**ASSIGNMENT 1**

**Title:** Write a program to implement Fractional knapsack using Greedy algorithm and 0/1 knapsack using dynamic programming. Show that Greedy strategy does not necessarily yield an optimal solution over a dynamic programming approach.

**Software Requirement:** Ubuntu, C++ Compiler

**Theory:**

Greedy algorithms are simple and straight forward. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future. They are easy to invent, easy to implement and most of the time quite efficient. Many problems cannot be solved correctly by greedy approach. Greedy algorithms are used to solve optimization problems

**1. Greedy Approach:** Greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision; whatever situation may arise later.

* When the problem has many feasible solutions with different cost or benefit, finding the best solution is known as an optimization problem. And best solution is known as the optimal solution.
* Decisions are completely locally optimal. This method constructs the solution simply by looking at current benefit without exploring future possibilities and hence known as greedy.
* The choice made under greedy solution procedure are irrevocable, means once we have selected the local best solution, it cannot be backtracked.

Thus, the choice made at each step in the greedy method should be:

* **Feasible:** Choice should satisfy problem constraints.
* **Locally optimal:** Best solution from all feasible solution at current stage should be selected.
* **Irrevocable:** Once the choice is made, it cannot be altered. i.e. if a feasible solution is selected(rejected) in step i, it cannot be rejected (selected) in subsequent stages.

Greedy algorithms are used to find an optimal or near optimal solution to many real-life problems.

* Knapsack problem
* Minimum Spanning Tree(MST)
* Single Source Shortest Path
* Job Sequencing Problem
* Huffman Code Generation

**Definitions of feasibility:** A feasible set (of candidates) is promising if it can be extended to produce not merely a solution, but an optimal solution to the problem. In particular, the empty set is always promising why? (because an optimal solution always exists)

A greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each problem to a smaller one.

**2. Structure Greedy Algorithm**

Algorithm Greedy\_Approach(L,n)

//Description: Solve the given problem using greedy approach

//Input: L - List of possible choices, n-size of solution of given problem.

//Output: Set solution containing solution of given problem.

Solution🡨 Ф

for i🡨 1 to n do

Choice 🡨 Select(L)

if(feasible(Choice U Solution)) then

Solution 🡨 Choice U Solution

end

end

return Solution

**3.Dyanmic Programming**

It is Applicable when sub-problems are not independent.

* Subproblems share subsubproblems
* A divide and conquer approach would repeatedly solve the common subproblems
* Dynamic programming solves every subproblem just once and stores the answer in a table

It isused for **optimization problems**

* A set of choices must be made to get an optimal solution
* Find a solution with the optimal value (minimum or maximum)
* There may be many solutions that lead to an optimal value

**4. Knapsack Problem**

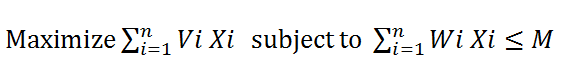
**Problem Statement:** Given a set of items having some weight and value/profit associated with it. The knapsack problem is to find the set of items such that the total weight is less than or equal to a given limit (size of knapsack) and the total value/profit earned is as large as possible.

* Let x = <x1, x2, ….. xn> be set of n items.

w = <w1, w2, ….. wn> set of weight associated with each item in x resp.

v = <v1, v2, ….. vn> set of value associated with each item in x resp.

* Knapsack problem can be



There are two versions of problem:

1. **Fractional knapsack problem**

The setup is same, we can take fractions of items, meaning that the items can be broken into smaller pieces so that we may decide to carry only a fraction of xi of item i, where 0 ≤ xi≤ 1. If a fraction xi of object i is placed into the knapsack, then a profit pi xi is earned.

* + The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
  + The idea is to calculate for each object the ratio of value/cost, and sort them according to this ratio. Then you take the objects with the highest ratios and add them until you can’t add the next object as whole. Finally add as much as you can of the next object.

So, for our example: v(weight) = {4, 2, 2, 1, 10}

c(profit) = {12, 1, 2, 1, 4}

r = {1/3, 2, 1, 1, 5/2}

From this it’s obvious that you should add the objects: 5, 2, 3, and 4 and then as much as possible of 1.

We can choose objects like this:

* + Added object 5 (10$, 4Kg) completely in the bag. Space left: 11.
  + Added object 2 (2$, 1Kg) completely in the bag. Space left: 10.
  + Added object 3 (2$, 2Kg) completely in the bag. Space left: 8.
  + Added object 4 (1$, 1Kg) completely in the bag. Space left: 7.
  + Added 58% (4$, 12Kg) of object 1 in the bag.
  + Filled the bag with objects worth 15.48$.

1. **0-1 knapsack problem**

The setup is the same, but the items may not be broken into smaller pieces, so we may decide either to take an item or to leave it (binary choice), but may not take a fraction of an item.

**5. Algorithm:**

1. **Fractional Knapsack Problem Using Greedy Approach**

Algorithm Knapsack (X, V, W, M)

//Description: Solve knapsack problem

//Input X: Array of n items.

V: Array of Profit associated with each item.

W: Array of Weight associated with each item.

M: Capacity of knapsack

//Output SW: Weight of selected items

SP: Profit of selected items

S 🡨 Ф // set of selected items, initially empty

SW🡨 0 // weight of selected items, initially zero

SP🡨 0 // profit of selected items, initially zero

i🡨1

while i<=n do

if (SW + W[i])<=M then

S🡨 S U X[i]

SW🡨 SW+ W[i]

SP🡨 SP + V[i]

else

frac = (M-SW)/W[i]

S🡨 S U X[i] \* frac

SW🡨 SW+ W[i] \* frac

SP🡨 SP + V[i] \* frac

end

i🡨 i+1

end

1. **Binary Knapsack Problem using Dynamic Programming**

Algorithm DP\_BIN\_Knapsack(*V, W, M*)

//Description: Solve Binary knapsack using DP

//Input: Set of item X, Set of Weight W, Profit of items V, and capacity M.

//Output: Array V, solution of problem.

for i🡨 1 to n do

V[i,0]🡨0

end

for i🡨 1 to M do

V[0,i]🡨0

end

for i🡨 1 to n do

for j🡨 0 to M do

if w[i] <=j

V[i,j] 🡨 max{V[i-1,j], V[i]+V[i-1, j-w[i]]}

else

V[i,j] 🡨 V[i-1,j]

end

end

end

Algorithm Track\_Knapsack(*W, V, M*)

SW🡨{ }

SP🡨{ }

i🡨n

j🡨M

while(j>0) do

if (V[i,j]==V([i-1,j] then

i🡨i-1

else

V[i,j] 🡨 V[i,j]-Vi

j🡨 j-w[i]

SW🡨SW +w[i]

SP🡨 SP +V[i]

end

end

**6. Complexity Analysis**

**A. Greedy Strategy: Fractional Knapsack Problem**

* The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
* If the items are already arranged in the required order, then while loop takes O(n) time.
* The average time complexity of Quick Sort is O(nlogn).
* Therefore, total time taken including the sort is O(nlogn).

**B. Dynamic Programming: Binary Knapsack Problem**

* It finds optimal solution by constructing table of size n\*M.
* Where n is no. of items and M is capacity of knapsack
* This table can be filled up in O(nM) time.
* Running time using DP is O(n\*M)

**Conclusion:**

Fractional knapsack using Greedy algorithm and 0/1 knapsack using dynamic programming is implemented successfully.