2144. Minimum Cost of Buying Candies With Discount

Easy Greedy Array Sorting

Leetcode Link

Problem Description

In this problem, we are working with a candy shop that has a particular discount scheme: for every two candies purchased, a third candy is given for free. The caveat is that the cost of the free candy can't exceed the cost of the least expensive candy bought among the two. We need to find out how much the customer will have to pay in total to buy all the candies available in the shop considering this discount.

The given data is an integer array `cost` which represents the price of each candy. The array is 0-indexed, meaning that the first element is at index 0, the second at index 1, and so on. Our task is to calculate the minimum total cost required to buy all candies while making the most out of the given discount.

Intuition

To minimize the total cost of buying all the candies while taking advantage of the discount, we should aim to get the more expensive candies for free when possible. Hence, the best strategy would be to sort the candies in descending order of their cost. That way, every time we pick two candies, they will be the most expensive ones available, and we can then choose one of the less expensive candies for free (following the rule that the free candy must be priced less than or equal to the cheaper of the two paid candies).

After sorting, we can observe that in every three consecutive candies starting from the first (index 0), the third one (at index 2) would be the free one if we adopt the purchasing pattern of buying in sets of three, so to speak. We continue this pattern throughout the array. Thus, by taking the sum of all candies' costs and subtracting the sum of every third candy post-sorting, we get the total minimum cost. This is exactly what the provided solution is doing: it sorts the list in reverse order, then sums up all elements, excluding every third element starting by the third (index 2, as the index is zero-based). That is the sum of all the costs of the free candies if we buy them in an optimal way.

Solution Approach

The solution approach takes advantage of sorting and list slicing operations. Here's a step-by-step explanation of the implementation:

1. **Sorting the list in reverse order**: We use the `sort` method with the `reverse=True` parameter to rearrange the elements of the `cost` list in descending order. Sorting is a common algorithmic pattern when we need to rearrange data for optimization purposes, such as minimizing or maximizing some value subject to constraints—in this case, the constraint is the "buy two, get one free" deal.

   ```
   cost.sort(reverse=True)
   ```

   After this operation, the most expensive candies are at the beginning of the list, and the least expensive ones are at the end.

2. **Calculating the total sum**: The `sum` function calculates the total cost of the candies if there were no discounts. This serves as our starting point for finding the minimum cost.

   ```
   total_cost = sum(cost)
   ```

3. **Identifying the free candies using slicing**: The third step uses list slicing to identify all the candies that would be free. The slice operation `cost[2::3]` takes every third candy from the list, starting with the one at index 2 (zero-based index, which is the third candy). This exploits a pattern that emerges from sorting the candies by cost: when they are sorted in descending order, each set of three contains two paid candies followed by one free candy.

   ```
   free_candies_cost = sum(cost[2::3])
   ```

4. **Subtracting the cost of free candies**: By subtracting the sum of the free candies' costs from the total cost, we arrive at the minimum cost to buy all candies. This effectively applies the "buy two, get one free" discount across the entire list.

   ```
   minimum_cost = total_cost - free_candies_cost
   ```

Putting it all together, the provided solution is a compact representation of these steps, combining the total calculation and the subtraction of the free candies' costs into a single line:

```
return sum(cost) - sum(cost[2::3])
```

This approach is both elegant and efficient, as it avoids the need for explicit loops or additional data structures. The sorting algorithm behind the `sort` method typically has O(n log n) complexity, where `n` is the number of candies, and the slicing operation has O(n) complexity. Thus, the overall complexity of the solution is dominated by the sorting step.

Example Walkthrough

Let's suppose we have the following candy prices:

```
cost = [1, 3, 2, 8, 5, 7]
```

To understand the solution approach, let's walk through it step by step using this example:

1. **Sorting the list in reverse order**: We sort the `cost` list in descending order to prioritize the purchase of the most expensive candies first.

   ```
   cost.sort(reverse=True)  // After sorting: cost = [8, 7, 5, 3, 2, 1]
   ```

2. **Calculating the total sum**: Before applying the discount, calculate the total cost of all candies.

   ```
   total_cost = sum(cost)  // total_cost = 8 + 7 + 5 + 3 + 2 + 1 = 26
   ```

3. **Identifying the free candies using slicing**: According to the discount, for every two candies purchased, the third one is free, starting from the least expensive of the paid ones. To find the free candies applying the discount optimally, we take every third candy in the sorted list, beginning with the one at index 2.

   ```
   free_candies_cost = sum(cost[2::3])  // The free candies are at indices 2 and 5, so free_candies_cost = 5 + 1 = 6
   ```

4. **Subtracting the cost of free candies**: Finally, to get the minimum cost after the discount, subtract the total cost of free candies from the total cost.

