If the current ice cream cost is more than the total coins,
            # we cannot buy this ice cream, so we return the count.
            if total_coins < cost:
                return ice_creams_bought
            # Otherwise, we can buy this ice cream
            # so we subtract its cost from the total coins
            total_coins -= cost
            # and increment the count of ice creams bought
            ice_creams_bought += 1

        # If we were able to buy all the ice creams without running out of coins,
        # return the total number of ice creams bought,
        # which is the length of the costs list
        return ice_creams_bought
```

Java

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

class Solution {

    // Method for calculating maximum number of ice creams that can be bought with the given coins
    public int maxIceCream(int[] costs, int coins) {
        // Sort the array to purchase cheaper ice creams first
        Arrays.sort(costs);

        // Get the number of ice cream prices in the array
        int numOfIceCreams = costs.length;

        // Initialize the count of ice creams bought
        int count = 0;

        // Iterate over the sorted array
        for (int i = 0; i < numOfIceCreams; ++i) {
            // If the current ice cream cost is more than the available coins
            // Return the count of ice creams bought so far
            if (coins < costs[i]) {
                return count;
            }

            // Subtract the cost of the current ice cream from the available coins
            coins -= costs[i];

            // Increment the count of ice creams bought
            count++;
        }

        // Return the total count of ice creams that can be bought
        // This line is reached when all ice creams can be bought with the available coins
        return count;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to calculate the maximum number of ice creams that can be bought with `coins`.
    int maxIceCream(vector<int>& costs, int coins) {
        // Sort the `costs` vector in non-decreasing order.
        std::sort(costs.begin(), costs.end());

        // Initialize the counter for the number of ice creams.
        int numIceCreams = 0;

        // Iterate through the sorted ice cream costs.
        for (int cost : costs) {
            // If we don't have enough coins to buy the current ice cream, break the loop.
            if (coins < cost) break;

            // Subtract the cost from our coins and increment the count of ice creams.
            coins -= cost;
            numIceCreams++;
        }

        // Return the number of ice creams purchased.
        return numIceCreams;
    }
};
```

TypeScript

```typescript
/**
 * Function to calculate the maximum number of ice creams that can be bought
 * with a given amount of coins.
 * @param {number[]} iceCreamCosts - The array of costs of different ice creams.
 * @param {number} totalCoins - The total number of coins you have.
 * @returns {number} The maximum number of ice creams that can be bought.
 */
function maxIceCream(iceCreamCosts: number[], totalCoins: number): number {
    // Sort the ice cream costs in ascending order
    iceCreamCosts.sort((a, b) => a - b);

    // The number of different ice creams available
    const numberOfIceCreams = iceCreamCosts.length;

    // Loop through each ice cream cost
    for (let i = 0; i < numberOfIceCreams; ++i) {
        // If the current ice cream cost is more than the remaining coins, return the count of ice creams bought so far
        if (totalCoins < iceCreamCosts[i]) {
            return i;
        }
        // Subtract the cost of the current ice cream from the total coins
        totalCoins -= iceCreamCosts[i];
    }

    // If all ice creams could be bought with the available coins, return the total number of ice creams
    return numberOfIceCreams;
}
```


```python
class Solution:
    def maxIceCream(self, ice_cream_costs: List[int], total_coins: int) -> int:
        # Sort the ice cream costs in non-decreasing order
        ice_cream_costs.sort()

        # Initialize a variable to count the number of ice creams bought
        ice_creams_bought = 0

        # Iterate over the sorted ice cream costs
        for cost in ice_cream_costs:
            # If the current ice cream cost is more than the total coins,
            # we cannot buy this ice cream, so we return the count.
            if total_coins < cost:
                return ice_creams_bought
            # Otherwise, we can buy this ice cream
            # so we subtract its cost from the total coins
            total_coins -= cost
            # and increment the count of ice creams bought
            ice_creams_bought += 1

        # If we were able to buy all the ice creams without running out of coins,
        # return the total number of ice creams bought,
        # which is the length of the costs list
        return ice_creams_bought
```


Time and Space Complexity

The time complexity of the provided code is `O(n log n)`. This complexity arises from the fact that the main operation in this algorithm is sorting the `costs` array, which is typically implemented with an algorithm like quicksort or mergesort that has an `O(n log n)` complexity in Python’s sort function.

After sorting, the code iterates through the sorted `costs` array only once, with each iteration being a constant time operation. However, since this linear scan (`O(n)`) is dominated by the sorting step, it does not change the overall time complexity of the algorithm.

The space complexity of the code is `O(log n)`. This is because the space complexity is attributed to the space that is used during the sorting of the array. Most sorting algorithms like quicksort and mergesort typically require `O(log n)` space on the call stack for recursive function calls.