The Idea of Dynamic Programming

- Dynamic programming is a method for solving optimization problems.
- **The idea:** Compute the solutions to the sub sub-problems
- *Once* and store the solutions in a table, so that they can be reused (repeatedly) later.
- **Remark:** We trade space for time.

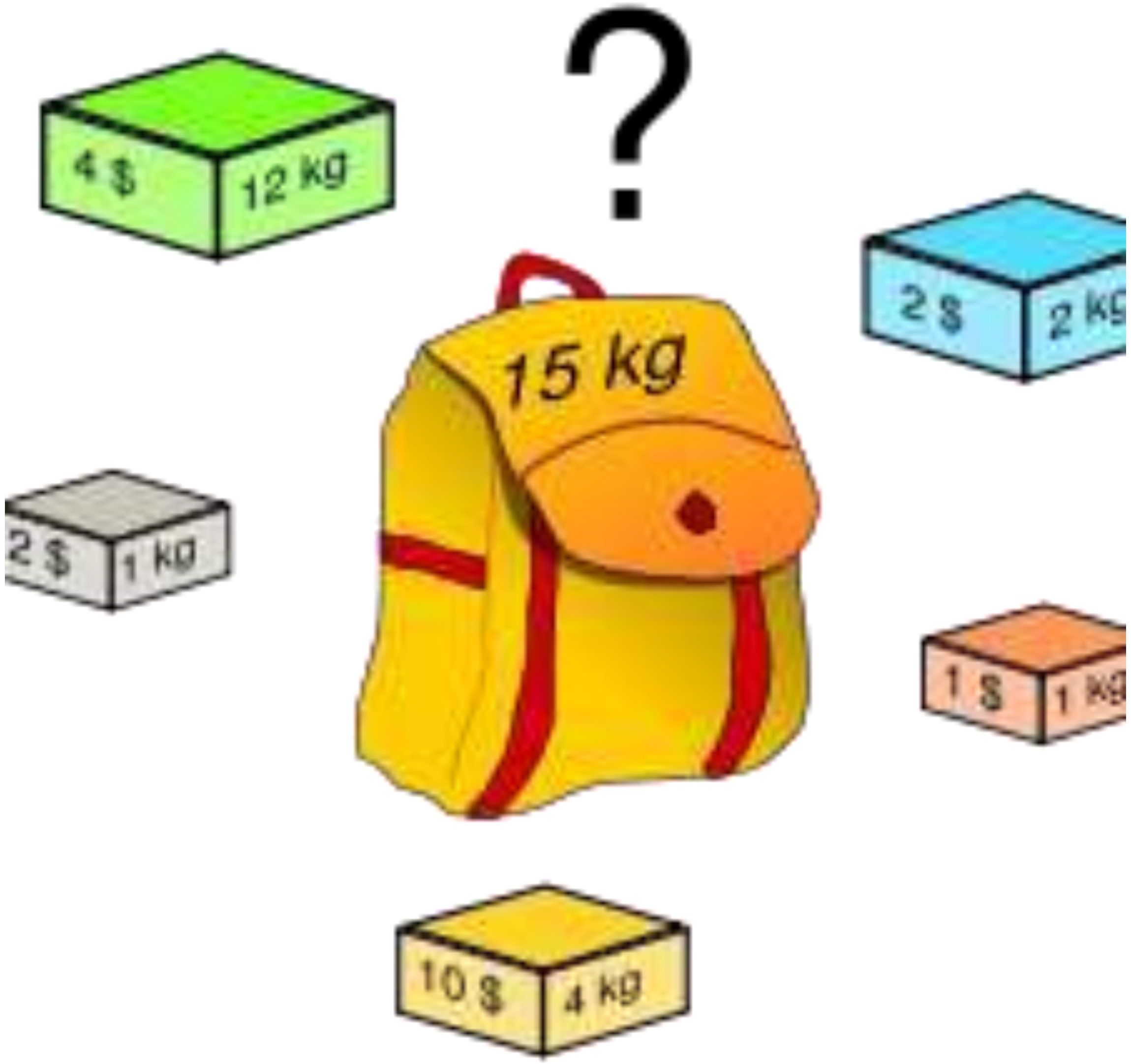
Knapsack Problems

Knapsack

- The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization. It derives its name from the following maximization problem of the best choice of essentials that can fit into one bag to be carried on a trip. 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 a given limit and the total value is as large as possible.

