3074. Apple Redistribution into Boxes

Easy Greedy Array Sorting

Problem Description

You have two lists: one represents the number of apples in `n` different packs (`apple`), and the other represents the capacity of `m` boxes (`capacity`). The goal here is to fit all the apples from the packs into the boxes. Now, you can distribute apples from any single pack into multiple boxes if necessary, but what you're trying to find out is the smallest number of boxes you can use to hold all the apples.

Imagine you're moving and have a collection of differently sized boxes and many items of varying amounts. You'll want to use as few boxes as possible, by filling up the largest boxes first. This problem demands a similar strategy. We want to use the biggest boxes to their full potential to minimize the number of boxes used overall.

Intuition

The underlying intuition of the solution is based on maximizing the utilization of larger boxes to minimize the total number used. Think about filling up a water tank using different-sized buckets—you'd use the largest buckets first to fill it more quickly.

Applying this mentality to the apples and boxes, you would sort the boxes from largest to smallest capacity. By using the bigger boxes first, you ensure that each box holds as many apples as possible, decreasing the total number of boxes needed.

Once sorted, it's simply a matter of going through the boxes in order, adding their capacity to a running total. You keep adding the capacities until you've accounted for all the apples. The number of boxes added at the point where the running total of capacity exceeds or meets the total number of apples is the minimum number of boxes needed.

This approach is known as a greedy algorithm because at each step, you're making the choice that seems best at the moment (using the largest box available).

Solution Approach

The given solution uses a simple yet effective greedy algorithm. The algorithm can be described by the following steps:

1. **Sorting**: First, the algorithm sorts the box capacities in descending order. This is done using the built-in `sort` method with the `reverse=True` flag set, which reorders the `capacity` list from highest to lowest. This allows us to use the boxes with the most capacity first.
2. **Summation**: Before we start allocating apples to boxes, we calculate the total number of apples we need to pack by summing all the elements of the `apple` array. This gives us the variable `s`, which represents the sum of all apples.
3. **Allocation**: We then go through the sorted list of boxes and subtract the capacity of each box from the running total of apples `s`. We start with the largest box and work our way down to the smallest.
4. **Check and Return**: After each box's capacity is subtracted from the total `s`, we check if `s` becomes less than or equal to zero. This check is done after each box is accounted for in the loop `for i, c in enumerate(capacity, 1)`. If `s` is less than or equal to zero, it means all apples have been allocated into the boxes we've considered so far. We return `i`, the index representing the count of boxes used, at that point.

This approach makes efficient use of the available space by prioritizing larger boxes, ensuring that the number of boxes we end up using is minimized. It's a classic example of a greedy algorithm, where making the locally optimal choice (using the biggest box next) also leads to a global optimum (using the smallest number of boxes).

```
class Solution:
    def minimumBoxes(self, apple: List[int], capacity: List[int]) -> int:
        capacity.sort(reverse=True)  # Step 1: Sorting
        s = sum(apple)               # Step 2: Summation
        for i, c in enumerate(capacity, 1):  # Step 3: Allocation
            s -= c
            if s <= 0:               # Step 4: Check and Return
                return i
```

We don't need complex data structures here; a simple list and basic operations like sorting and iteration are sufficient to implement this algorithm effectively.

Example Walkthrough

Suppose we have the following small example:

- `apple` = [4, 5, 7]
- `capacity` = [6, 4, 10, 3]

The question is: What is the smallest number of boxes we can use to fit all the apples?

Let's follow the solution steps to find out:

1. **Sorting**:
   - We sort `capacity` in descending order: [10, 6, 4, 3]
2. **Summation**:
   - By summing [4, 5, 7], we find that `s` (total number of apples) is 16.
3. **Allocation**:
   - We take boxes in order from our sorted list, subtract their capacity from `s`, and count the number of boxes used.
   - 1st box: `s` = 16 - 10 = 6 (1 box used)
   - 2nd box: `s` = 6 - 6 = 0 (2 boxes used)
4. **Check and Return**:
   - After using the second box, `s` is now 0, which means we have allocated all the apples into the boxes.

According to the walk through, to fit all our apples, we need a minimum of 2 boxes. This concludes our allocation, and the function would return `i = 2`.

