Dynamic Programming

Definition

Dynamic programming (DP) is a general algorithm design technique for solving problems with overlapping sub-problems. This technique was invented by American mathematician "Richard Bellman" in 1950s.

Key Idea

The key idea is to save answers of overlapping smaller sub-problems to avoid re-computation.

Dynamic Programming Properties

- An instance is solved using the solutions for smaller instances.
- The solutions for a smaller instance might be needed multiple times, so store their results in a table.
- Thus each smaller instance is solved only once.
- Additional space is used to save time.

Dynamic Programming vs. Divide & Conquer

LIKE divide & conquer, dynamic programming solves problems by combining solutions to sub-problems. UNLIKE divide & conquer, sub-problems are NOT independent in dynamic programming.

Divide & Conquer	**Dynamic Programming**
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

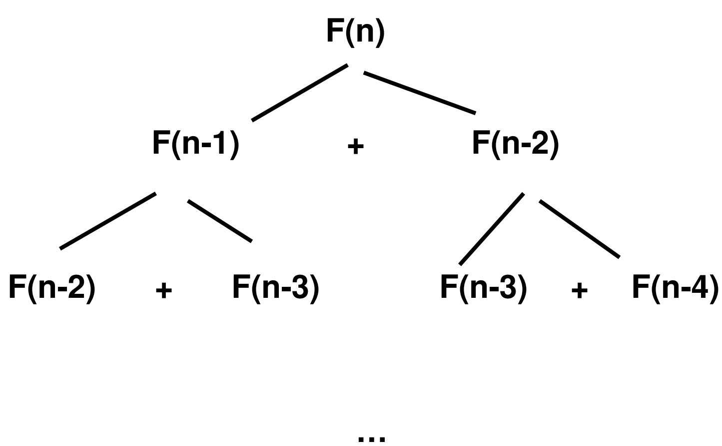
Dynamic Programming vs. Divide & Conquer: EXAMPLE
Computing Fibonacci Numbers

1. Using standard recursive formula:

$$F(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

Algorithm F(n)
// Computes the nth Fibonacci number recursively by using its definitions
// Input: A non-negative integer n
// Output: The nth Fibonacci number
if n==0 || n==1 then
    return n
else
    return F(n-1) + F(n-2)

Algorithm F(n): Analysis
- Is too expensive as it has repeated calculation of smaller Fibonacci numbers.
- Exponential order of growth.



2. **Using Dynamic Programming:**

Algorithm F(n)

```
// Computes the nth Fibonacci number by using dynamic programming method
// Input: A non-negative integer n
// Output: The nth Fibonacci number
A[0] ← 0
A[1] ← 1
for i ←2 to n do
        A[i] ← A[i-1] + A[i-2]
return A[n]
```

Algorithm F(n): Analysis

- Since it caches previously computed values, saves time from repeated computations of same sub-instance
- Linear order of growth

Rules of Dynamic Programming

1. **OPTIMAL SUB-STRUCTURE:** An optimal solution to a problem contains optimal solutions to sub-problems
2. **OVERLAPPING SUB-PROBLEMS:** A recursive solution contains a "small" number of distinct sub-problems repeated many times
3. **BOTTOM UP FASHION:** Computes the solution in a bottom-up fashion in the final step

Three basic components of Dynamic Programming solution

The development of a dynamic programming algorithm must have the following three basic components

1. A recurrence relation
2. A tabular computation
3. A backtracking procedure

Example Problems that can be solved using Dynamic Programming method

1. Computing binomial co-efficient
2. Compute the longest common subsequence
3. Warshall's algorithm for transitive closure
4. Floyd's algorithm for all-pairs shortest paths
5. Some instances of difficult discrete optimization problems like
    - knapsack problem
    - traveling salesperson problem

Binomial Co-efficient

Definition:

The binomial co-efficient C(n,k) is the number of ways of choosing a subset of k elements from a set of n elements.