The Original Knapsack Problem (1)

- Problem Definition
  - Want to carry essential items in one bag
  - Given a set of items, each has
    - A cost (i.e., 12kg)
    - A value (i.e., 4$)
- Goal
  - To determine the # of each item to include in a collection so that
    - The total cost is less than some given cost
    - And the total value is as large as possible



The Original Knapsack Problem (2)

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

- Complexity Analysis
  - The general knapsack problem is known to be NP-hard
    - No polynomial-time algorithm is known for this problem
  - Here, we use greedy heuristics which cannot guarantee the optimal solution

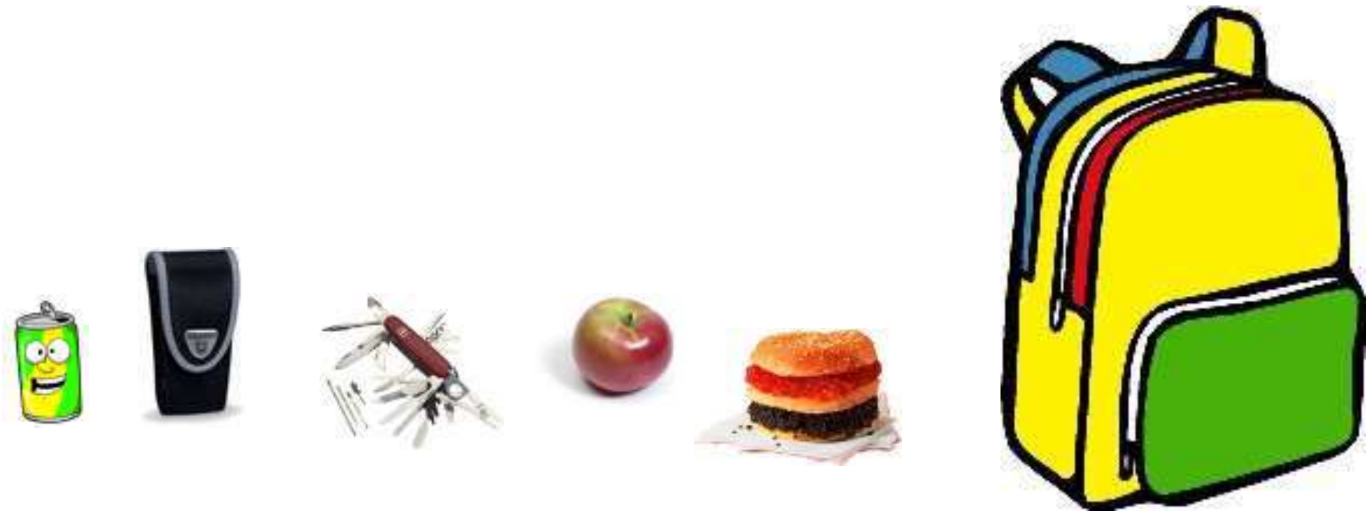
Fractional Knapsack Problem Using Greedy Method

- Fractional knapsack problem is solved using greedy method in the following steps-
- Step-01: For each item, compute its value / weight ratio.
- Step-02: Arrange all the items in decreasing order of their value / weight ratio.
- Step-03: Start putting the items into the knapsack beginning from the item with the highest ratio.

- Put as many items as you can into the knapsack.

0/1 Knapsack Problem (1)

- Problem: John wishes to take n items on a trip
  - The weight of item i is w_i & items are all different (0/1 Knapsack Problem)
  - The items are to be carried in a knapsack whose weight capacity is c
    - When sum of item weights ≤ c, all n items can be carried in the knapsack
    - When sum of item weights > c, some items must be left behind
- Which items should be taken/left?



0/1 knapsack problem is solved using dynamic programming in the following steps-

- **<u>Step-01:</u>**
- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0th row and 0th column with zeroes as shown-

	0	1	2	3		W
0	0	0	0	0	……	0
1	0					
2	0					
	……					
n	0					

T-Table

Step-02:

- Start filling the table row wise top to bottom from left to right.

Use the following formula-

$$\mathbf{T\,(i\,,\,j) = max\,\{\,T\,(\,i\text{-}1\,,\,j\,)\,,\,value_i + T(\,i\text{-}1\,,\,j - weight_i\,)\,\}}$$

- Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.
- This step leads to completely filling the table.
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step-03:

- To identify the items that must be put into the knapsack to obtain that maximum profit,
- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Example

For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

Step-01:

Draw a table say ‘T’ with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.