Solution Implementation

Python

```
class Solution:
    def minimum_boxes(self, apples: List[int], capacities: List[int]) -> int:
        # Sort the capacities in descending order
        capacities.sort(reverse=True)

        # Calculate the total number of apples to distribute
        total_apples = sum(apples)

        # Initialize the number of boxes used
        boxes_used = 0

        # Iterate over the sorted capacities to distribute the apples
        for capacity in capacities:
            # Subtract the current box capacity from the total apples
            total_apples -= capacity

            # Increment the number of boxes used
            boxes_used += 1

            # Check if all apples have been distributed
            if total_apples <= 0:
                # If all apples are distributed, return the number of boxes used
                return boxes_used

        # If the code reaches here, it implies more boxes are needed
        # than are available in 'capacities' to store all 'apples'
        raise ValueError("Insufficient number of boxes to store all apples")

# The List type needs to be imported from typing module
from typing import List
```

Java

```
import java.util.Arrays; // Required for using the Arrays.sort() method

class Solution {

    /**
     * Finds the minimum number of containers required to store all apples.
     *
     * @param apples Array representing the number of apples in each box.
     * @param capacities Array representing the capacity of each container.
     * @return The minimum number of containers required.
     */
    public int minimumBoxes(int[] apples, int[] capacities) {
        // Sort the capacities array in ascending order so we can use the largest capacities last
        Arrays.sort(capacities);

        // Calculate the total number of apples that need to be stored.
        int totalApples = 0;
        for (int apple : apples) {
            totalApples += apple;
        }

        // Start using containers from the largest to store the apples.
        for (int i = 1, n = capacities.length; ++i) {
            // Subtract the capacity of the used container from the total apples count.
            totalApples -= capacities[n - i];

            // Check if all apples are stored. If so, return the number of containers used.
            if (totalApples <= 0) {
                return i;
            }
        }
        // Note: The loop will always terminate with a return inside the loop,
        // so there is no need for an additional return statement here.
    }
}
```

C++

```
#include <vector>
#include <algorithm>
#include <numeric>

class Solution {
public:
    // Function to determine the minimum number of boxes required to hold
    // a specific number of apples.
    int minimumBoxes(vector<int>& apples, vector<int>& capacities) {
        // Sort capacities in non-increasing order to use the largest boxes first.
        sort(capacities.rbegin(), capacities.rend());

        // Accumulate the total number of apples that need to be boxed.
        int totalApples = accumulate(apples.begin(), apples.end(), 0);

        // Iterate through the sorted capacities to find the minimum number of boxes required.
        for (int boxCount = 1; ; ++boxCount) {
            // Subtract the current box capacity from the total apples.
            totalApples -= capacities[boxCount - 1];

            // If all apples are accounted for with the current number of boxes, return it.
            if (totalApples <= 0) {
                return boxCount;
            }
        }
        // Note: The loop has no exit condition besides the return within the loop,
        // which assumes that the given 'capacities' vector is sufficient.
    }
};
```

TypeScript

```
function minimumBoxes(apples: number[], capacities: number[]): number {
    // Sort the capacities array in descending order
    capacities.sort((a, b) => b - a);

    // Calculate the total number of apples
    let totalApples = apples.reduce((accumulator, current) => accumulator + current, 0);

    // Initialize the index (which will represent the number of boxes used)
    let boxIndex = 0;

    // Iterate until all apples are placed in boxes
    while (totalApples > 0 && boxIndex < capacities.length) {
        // Deduct the capacity of the current largest box from total apples
        totalApples -= capacities[boxIndex];

        // Move to the next box
        boxIndex++;
    }

    // If totalApples is less than or equal to 0, all apples are in boxes
    // Return the count of boxes used (boxIndex)
    // If totalApples is not less than or equal to 0, we've run out of boxes
    // before accommodating all apples, hence return boxIndex
    return boxIndex;
}
```

```
class Solution:
    def minimum_boxes(self, apples: List[int], capacities: List[int]) -> int:
        # Sort the capacities in descending order
        capacities.sort(reverse=True)

        # Calculate the total number of apples to distribute
        total_apples = sum(apples)

        # Initialize the number of boxes used
        boxes_used = 0

        # Iterate over the sorted capacities to distribute the apples
        for capacity in capacities:
            # Subtract the current box capacity from the total apples
            total_apples -= capacity

            # Increment the number of boxes used
            boxes_used += 1

            # Check if all apples have been distributed
            if total_apples <= 0:
                # If all apples are distributed, return the number of boxes used
                return boxes_used

        # If the code reaches here, it implies more boxes are needed
        # than are available in 'capacities' to store all 'apples'
        raise ValueError("Insufficient number of boxes to store all apples")

# The List type needs to be imported from typing module
from typing import List
```

Time and Space Complexity

The time complexity of the function `minimumBoxes` is `O(n * log(m) + n)`. This is because the sort function applied to the `capacity` list has a complexity of `O(m * log(m))`, where `m` is the length of the `capacity` list. Following the sort, there is a for loop which iterates over the sorted `capacity` list, and this loop may run up to `n` times, where `n` is the length of the `apple` list. Therefore, the iteration adds an `O(n)` complexity to the total, making the combined time complexity `O(m * log(m) + n)`.

The space complexity of the function is `O(log(m))`. This is due to the space needed for the sorting algorithm for a list of length `m`. Most sorting algorithms, such as Timsort (used in Python's sort function), have a logarithmic space footprint because they need additional space to temporarily store elements while sorting.