1196. How Many Apples Can You Put into the Basket

Easy Greedy Array Sorting

Leetcode Link

Problem Description

The problem involves a basket with a maximum weight limit of `5000` units and a list of integers representing the weights of several apples. The task is to find the maximum number of apples that can be placed in the basket without exceeding the weight limit. This is an optimization problem where you must maximize the quantity of items selected under a certain constraint, in this case, the weight capacity of the basket.

Intuition

The intuition behind the solution is to use a greedy algorithm approach. It makes intuitive sense to start by adding the lightest apples to the basket first, as this would allow us to fit more apples into the basket. By sorting the weights of the apples, we ensure we always consider the lightest apple available for placement in the basket next.

Once we have the sorted list, we iterate through the list, keeping track of the total weight added to the basket so far. We continue to add apples one at a time until adding another apple would exceed the weight limit of `5000` units. When we reach an apple that we cannot add without exceeding the limit, we stop and return the number of apples we have added up to that point.

Solution Approach

The solution follows a straightforward approach aligned with greedy algorithms, which often provide an optimal solution to optimization problems under certain conditions. Here is how it's implemented:

1. **Sorting**: We sort the `weight` array to ensure that we pick apples starting from the lightest to the heaviest. Sorting is a common preparatory step in greedy algorithms, facilitating a sequential choice of items based on some criteria—in this case, the weight.
2. **Iteration with Accumulation**: We initialize a variable `s` to zero, representing the current total weight of apples in the basket.
3. **Iterative Weight Addition**: Sequentially, we iterate over the sorted apple weights. For each apple, we add its weight to the current total `s`. As we do this, we check whether the running total exceeds the maximum allowed weight of `5000` units.
4. **Condition Checking**: If the addition of the current apple's weight causes the total weight `s` to exceed the limit, the loop breaks, and we return the number of apples that have been successfully added so far, which is given by the current index `i`.
5. **Returning the Result**: If we complete the iteration without exceeding the weight limit, it means all apples in the list have been added to the basket. Hence, we return the total count of apples, which can be obtained using `len(weight)`.

Using this method, the algorithm reaches an efficient solution that maximizes the number of apples within the allowed weight limit. The use of sorting followed by a linear scan of the list is characteristic of many greedy strategies and is effectively employed in this implementation.

Example Walkthrough

To illustrate the solution approach, let's consider a small example. Suppose we have the following weights for the apples: `[600, 1000, 950, 200, 400]`. We aim to place as many of these apples into a basket with a weight limit of 5000 units.

1. **Sorting**: First, we sort the apples by their weight, resulting in the sorted list `[200, 400, 600, 950, 1000]`.
2. **Iteration with Accumulation**: We start with a total current weight `s` of zero and begin iterating over the list.
3. **Iterative Weight Addition**: As we scan the list, we add the weight of each apple to `s`:
   - Add 200 (s = 200)
   - Add 400 (s = 600)
   - Add 600 (s = 1200)
   - Add 950 (s = 2150)
   - Add 1000 (s = 3150)

   We successfully add all the apples in the sorted list to the basket since the total weight does not exceed 5000 units.
4. **Condition Checking**: At each step of adding an apple, we check if the next addition will exceed the weight limit. In this case, we never reach a point where `s` exceeds 5000, so the condition check always passes.
5. **Returning the Result**: Since we were able to add all apples without surpassing the weight limit, we return the total number of apples, which is 5.

Therefore, using our greedy approach, we concluded that we can fit all five apples in the basket because their combined weight is 3150 units, which is within the 5000-unit limit.

Python Solution

```python
class Solution:
    def maxNumberOfApples(self, arr: List[int]) -> int:
        # Sort the weights of the apples in increasing order.
        arr.sort()

        # Initialize the total weight accumulator.
        total_weight = 0

        # Iterate over the sorted apple weights.
        for index, weight in enumerate(arr):
            # Add the weight of the current apple to the total weight.
            total_weight += weight

            # Check if the total weight exceeds 5000 grams.
            if total_weight > 5000:
                # If so, return the current number of apples.
                return index

        # If the total weight never exceeds 5000, return the total number of apples.
        return len(arr)

```

Java Solution

```java
class Solution {
    public int maxNumberOfApples(int[] weights) {
        // Sort the array of apple weights in non-decreasing order
        Arrays.sort(weights);

        // Initialize the total weight of apples so far to 0
        int totalWeight = 0;

        // Loop through the sorted apple weights
        for (int i = 0; i < weights.length; ++i) {
            // Add the current apple's weight to the total weight
            totalWeight += weights[i];

            // Check if the total weight exceeds 5000 grams
            if (totalWeight > 5000) {
                // If so, return the current number of apples
                return i;
            }
        }

        // If the total weight never exceeds 5000 grams, return the total number of apples
        return weights.length;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to calculate the maximum number of apples with a weight limit.
    int maxNumberOfApples(vector<int>& appleWeights) {
        // Sort the apple weights in non-decreasing order
        sort(appleWeights.begin(), appleWeights.end());

        int currentWeightSum = 0; // To keep track of the current total weight

        // Iterate through the sorted weights of the apples
        for (int i = 0; i < appleWeights.size(); ++i) {
            // Add the weight of the current apple to the total weight
            currentWeightSum += appleWeights[i];

            // Check if the weight limit is exceeded
            if (currentWeightSum > 5000) {
                // Return the maximum number of apples before exceeding the weight limit
                return i;
            }
        }

        // If the weight limit is not exceeded, return the total count of apples
        return appleWeights.size();
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the maximum number of apples that can fit within a weight limit.
 * Assumes each apple's weight is given in an array, with a weight limit of 5000.
 * @param appleWeights Array of individual apple weights.
 * @returns The maximum number of apples that can be included without exceeding the weight limit.
 */
function maxNumberOfApples(appleWeights: number[]): number {
    // Sort the array of apple weights in ascending order to pack lighter apples first.
    appleWeights.sort((a, b) => a - b);

    // Initialize the sum of apple weights.
    let totalWeight = 0;

    // Loop through the sorted apple weights.
    for (let i = 0; i < appleWeights.length; ++i) {
        // Add the weight of the current apple to the total weight sum.
        totalWeight += appleWeights[i];

        // If the weight limit is exceeded with the current apple, return the number of apples before this one.
        if (totalWeight > 5000) {
            return i;
        }
    }

    // If the total weight never exceeds the limit, return the total number of apples.
    return appleWeights.length;
}

```

Time and Space Complexity

The given Python code snippet sorts the input list `weight` and then iterates through the sorted list to calculate the cumulative weight until it exceeds the limit of 5000, or until all the apples are counted.

Time Complexity:

The time complexity of the algorithm is dominated by the sorting operation at the beginning. Sorting a list of `n` elements typically has a time complexity of `O(n log n)` when using the built-in sorted function in Python (Timsort algorithm). After sorting, there is a single pass over the input list to calculate the cumulative weight. The single pass has a time complexity of `O(n)`, where `n` is the number of elements in `weight`.

Since the sorting cost is higher than the single pass, the overall time complexity of the code is `O(n log n)`.

Space Complexity:

The space complexity is `O(1)` (constant space) aside from the space required to store the input, because the sorting is done in-place (altering the input list itself) and only a few extra variables (`i`, `x`, `s`) are used to keep track of the running total and index. Therefore, the space required does not depend on the size of the input list.