   ```
   minimum_cost = total_cost - free_candies_cost  // minimum_cost = 26 - 6 = 20
   ```

So, the minimum total cost to buy all candies while taking full advantage of the discounts is 20.

Using our approach, we've paired the most expensive candies first ([8, 7], [5, 3]), then we get the least expensive candies as the free items ([2] and [1]). The 2 and 1 correspond to the third items in each buying set, which are the free items according to our sorted `cost` list.

By following these steps, the provided solution effectively minimizes the total cost, ensuring that customers get the highest-priced candies possible for free, in compliance with the shop's discount policy.

Python Solution

```python
class Solution:
    def minimumCost(self, costs: List[int]) -> int:
        # Sort the list of costs in descending order
        costs.sort(reverse=True)

        # Initialize the minimum cost to the sum of all costs
        min_cost = sum(costs)

        # Subtract the cost of every third item starting from the third item
        # because it would be the free item as per the typical "buy two, get one free" offer.
        min_cost -= sum(costs[2::3])

        # Return the calculated minimum cost
        return min_cost

```

Java Solution

```java
class Solution {

    /**
     * Calculates the minimum cost to purchase all the items given that
     * every third item is free after sorting the array in non-decreasing order.
     *
     * @param costs Array of individual costs of items.
     * @return The minimum total cost to purchase all items.
     */
    public int minimumCost(int[] costs) {
        // Sort the array of costs in non-decreasing order.
        Arrays.sort(costs);

        int totalCost = 0; // Initialize total cost to 0.
        // Start purchasing items from the most expensive one.
        for (int i = costs.length - 1; i >= 0; i -= 3) {
            totalCost += costs[i]; // Purchase the most expensive item.
            if (i > 0) { // Check if there is a second item to purchase.
                totalCost += costs[i - 1]; // Purchase the second item.
            }
        }
        return totalCost; // Return the total minimum cost.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Included for std::sort

class Solution {
public:
    /**
    * Calculate the minimum cost of buying items based on the offer to pay for two
    * items and get one free, optimizing to always get the most expensive item for free.
    *
    * @param costs Vector of individual item costs.
    * @return The minimum total cost with the offer applied.
    */
    int minimumCost(vector<int>& costs) {
        // Sort the costs in descending order to prioritize the most expensive items.
        std::sort(costs.rbegin(), costs.rend());

        int totalCost = 0; // Initialize total cost to 0

        // Iterate over every three items to apply the "buy two get one free" offer
        for (int i = 0; i < costs.size(); i += 3) {
            totalCost += costs[i]; // Always add the cost of the first item (most expensive)

            // Check if there is a second item to also add its cost
            if (i + 1 < costs.size()) {
                totalCost += costs[i + 1]; // Add the cost of the second item
            }

            // The third item is free, so it is not added to the total
        }

        // Return the calculated minimum total cost
        return totalCost;
    }
};

```

Typescript Solution

```typescript
function minimumCost(cost: number[]): number {
    // Sort the array in ascending order to ensure that the minimum values are skipped
    cost.sort((a, b) => a - b);

    let totalCost = 0; // Initialize the total cost variable

    // Start iterating from the end of array towards the front
    // We jump three steps at a time since every third flower is free
    for (let i = cost.length - 1; i >= 0; i -= 3) {
        totalCost += cost[i]; // Add the cost of the current flower
        // If there is another flower before the current one, add that cost too
        if (i > 0) {
            totalCost += cost[i - 1];
        }
    }

    // Return the total cost after applying the discount for every third flower
    return totalCost;
}

```

Time and Space Complexity

The given Python code aims to calculate the minimum cost of purchasing items given that every third item is free when items are purchased in descending order of cost. We will analyze the time and space complexity of the provided code.

Time Complexity:

1. The `sort` function on the list `cost` has a time complexity of `O(n log n)` where `n` is the number of elements in the list `cost`. This is because Python's Timsort algorithm is used for sorting, which has this time complexity.
2. The `sum` function iterates over the entire list, resulting in `O(n)` time complexity.
3. The slicing operation `cost[2::3]` generates a new list containing every third element from the list `cost`, leading to `O(n/3)` complexity, which simplifies to `O(n)`.

Combining these operations, the dominant term is the sorting operation. Thus, the total time complexity of the code is `O(n log n)`.

Space Complexity:

1. The sorting is done in-place, so there is no additional space required for this operation.
2. The `sum` function operates on the list without creating new structures, so it uses `O(1)` space.
3. The slicing operation `cost[2::3]` creates a new list, which in the worst case, could be `O(n/3)` space complexity. However, since constants are dropped in Big O notation, it simplifies to `O(n)` space complexity.

Therefore, the space complexity of the code is `O(n)`, accounting for the space needed to store the new list generated by the slicing operation.