2548. Maximum Price to Fill a Bag

Medium | Greedy | Array | Sorting Leetcode Link

Problem Description

The problem provides us with a list of items, each characterized by two values: `price` and `weight`. The `price` represents the value of the item, while the `weight` represents its physical weight. We are also given a `capacity`, which is the maximum weight limit of a bag we want to fill with items (or portions of items) to maximize the total price of items inside the bag.

A key aspect of the problem is that items can be divided into portions, with each portion keeping the same price-to-weight ratio as the original item. That means we can take part of an item if taking the whole item would exceed the bag's weight limit. The challenge is to find the maximum possible total price we can achieve given the bag's `capacity`.

Since the result must be within `10^-5` of the actual answer, we're dealing with an approximation and a floating-point number for the end result.

Intuition

The solution approach to this problem is similar to the classic knapsack problem which is generally solved using dynamic programming. However, since this problem allows for dividing the items (thus making it a fractional knapsack problem), we can use a greedy algorithm instead.

To maximize the total price, we intuitively want to prioritize taking items or portions of items that have the highest price-to-weight ratio first, since they provide the most value for the least amount of weight.

Here's how we arrive at the solution:

1. We sort the items based on their price-to-weight ratio in ascending order. Sorting by `x[1] / x[0]` implies sorting by weight-to-price ratio, which is the inverse of our desired price-to-weight ratio, effectively sorting by cheapest cost efficiency first.

2. Then, we iterate over the sorted items and keep adding them to our bag.

3. For each item, `v = min(w, capacity)` determines the actual weight we can add to the bag. This will either be the full weight `w` of the item if it fits, or the remaining `capacity` of the bag if there isn't enough space for the whole item.

4. `ans += v / w * p` calculates the price of the portion added to the bag. We update the total price in `ans` accordingly.

5. We deduct the weight of the item or portion of the item from the remaining `capacity`.

6. If the loop terminates and `capacity` is not zero, it means we were unable to fill the bag completely, which should not happen given we can take portions of items. Therefore, if `capacity` is zero, we return `ans` which holds the maximum total price, and if `capacity` is greater than zero (it shouldn't be according to the problem statement), we return `-1`.

By employing a greedy strategy, we ensure that we always take the item or portion of the item that has the most value relative to weight without exceeding the capacity of our bag.

Solution Approach

The solution approach leverages a greedy algorithm, which is often used when we're looking to make a sequence of choices that are locally optimal (maximizing or minimizing some criteria) with the goal of finding a global optimum.

The algorithm sorts the array of `items` based on a key, which is the ratio of weight to price (`x[1] / x[0]`). This sorting step is crucial as it allows us to later iterate through the items in order of what provides the least value per weight unit, due to the ascending order.

Once the items are sorted, the implementation uses a for loop to iterate over each item. The `min` function determines how much of the current item's weight can be used without exceeding the bag's `capacity`. This means that if the bag's remaining `capacity` is greater than or equal to the current item's weight, we can take the full item. Otherwise, we can only take a portion of it that fits the remaining capacity.

Post this evaluation, it calculates the price of the amount taken by multiplying the ratio of the weight we could fit to the item's total weight with the item's total price (`v / w * p`). This product gives us the value of the portion of the item being considered, which is accumulated in the `ans` variable representing the current total price.

The capacity of the bag is decreased by the weight we just decided to take. This ensures that in the next iteration we're accounting only for the remaining capacity.

After the loop, the conditional `-1 if capacity else ans` serves as a final check. The intent here is that if we have any remaining `capacity`, the algorithm should return `-1`, indicating that the bag was not filled correctly. However, based on the problem description, this condition should not occur because we are allowed to take any fractional part of an item, which implies that we should always end up with `capacity` reaching zero before we run out of items. Still, this is included perhaps as a safety check or due to an oversight that leads to redundancy. If the `capacity` is indeed zero, `ans` is returned, giving us the maximum total price for filling the bag.

In terms of data structures, the input array `items` and the iteration for accessing each `(p, w)` are straightforward. No additional data structures are used outside of the variables for accumulating the total price and maintaining remaining capacity.

Example Walkthrough

Let's consider an example to illustrate the solution approach. Imagine we have a list of items with the following `price` and `weight`:

1. Item 1: price = 60, weight = 10
2. Item 2: price = 100, weight = 20
3. Item 3: price = 120, weight = 30

And let's say the capacity of our bag is 50.

Now, here is how we would apply the solution approach step-by-step:

1. First, we calculate the price-to-weight ratio for each item:
   - Item 1: 60 / 10 = 6.0
   - Item 2: 100 / 20 = 5.0
   - Item 3: 120 / 30 = 4.0
2. Next, we sort the items based on their price-to-weight ratio in descending order, so we first fill our bag with items that give the maximum value:

   Since we want to sort based on highest value first we have:
   - Item 1: ratio = 6.0
   - Item 2: ratio = 5.0
   - Item 3: ratio = 4.0

   The items are already sorted in descending order of price-to-weight ratio.
3. Now, we start to fill the bag. We take the first item in full because the capacity allows it.
   - Add Item 1: price = 60, weight = 10. Remaining capacity = 40.
4. Move to the second item:
   - Add Item 2: price = 100, weight = 20. Remaining capacity = 20.
5. Finally, for the third item, we can't add it in full because the remaining capacity is only 20 and the weight is 30. So, we calculate how much we can take:
   - We take (\frac{20}{30}) of Item 3: price contribution = (\frac{20}{30} \times 120 = 80).
6. We calculate the final price:
   - Total price = price of Item 1 + price of Item 2 + fraction of price from Item 3.
   - Total price = 60 + 100 + 80 = 240.
7. The remaining capacity of the bag is now zero. We've maximized the total price of items in the bag without exceeding the weight capacity, successfully applying the greedy strategy to this fractional knapsack problem.

