University Of Asia Pacific

Department of CSE

Course Code : **CSE 208**

Course Title : **Data Structures and Algorithms II Lab**

No Of Assignment : **06**

Date of Submission : **24.11.2024**

Submitted By:

Name : **Nusrat Ahmmed Ekra**

Student ID : 22201251

Section : E2

Semester : 2nd Year 2nd Semester

Submitted To:

Suri Dipannita Sayeed

Lecturer,

CSE,UAP

Problem:02: Discuss the difference with naive approach that is trying all possible subsets with all different fraction.

Solution:

Naive Approach: Exploring All Possible Subsets with All Different Fractions

The naive approach for solving the fractional knapsack problem involves trying all possible combinations of items and their possible fractions, essentially brute-forcing through the entire solution space. This is a very exhaustive method and guarantees an optimal solution but is highly computationally expensive. Let's break it down:

1. **Subset Generation**:
   The naive approach first generates all possible subsets of the given items. There are 2^n possible subsets of items where *n*is the number of items. Each subset can include or exclude any combination of the items.
2. **Fraction Handling**:
   For each subset, the naive approach also needs to consider every possible fraction of each item. For instance, if an item weighs *w* and the knapsack can hold more weight, you would consider fractions like 0.1, 0.2, ..., up to 1. This significantly increases the number of options to check, as the number of fractions for each item adds to the complexity.
3. **Total Value Calculation**:
   After generating each subset and considering all possible fractions for each item, the naive approach calculates the total value for each subset/fraction combination. The total value is compared with the capacity of the knapsack to see if it fits.
4. **Selecting the Best Combination**:
   After evaluating all the subsets and fractions, the best combination (the one yielding the highest total value without exceeding the knapsack's capacity) is chosen.

Time Complexity of Naive Approach:

- The time complexity of generating all subsets of *n* items is ,$O(2^n)$; where *n* is the number of items.
- For each subset, you also need to evaluate the possible fractions of each item. If there are *m* possible fractions (e.g., 10 possible fractions per item), then the total number of combinations to evaluate is $O(2^n*m)$.
- Thus, the overall time complexity of the naive approach is $O(2^n*m)$, which is **exponential** and becomes very inefficient for larger values of *n*.

Greedy Approach: Optimized and Efficient

The greedy approach, on the other hand, takes a much more efficient path by considering the value-to-weight ratio of the items and making a series of greedy choices. Here's how it differs:

1. **Value-to-Weight Ratio**:
   The greedy approach calculates the value-to-weight ratio for each item, which tells you how much value you get per unit weight. This allows for prioritizing items that provide the most value per unit weight.
2. **Sorting**:
   Once the ratios are computed, the items are sorted in descending order of this ratio. This means that the item providing the most value per unit weight will be considered first, followed by the next best item, and so on.
3. **Greedy Selection**:
   a. Starting from the item with the highest value-to-weight ratio, the greedy algorithm adds the entire item to the knapsack if it fits. If not, it takes the fraction of the item that can fit into the remaining capacity.
   b. This process continues until the knapsack is full or all items have been considered.

Time Complexity of Greedy Approach:

- Sorting the items based on the value-to-weight ratio takes O(nlogn), where *n* is the number of items.
- After sorting, iterating through the items and adding them (or their fractions) to the knapsack takes O(n).
- Hence, the overall time complexity of the greedy approach is O(nlogn), which is **polynomial** and much more efficient than the naive approach.

Comparison Between the Approaches

Aspect	Naive Approach	Greedy Approach
Strategy	Try all possible subsets of items and all possible fractions.	Select items based on value-to-weight ratio, prioritize high ratios.
Subset Generation	Generates all subsets of items .	No subset generation needed; only sorting and selecting items.
Fraction Handling	Considers all possible fractions for each item.	Only considers whole items or fractions as required based on capacity.
Time Complexity	$O(2^n*m)$. (Exponential time).	O(nlogn) 9Polynomial time).
Space Complexity	High, as it needs to store all subsets and fractions.	Low, only needs to store items and their ratios.
Optimality	Guarantees an optimal solution.	Guarantees an optimal solution for fractional knapsack.
Efficiency	Computationally expensive for large *n*.	Much more efficient and practical for large datasets.
Practical Use	Impractical for large datasets due to exponential growth.	Practical and scalable for large datasets.

Problem:03: Discuss coin change greedy approach and computational complexity.

Solution:

Greedy Approach

The goal is to find the minimum number of coins needed to make a given amount using a set of coin denominations.

Steps:

1. **Sort the Coins**: Arrange the coins in descending order of their values.
2. **Pick the Largest Coin**: Start with the largest coin denomination that is less than or equal to the remaining amount.
3. **Reduce the Remaining Amount**: Subtract the value of the chosen coin from the remaining amount.
4. **Repeat**: Continue until the amount becomes zero or no coins can be used.

Assumptions:

This approach works well when the coin denominations form a canonical system (e.g., the denominations are multiples of smaller ones, like [1, 5, 10, 25]).

Example:

Coins: [1, 5, 10, 25]
Amount: 63 Process:

- Pick 2 coins of 25 (50 remaining = 13).
- Pick 1 coin of 10 (remaining = 3).
- Pick 3 coins of 1 (remaining = 0).

Output: 6 coins.

Analysis of Computational Complexity

Sorting:
Sorting the coin denominations in descending order takes O(nlogn) where n is the number of denominations.

Greedy Iteration:
Iterating through the coins to calculate the number of coins needed takes O(n).

Overall Time Complexity:
The total complexity is dominated by the sorting step: O(nlogn).

Naive Recursive Algorithm

Recursive Relation:

- At each step, decide whether to include a coin in the solution or skip it.
- If the amount becomes zero, one valid way is found.

- If the amount becomes negative or no coins are left, it's an invalid way.

Base Cases:

- If amount == 0, return 1 (a valid solution is found).
- If amount < 0 or no coins left, return 0 (no solution).

Example

Input:

coins = [1, 2, 3]

amount = 4

Execution:

Start with coinChange([1, 2, 3], 3, 4)

Explore two cases:

Include the coin 3 → coinChange([1, 2, 3], 3, 1)

Exclude the coin 3 → coinChange([1, 2, 3], 2, 4)

Continue recursively until all possibilities are explored.

Output:

The total number of ways to make the amount (e.g., 4 for this example).

Complexity

Time Complexity: $O(2^n)$ in the worst case because each coin leads to two recursive calls.

Space Complexity: O(n) for the recursive stack.

Comparison to Other Approaches

Approach	Complexity	Optimality	Notes
Greedy	O(nlogn)	Optimal for canonical systems.	Fast, easy to implement, but fails for non-canonical systems.
Dynamic Programming	O(n*V)	Always optimal.	Solves all cases; slower for large amounts V.
Naive Brute Force	Exponential $O(2^n)$	Always optimal.	Computationally expensive; impractical for large inputs.