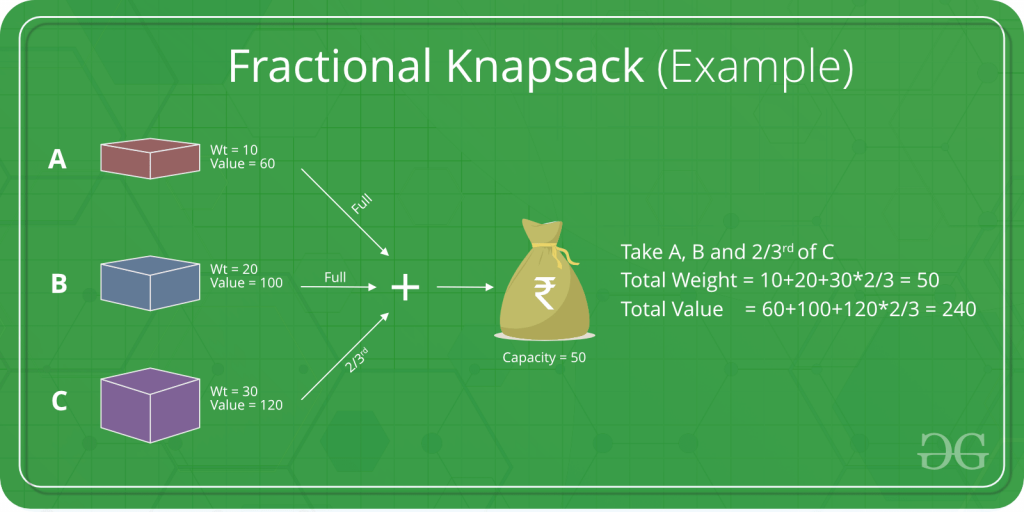
|  |  |
| --- | --- |
| **AssignmentNo.** | **3** |
| **Title** | To Implement fractional Knapsack problem using a greedy method |
| **PROBLEM STATEMENT/DEFINITION** | Write a program to solve a fractional Knapsack problem using a greedy method |
| **Objectives** | **Understand and implement fractional Knapsack problem using a greedy method.** |
| **Software packages and hardware apparatus used** | Technology : Java/Python  Ubuntu /Linux  Latest Version of 64 bit Operating Systems, Open Source Fedora-GHz. 8 G.B.  RAM, 500 G.B. HDD, 15"Color Monitor, Keyboard, Mouse |
| **References** | 1. <https://www.geeksforgeeks.org/fractional-knapsack-problem/> 2. <https://www.scaler.com/topics/fibonacci-series-in-c/> |
|  |  |

**Concepts Related Theory**

**Greedy Algorithms**

**Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are the best fit for Greedy.**

**For example consider the Fractional Knapsack Problem. The local optimal strategy is to choose the item that has maximum value vs weight ratio. This strategy also leads to a globally optimal solution because we are allowed to take fractions of an item.**



**Fractional Knapsack Problem using Greedy algorithm:**

The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them until we can’t add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

Given the weights and values of N items, in the form of {value, weight} put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack

Note: In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don’t take it.

**Steps:**

* ****Step 1****: Node root represents the initial state of the knapsack, where you have not selected any package.
* TotalValue = 0.
* The upper bound of the root node UpperBound = M \* Maximum unit cost.
* ****Step 2****: Node root will have child nodes corresponding to the ability to select the package with the largest unit cost. For each node, you re-calculate the parameters:
* TotalValue = TotalValue (old) + number of selected packages \* value of each package.
* M = M (old) – number of packages selected \* weight of each package.
* UpperBound = TotalValue + M (new) \* The unit cost of the packaced to be considered next.
* ****Step 3****: In child nodes, you will prioritize branching for the node having the larger upper bound. The children of this node correspond to the ability of selecting the next package having large unit cost. For each node, you must re-calculate the parameters TotalValue, M, UpperBound according to the formula mentioned in step 2.
* ****Step 4****: Repeat Step 3 with the note: for nodes with upper bound is lower or equal values to the temporary maximum cost of an option found, you do not need to branch for that node anymore.
* ****Step 5****: If all nodes are branched or cut off, the most expensive option is the one to look for.

**Algorithm:**

* Calculate the ratio(value/weight) for each item.
* Sort all the items in decreasing order of the ratio.
* Initialize res =0, curr\_cap = given\_cap.
* Do the following for every item “i” in the sorted order:
* If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result.
* Else add the current item as much as we can and break out of the loop.
* Return res.

**Pseudocode:**

Pseudo code for the algorithm:

Fractional Knapsack (Array W, Array V, int M)

1. for i ****<-**** 1 to size (V)

2. calculate cost[i] ****<-**** V[i] / W[i]

3. Sort-Descending (cost)

4. i ← 1

5. while (i <= size(V))

6. if W[i] <= M

7. M ← M – W[i]

8. total ← total + V[i];

9. if W[i] > M

10. i ← i+1

## Conclusion:

The algorithm uses sorting to sort the items which takes O(n×logn) time complexity and then loops through each item which takes O(n).

Hence summing up to a **time complexity of O(n×logn+n)=O(n×logn) and Space Complexity isO ( n ) O(n) O(n).**