The resulting maximum total price to fill the bag is 240, following the greedy algorithm based on the price-to-weight ratio.

Python Solution

```python
from typing import List

class Solution:
    def max_price(self, items: List[List[int]], capacity: int) -> float:
        # Initialize the maximum price achieved to zero
        max_price = 0
        # Sort items by their value to weight ratio in ascending order
        items.sort(key=lambda item: item[0] / item[1], reverse=True)

        # Loop through the sorted items
        for price, weight in items:
            # Take the minimum of item's weight or remaining capacity
            weight_to_take = min(weight, capacity)
            # Calculate the price for the weight taken
            price_for_weight = (weight_to_take / weight) * price
            # Add to the total maximum price
            max_price += price_for_weight
            # Decrease the remaining capacity
            capacity -= weight_to_take
            # Break if the capacity is filled
            if capacity == 0:
                break

        # Return the total maximum price if the capacity has been completely used, else return -1
        return max_price if capacity == 0 else -1

# Example usage:
# solution = Solution()
# items = [[60, 10], [100, 20], [120, 30]]  # Each item is [price, weight]
# capacity = 50
# print(solution.max_price(items, capacity))  # Expected output is the maximum price that fits into the capacity
```

Java Solution

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

class Solution {

    // Function to calculate the maximum price achievable within the given capacity
    public double maxPrice(int[][] items, int capacity) {
        // Sort the items array based on value-to-weight ratio in descending order
        Arrays.sort(items, (item1, item2) -> item2[0] * item1[1] - item1[0] * item2[1]);

        // Variable to store the cumulative value of chosen items
        double totalValue = 0;

        // Iterate through each item
        for (int[] item : items) {
            int price = item[0];
            int weight = item[1];

            // Determine the weight to take, up to the remaining capacity
            int weightToTake = Math.min(weight, capacity);

            // Compute value contribution of this item based on the weight taken
            double valueContribution = (double) weightToTake / weight * price;

            // Add the value contribution to the total value
            totalValue += valueContribution;

            // Subtract the weight taken from the remaining capacity
            capacity -= weightToTake;

            // If no capacity is left, break the loop as no more items can be taken
            if (capacity == 0) {
                break;
            }
        }

        // If there is unused capacity, the requirement to fill the exact capacity is not met
        // In this context, return -1 to indicate the requirement is not fulfilled
        return capacity > 0 ? -1 : totalValue;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to calculate maximum price of items fit into a given capacity
    double maxPrice(std::vector<std::vector<int>>& items, int capacity) {
        // Sorting the items based on the price-to-weight ratio in descending order
        std::sort(items.begin(), items.end(), [&](const auto& item1, const auto& item2) {
            return item1[1] * item2[0] < item1[0] * item2[1];
        });

        double totalValue = 0.0; // Initialize total value to accumulate

        // Iterate through each item
        for (const auto& item : items) {
            int price = item[0]; // Price of the current item
            int weight = item[1]; // Weight of the current item
            int weightToTake = std::min(weight, capacity); // Weight to take of current item

            // Add value of the current item fraction to total value
            totalValue += static_cast<double>(weightToTake) / weight * price;

            // Decrease the capacity by the weight taken
            capacity -= weightToTake;

            // Break the loop if the capacity is fully utilized
            if (capacity == 0) {
                break;
            }
        }

        // Return -1 if there's remaining capacity, indicating incomplete filling
        // Otherwise return the total value of items taken
        return capacity > 0 ? -1 : totalValue;
    }
};
```

Typescript Solution

```typescript
// Define the maxPrice function which calculates the maximum price that can
// be achieved given a set of items and a capacity constraint
function maxPrice(items: number[][], capacity: number): number {
    // Sort the items based on the unit price in descending order
    items.sort((a, b) => b[1] / b[0] - a[1] / a[0]);

    // Initialize the maximum price achievable to 0
    let maxPrice = 0;

    // Loop through each item
    for (const [price, weight] of items) {
        // Determine how much of the item's weight can be used, up to the remaining capacity
        const usableWeight = Math.min(weight, capacity);

        // Increment the maximum price by the value of the current item,
        // prorated by the fraction of usable weight to its full weight
        maxPrice += (usableWeight / weight) * price;

        // Decrease the remaining capacity by the weight of the current item used
        capacity -= usableWeight;

        // If no capacity is left, break out of the loop
        if (capacity === 0) break;
    }

    // If there is no capacity left (i.e., the knapsack is filled to its limit),
    // return the maximum price, otherwise, return -1 indicating not all capacity was used
    return capacity === 0 ? maxPrice : -1;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code consists of two main operations: sorting the list and iterating through the list.

- Sorting the list of `items` has a time complexity of `O(NlogN)`, where `N` is the length of the `items`. This is because the built-in `sorted()` function in Python uses TimSort (a combination of merge sort and insertion sort) which has this time complexity for sorting an array.

- The iteration through the sorted list has a time complexity of `O(N)` since each item is being accessed once to calculate the proportional value and update the remaining `capacity`.

Combining these two operations, the overall time complexity is `O(NlogN) + O(N)`, which simplifies to `O(NlogN)` as the sorting operation is the dominant term.

Space Complexity

- The space complexity of sorting in Python is `O(N)` because the `sorted()` function generates a new list.

- The additional space used in the code for variables like `ans` and `v` are constant `O(1)`.

Therefore, the overall space complexity is `O(N)` due to the sorted list that is created and used for iteration.