1710. Maximum Units on a Truck

Problem Description

Greedy Array Sorting

The given problem is a variation of the classical knapsack problem, where we're tasked with loading a truck with boxes. Each box

boxTypes where boxTypes[i] = [number0fBoxes_i, number0fUnitsPerBox_i]. number0fBoxes_i tells us the number of available boxes of type i and numberOfUnitsPerBox_i indicates the number of units in each box of type i. We are also given truckSize which represents the maximum number of boxes that fit onto the truck. Our goal is to maximize the total number of units loaded onto the truck without exceeding the truck size limit.

contains a certain number of units, and each type of box has a corresponding number of units. We're provided with an array

The intuition behind the solution is based on a greedy algorithm. In order to maximize the number of units, we prioritize loading

Intuition

boxes with the highest number of units per box. This is because filling the truck with boxes that contain the most units will generally lead to a higher total number of units on the truck.

Easy

We start by sorting the boxTypes array in descending order based on numberOfUnitsPerBox, so the box types with the most units per box come first. This allows us to go through the boxTypes array and add boxes to the truck in the order that maximizes the number of units. For each type of box, we take as many boxes as we can until we either run out of that type of box or reach the truck's capacity.

or negative). If at any point the remaining truck size becomes less than the number of boxes available, we only take as many boxes as will fit in the truck, using the min function. This ensures we do not exceed the truckSize constraint.

The number of units from each box type is added to our cumulative answer until the truck is full (when truckSize becomes zero

By repeatedly choosing the boxes with the maximum units per box and keeping track of the remaining capacity on the truck, we can ensure that the final number of units is maximized. Solution Approach

The provided solution uses a greedy algorithm to solve the problem of maximizing the number of units on the truck. Here is a step-by-step explanation of the implementation strategy:

of each sub-array (-x[1]). The negative sign indicates that we want a descending order.

by the number of boxes used, which is a (representing number of Boxes).

Sorting the boxTypes array: The solution starts by sorting the boxTypes array in descending order of the number of units per

box. This is achieved by passing a custom lambda function to the sorted method, which sorts based on the second element

Initializing the answer variable: A variable ans is initialized to store the total number of units that can be loaded onto the

minimum of the remaining truckSize and the numberOfBoxes of the current type, then multiplying it by numberOfUnitsPerBox

Updating the remaining truckSize: After adding the units from the current box type to the answer, the truckSize is reduced

Returning the result: Once the loop completes, whether by filling the truck or exhausting all box types, the total number of

The use of sorting and a greedy approach is the key aspect of the algorithm, making sure that every step taken is optimal in

terms of the number of units added per box. The for loop along with the min function and a simple accumulator variable (ans)

are the major elements in this simple and efficient implementation that guarantees the maximum number of units that can be

sorted(boxTypes, key=lambda x: -x[1])

truckSize -= a

units ans is returned as the solution.

loaded onto the truck within the given constraints.

boxTypes = [[1, 3], [2, 2], [3, 1]]

Imagine we are given the following boxTypes array and truckSize:

truck.

Iterating through sorted boxTypes: The algorithm iterates over each box type in the sorted boxTypes array. Calculating units to add: For each box type, the solution calculates how many units can be added. This is done by taking the

- to get the total units for that transaction. ans += b * min(truckSize, a)
- Checking if the truck is full: The loop will exit early using a break statement if there is no more capacity in the truck (truckSize <= 0). This prevents unnecessary iterations once the truck is filled.
- **Example Walkthrough** Let's walk through a small example to illustrate the solution approach.

 There are 3 boxes with 1 unit per box. The truck can carry at most 4 boxes. **Step-by-Step Solution**

The boxTypes array consists of subarrays where the first element is the number of boxes and the second element is the number

sorted(boxTypes, key=lambda x: -x[1])

truckSize = 4

of units per box. Here:

• There is 1 box with 3 units per box.

There are 2 boxes with 2 units per box.

After sorting, our boxTypes array looks like this:

boxTypes = [[1, 3], [2, 2], [3, 1]]

numberOfUnitsPerBox (which is 3).

ans += 1 * min(4, 1) # Results in ans = 3truckSize -= min(4, 1) # Decreases truckSize to 3

(which is 2) multiplied by numberOfUnitsPerBox (which is 2).

(which is 3) multiplied by numberOfUnitsPerBox (which is 1).

units we can load into the truck given the constraints.

space with lower value boxes until the truck reaches capacity.

def maximumUnits(self, boxTypes: List[List[int]], truckSize: int) -> int:

Sort the box types by the number of units per box in a non-increasing order.

Increment the maximum units by the units from the loaded boxes.

by comparing what's left of the truck's capacity with the available boxes.

// Sort the array of box types in descending order based on the number of units per box.

// This will help to maximize the number of units we can load onto the truck.

// Initialize the result variable that will hold the maximum units.

maxUnits += unitsPerBox * Math.min(truckSize, numberOfBoxes);

// Return the total maximum units we can carry in the truck.

// Initialize the total number of units to add to the truck

// Initialize the number of boxes accumulated on the truck

// If adding the current number of boxes doesn't exceed the truck size

// Add the current number of boxes to the accumulated count

// Add as many units as can fit in the remaining space and break

Sort the box types by the number of units per box in a non-increasing order.

Increment the maximum units by the units from the loaded boxes.

by comparing what's left of the truck's capacity with the available boxes.

