**Knapsack Problem:**

The **knapsack problem** is a problem in [combinatorial optimization](https://en.wikipedia.org/wiki/Combinatorial_optimization): Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size [knapsack](https://en.wikipedia.org/wiki/Knapsack) and must fill it with the most valuable items. The problem often arises in [resource allocation](https://en.wikipedia.org/wiki/Resource_allocation) where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

**Definition:**

The most common problem being solved is the **0-1 knapsack problem**, which restricts the number *{\displaystyle x\_{i}}* of copies of each kind of item to zero or one. Given a set of *{\displaystyle n}* items numbered from 1 up to *{\displaystyle n}*, each with a weight *{\displaystyle w\_{i}}* and a value *{\displaystyle v\_{i}}*, along with a maximum weight capacity *{\displaystyle W}*, maximize {\displaystyle \sum \_{i=1}^{n}v\_{i}x\_{i}} subject to {\displaystyle \sum \_{i=1}^{n}w\_{i}x\_{i}\leq W} and {\displaystyle x\_{i}\in \{0,1\}}.Here *{\displaystyle x\_{i}}* represents the number of instances of item *{\displaystyle i}* to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

The **bounded knapsack problem** (**BKP**) removes the restriction that there is only one of each item, but restricts the number {\displaystyle x\_{i}} of copies of each kind of item to a maximum non-negative integer value. {\displaystyle c}{\displaystyle \sum \_{i=1}^{n}v\_{i}x\_{i}}

The **unbounded knapsack problem** (**UKP**) places no upper bound on the number of copies of each kind of item and can be formulated as above except for that the only restriction on {\displaystyle x\_{i}} is that it is a non-negative integer.{\displaystyle x\_{i}\geq 0,\ x\_{i}\in \mathbb {Z} .}

A similar dynamic programming solution for the 0-1 knapsack problem also runs in pseudo-polynomial time. Assume {\displaystyle w\_{1},\,w\_{2},\,\ldots ,\,w\_{n},\,W} are strictly positive integers. Define {\displaystyle m[i,w]} to be the maximum value that can be attained with weight less than or equal to {\displaystyle w} using items up to {\displaystyle i} (first {\displaystyle i} items).

We can define {\displaystyle m[i,w]} recursively as follows: **(Definition A)**

* {\displaystyle m[0,\,w]=0}{\displaystyle m[i,\,w]=m[i-1,\,w]}The following is pseudocode for the dynamic program:

*// Input:*

*// Values (stored in array v)*

*// Weights (stored in array w)*

*// Number of distinct items (n)*

*// Knapsack capacity (W)*

*// NOTE: The array "v" and array "w" are assumed to store all relevant values starting at index 1.*

array m[0..n, 0..W];

**for** j from 0 to W **do**:

m[0, j] := 0

**for** i from 1 to n **do**:

m[i, 0] := 0

**for** i from 1 to n **do**:

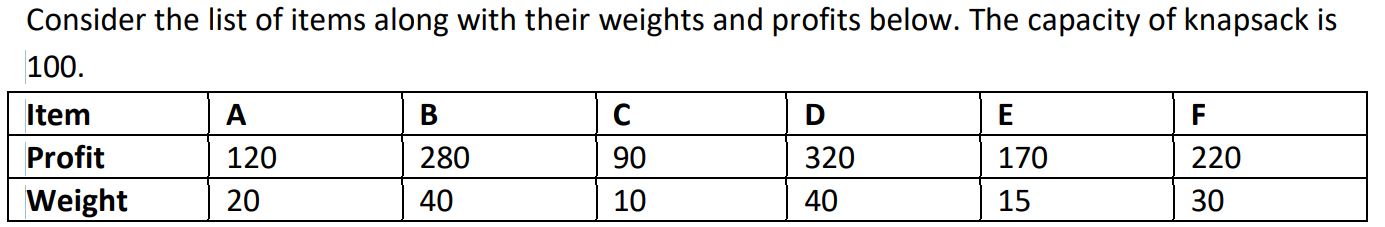
**for** j from 0 to W **do**:

**if** w[i] > j then:

m[i, j] := m[i-1, j]

**else**:

m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])



Total weight: 100

**We will use Max Profit Method**

First Pick D

Profit 320

Weight 40

Remaining weight 100-40= 60

Now we will pick B

Profit 280

Weight 40

Remaining weight 60-40= 20

Now we will Pick F

Profit 220\*20/30 = 146

Weight 20

**Total Products= D+B+F**

**Total Weight= 40 + 40 + 20 = 100**

**Total Profit= 320 + 280 + 146 = 746**

**Another Method using Greedy Method:**

**Ratios of Product’s Profit/Weight**

A=120/20= 6

B=280/40=7

C=90/10=9

D=320/40=8

E=170/15=11

F=220/30=7

We will choose the ones with the highest ratio first

Pick E First

Weight 15

Remaining Weight 85

Then Pick C

Weight 10

Remaining Weight 75

Now Pick D

Weight 40

Remaining weight 35

Now Pick F

Weight 30

Remaining Weight 5

Now Pick A

Weight 20/4 = 5

Profit 120/4 = 30

**Total Products: E + C + D + F +A**

**Total Profit: 170 + 90 + 320 + 220 +30 = 830**

**Total Weight: 15 + 10 + 40 + 30 + 5 = 100**

**Explain the time complexity of 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**](https://www.gatevidyalay.com/quick-sort-sorting-algorithms/) is O(nlogn).
* Therefore, total time taken including the sort is O(nlogn).