Fill all the boxes of 0^{th} row and 0^{th} column with 0.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

Finding T(1,1)-

We have,

$i = 1$

$j = 1$

$(value)_i = (value)_1 = 3$

$(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,1) = \max \{ T(1-1, 1), 3 + T(1-1, 1-2) \}$

$T(1,1) = \max \{ T(0,1), 3 + T(0,-1) \}$

$T(1,1) = T(0,1)$ { Ignore $T(0,-1)$ }

$T(1,1) = 0$

Finding T(1,2)-

We have,

$i = 1$

$j = 2$

$(value)_i = (value)_1 = 3$

$(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,2) = \max \{ T(1-1, 2), 3 + T(1-1, 2-2) \}$

$T(1,2) = \max \{ T(0,2), 3 + T(0,0) \}$

$T(1,2) = \max \{0, 3+0\}$

$T(1,2) = 3$

Step-03

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-

The last entry represents the **maximum possible value** that can be put into the knapsack.

So, maximum possible value that can be put into the **knapsack = 7.**

Step-04

We mark the rows labelled "1" and "2".
Thus, items that must be put into the knapsack to obtain the maximum value 7 are-
Item-1 and Item-2

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

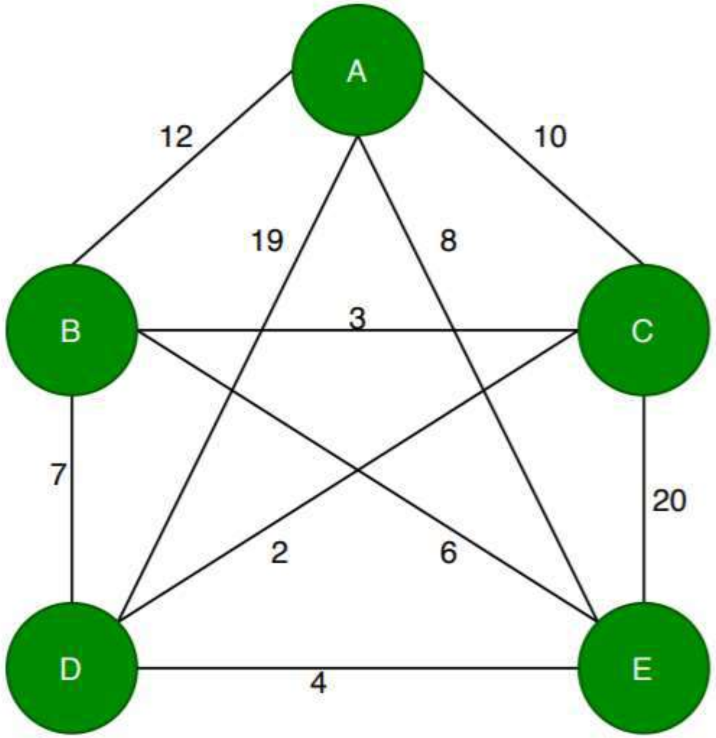
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

Travelling Salesman Problem (DP)

Traveling salesman problem is stated as,

"Given a set of n cities and distance between each pair of cities, find the minimum length path such that it covers each city exactly once and terminates the tour at starting city."



Algorithm for Traveling salesman problem

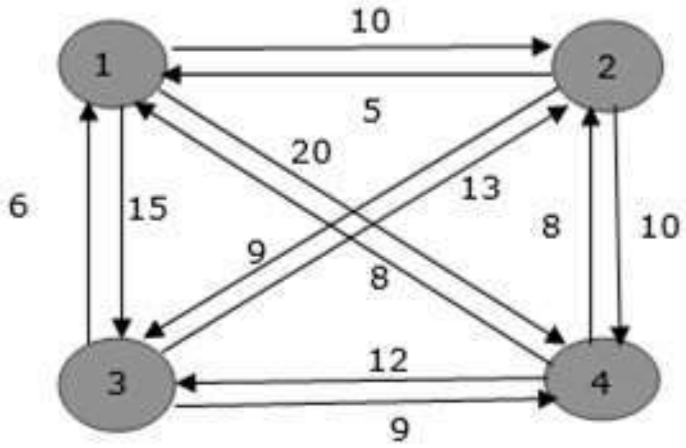
- **Step 1:**
  - Let d[i, j] indicates the distance between cities i and j.
  - Function C[x, V – { x }]is the cost of the path starting from city x.
  - V is the set of cities/vertices in given graph.
  - The aim of TSP is to minimize the cost function.
- **Step 2:**
  - Assume that graph contains n vertices V_1, V_2, ..., V_n.
  - TSP finds a path covering all vertices exactly once, and the same time it tries to minimize the overall traveling distance.