// Add the units from all current boxes to the total

const remainingSpace = truckSize - boxesAccumulated;

// Return the total number of units that can be carried by the truck

// Calculate the remaining space on the truck

totalUnits += remainingSpace * unitsPerBox;

// Iterate through the sorted array of box types

for (const [numBoxes, unitsPerBox] of boxTypes) {

boxesAccumulated += numBoxes;

boxTypes.sort(key=lambda x: -x[1])

truckSize -= boxes_to_load

if truckSize <= 0:</pre>

for number of boxes, units per box in boxTypes:

Calculate the number of boxes the truck can load

Decrease the truckSize by the number of boxes loaded.

boxes_to_load = min(truckSize, number_of_boxes)

If the truck is full, break out of the loop.

max_units += units_per_box * boxes_to_load

Iterate through the box types.

if (boxesAccumulated + numBoxes < truckSize) {</pre>

totalUnits += unitsPerBox * numBoxes;

let totalUnits = 0;

} else {

break;

return totalUnits;

from typing import List

let boxesAccumulated = 0;

// Subtract the number of boxes we've taken from the truckSize.

// If the truck is full, no need to continue checking other box types.

Initialize the maximum number of units the truck can carry.

Calculate the number of boxes the truck can load

Decrease the truckSize by the number of boxes loaded.

boxes_to_load = min(truckSize, number_of_boxes)

If the truck is full, break out of the loop.

public int maximumUnits(int[][] boxTypes, int truckSize) {

Arrays.sort(boxTypes, (a, b) -> b[1] - a[1]);

// Loop through the sorted boxTypes array.

// Number of boxes of this type.

int numberOfBoxes = boxType[0];

truckSize -= numberOfBoxes;

if (truckSize <= 0) {</pre>

break;

return maxUnits;

for (int[] boxType : boxTypes) {

max_units += units_per_box * boxes_to_load

ans += 1 * min(1, 3) # Adds 1 to ans, resulting in ans = 8

Note that in this example, the array was already sorted, so it remains the same.

We initialize our answer variable, ans, to 0. It will accumulate the total units placed into the truck.

We iterate through the sorted boxTypes array. Our goal is to consider box types with the most units first.

For the first box type [1, 3], we take the minimum of truckSize (which is 4) and numberOfBoxes (which is 1) multiplied by

Next, for the second box type [2, 2], we take the minimum of remaining truckSize (which is now 3) and numberOfBoxes

Continuing, for the third box type [3, 1], we again find the minimum of remaining truckSize (now 1) and numberOfBoxes

Return the result: We've finished the process, and the ans variable now holds the value 8, which is the maximum number of

By following these steps using a greedy approach, we have maximized the number of units in our truck. The key was to prioritize

the box types with the most units, aiming to fill the truck with the most valuable (unit-wise) boxes first, and then fill the remaining

First, we sort boxTypes based on the number of units per box in descending order:

- ans += 2 * min(3, 2) # Adds 4 to ans, resulting in ans = 7truckSize -= min(3, 2) # Decreases truckSize to 1
- truckSize -= min(1, 3) # Decreases truckSize to 0 Now, the truck is full (truckSize is now 0), so even though there are still boxes left, we cannot add more.
- **Solution Implementation Python**
 - if truckSize <= 0:</pre> break # Return the total maximum units the truck can carry. return max_units

// Number of units per box of this type. int unitsPerBox = boxType[1]; // Calculate the number of boxes we can take of this type without exceeding truckSize, // and add the number of units those boxes contribute to the total maxUnits.

int maxUnits = 0;

from typing import List

max_units = 0

boxTypes.sort(key=lambda x: -x[1])

truckSize -= boxes_to_load

for number of boxes, units per box in boxTypes:

Iterate through the box types.

class Solution:

Java

class Solution {

```
class Solution {
public:
    int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
       // Sort the box types based on the number of units per box in descending order
        sort(boxTypes.begin(), boxTypes.end(), [](const auto& a, const auto& b) {
            return a[1] > b[1];
        });
        int totalUnits = 0; // This will store the total number of units we can load on the truck
       // Iterate through each type of box, since they are sorted we will pick the ones with
        // the most units first, maximizing our total units carried
        for (const auto& boxType : boxTypes) {
            int numberOfBoxes = boxType[0]; // Number of boxes of this type
            int unitsPerBox = boxType[1]; // Number of units per box for this type
            // Calculate the number of boxes we can actually take of this type, which is the
            // smaller between the number of boxes available, and the truck size remaining
            int boxesToTake = min(truckSize, numberOfBoxes);
            // Add the corresponding number of units to the total units
            totalUnits += unitsPerBox * boxesToTake;
            // Deduct the number of boxes taken from the truck's remaining capacity
            truckSize -= boxesToTake;
            // If the truck is full, we break out of the loop as we cannot load more boxes
            if (truckSize <= 0) break;</pre>
        // Return the total number of units that fit in the truck
        return totalUnits;
};
TypeScript
function maximumUnits(boxTypes: number[][], truckSize: number): number {
    // Sort the array of box types in descending order based on the number of units per box
    boxTypes.sort((a, b) => b[1] - a[1]);
```

class Solution: def maximumUnits(self, boxTypes: List[List[int]], truckSize: int) -> int: # Initialize the maximum number of units the truck can carry. max_units = 0

break # Return the total maximum units the truck can carry. return max_units Time and Space Complexity The time complexity of the provided code is $O(n \log n)$ where n is the length of the boxTypes list. This is because the code sorts boxTypes by the number of units in each box in descending order, which takes 0(n log n) time. After sorting, the code iterates through the boxTypes list, which in the worst case takes O(n) time if the truck can carry all boxes. However, sorting dominates the complexity. Therefore, the overall time complexity remains $O(n \log n)$.

The space complexity of the code is 0(1) as no additional space is used that scales with the input size. The sorting is done inplace (assuming the sort algorithm used by Python, Timsort, which has a space complexity of 0(1) for this scenario), and only a fixed number of variables are used regardless of the input size.