Algorithm Contd….

- **Step 3:**
  - Mathematical formula to find minimum distance is stated below:
    - **$C(i, V) = \min \{ d[i, j] + C(j, V - \{ j \}) \}$,** $j \in V$ and $i \notin V$.
- TSP problem possesses the principle of optimality, i.e. for $d[V_1, V_n]$ to be minimum, any intermediate path (V_i, V_j) must be minimum.

Problem

- **Solve the traveling salesman problem with the associated cost adjacency matrix using dynamic programming.**



Table

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Solution….

- Let us start our tour from city 1.

Step 1:

- **Initially, we will find the distance between city 1 and city {2, 3, 4, } without visiting any intermediate city.**
  - Cost(x, y, z) represents the distance from x to z and y as an intermediate city.
  - **Cost(2, Φ, 1) = d[2, 1] = 5**
  - **Cost(3, Φ, 1) = d[3, 1] = 6**
  - **Cost(4, Φ , 1) = d[4, 1] = 8**

Contd…

Step 2:

- In this step, we will find the minimum distance by visiting 1 city as intermediate city.
  - $C(i, V) = \min \{ d[i, j] + C(j, V - \{ j \}) \},$
  - $Cost\{2, \{3\}, 1\} = d[2, 3] + Cost(3, \Phi, 1) = 9 + 6 = 15$
  - $Cost\{2, \{4\}, 1\} = d[2, 4] + Cost(4, \Phi, 1) = 10 + 8 = 18$
  - $Cost\{3, \{2\}, 1\} = d[3, 2] + Cost(2, \Phi, 1) = 13 + 5 = 18$
  - $Cost\{3, \{4\}, 1\} = d[3, 4] + Cost(4, \Phi, 1) = 12 + 8 = 20$
  - $Cost\{4, \{3\}, 1\} = d[4, 3] + Cost(3, \Phi, 1) = 9 + 6 = 15$
  - $Cost\{4, \{2\}, 1\} = d[4, 2] + Cost(2, \Phi, 1) = 8 + 5 = 13$

Contd…

Step 3:

- **In this step, we will find the minimum distance by visiting 2 city as intermediate city.**

$C(i, V) = \min \{ d[i, j] + C(j, V - \{ j \}) \},$

- $Cost(2, \{3, 4\}, 1) = \min \{ d[2, 3] + Cost(3, \{4\}, 1), d[2, 4] + Cost(4, \{3\}, 1)]\}$
  $= \min \{ [9 + 20], [10 + 15] \}$
  $= \min \{29, 25\} = 25.$
- $Cost(3, \{2, 4\}, 1) = \min \{ d[3, 2] + Cost(2, \{4\}, 1), d[3, 4] + Cost(4, \{2\}, 1)]\}$
  $= \min \{ [13 + 18], [12 + 13] \}$
  $= \min \{31, 25\} = 25.$
- $Cost(4, \{2, 3\}, 1) = \min\{ d[4, 2] + Cost(2, \{3\}, 1), d[4, 3] + Cost(3, \{2\}, 1)]\}$
  $= \min \{ [8 + 15], [9 + 18] \}$
  $= \min \{23, 27\} = 23.$

Contd…

Step 4:

- In this step, we will find the minimum distance by visiting 3 city as intermediate city.
  - $C(i, V) = \min \{ d[i, j] + C(j, V - \{ j \}) \}$,
  - $Cost(1, \{2, 3, 4\}, 1) = \min \{ d[1, 2] + Cost(2, \{3, 4\}, 1), d[1, 3] + Cost(3, \{2, 4\}, 1), d[1, 4] + Cost(4, \{2, 3\}, 1)\}$

    $= \min \{ 10 + 25, 15 + 25, 20 + 23\}$

    $= \min\{35, 40, 43\} = 35.$
- Thus, minimum length tour would be of 35.
- Trace the path:
  - Let us find the path that gives the distance of 35.
  - Cost(1, {2, 3, 4}, 1) is minimum due to d[1, 2], so move from 1 to 2. Path = {1, 2}.
  - Cost(2, {3,4}, 1) is minimum due to d[2,4], so move from 2 to 4. Path = {1, 2, 4}.
  - Cost(4, {3}, 1) is minimum due to d[4, 3], so move from 4 to 3. Path = {1, 2, 4, 3}.
  - All cities are visited so come back to 1. Hence the optimum tour would be 1 – 2 – 4– 3 – 1.