1. Factorial definition

For non-negative integers n & k, we have

$$C(n,k) = n!/\ k!\ (n-k)!$$
$$= (n(n-1)\ldots (n-k+1))/\ k(k-1)\ldots 1 \qquad \text{if } k \in \{0,1,\ldots,n\}$$

and

$$C(n,k) = 0 \qquad \text{if } k > n$$

2. Recursive definition

$$C(n,k) = \begin{cases} 1 & \text{if } k=0 \\ 1 & \text{if } n = k \\ C(n-1,k-1) + C(n-1, k) & \text{if } n>k>0 \end{cases}$$

Solution

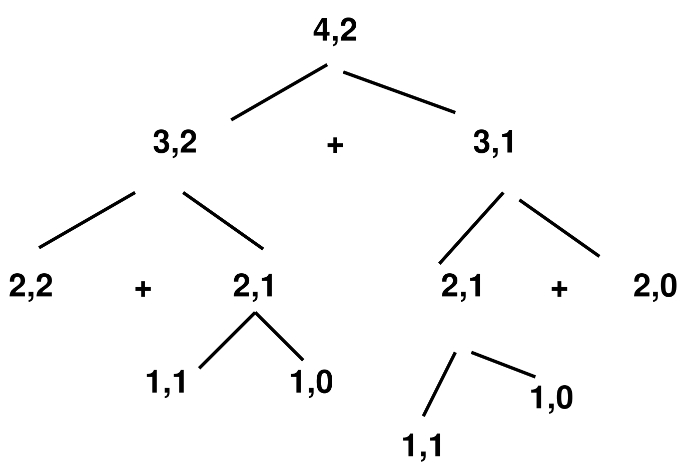
Using crude DIVIDE & CONQUER method we can have the algorithm as follows:

Algorithm binomial (n, k)

```
if k==0 OR k==n
        return 1
else
        return binomial(n-1, k) + binomial(n-1, k-1)
```

Algorithm binomial (n, k): Analysis

- Re-computes values large number of times
- In worst case (when k=n/2), we have O(2n/n) efficiency



Using DYNAMIC PROGRAMMING method: This approach stores the value of C(n,k) as they are computed i.e. Record the values of the binomial co-efficient in a table of n+1 rows and k+1 columns, numbered from 0 to n and 0 to k respectively.

Table for computing binomials is as follows:

	0	1	2	…	k-1	k
0	1					
1	1	1				
2						
…						
n-1					C(n-1, k-1)	C(n-1, k)
n						C(n, k)

Algorithm binomial (n, k)

```
// Computes C(n, k) using dynamic programming
// input: integers n≥k≥0
// output: The value of C(n, k)
for i ← 0 to n do
        for j ← 0 to min (i, k)  do
                if j == 0 or j == i then
                        A[i, j] ← 1
                else
                        A[i, j] ← A[i-1, j-1] + A[i-1, j]
return A[n, k]
```

Algorithm binomial (n, k): Analysis

- Input size: n, k
- Basic operation: Addition
- Let A(n, k) be the total number of additions made by the algorithm in computing C(n,k)
- The first k+1 rows of the table form a triangle while the remaining n-k rows form a rectangle. Therefore we have two parts in A(n,k).

Warshall's Algorithm
-to find TRANSITIVE CLOSURE

Some useful definitions:

- **Directed Graph:** A graph whose every edge is directed is called directed graph OR digraph
- **Adjacency matrix:** The adjacency matrix A = $\{a_{ij}\}$ of a directed graph is the boolean matrix that has
  - o 1 - if there is a directed edge from ith vertex to the jth vertex
  - o 0 - Otherwise
- **Transitive Closure:** Transitive closure of a directed graph with n vertices can be defined as the n-by-n matrix T={tij}, in which the elements in the ith row ($1 \le i \le n$) and the jth column($1 \le j \le n$) is 1 if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the ith vertex to the jth vertex, otherwise tij is 0.

  The transitive closure provides reach ability information about a digraph.

Computing Transitive Closure:

- We can perform DFS/BFS starting at each vertex
  - Performs traversal starting at the ith vertex.
  - Gives information about the vertices reachable from the ith vertex
  - **Drawback:** This method traverses the same graph several times.
  - **Efficiency** : (O(n(n+m))
- Alternatively, we can use dynamic programming: the Warshall's Algorithm

Underlying idea of Warshall's algorithm:

- Let A denote the initial boolean matrix.
- The element r(k) [i, j] in ith row and jth column of matrix Rk (k = 0, 1, …, n) is equal to 1 if and only if there exists a directed path from ith vertex to jth vertex with intermediate vertex if any, numbered not higher than k
- **Recursive Definition:**
  - **Case 1:**

    A path from *vi* to *vj* restricted to using only vertices from {*v*1,*v*2,…,*vk*} as intermediate vertices does not use *vk*, Then

    $$R(k) [i, j] = R(k-1) [i, j].$$
  - **Case 2:**

    A path from *vi* to *vj* restricted to using only vertices from {*v*1,*v*2,…,*vk*} as intermediate vertices do use vk. Then

    $$R(k) [i, j] = R(k-1) [i, k] \textbf{ AND } R(k-1) [k, j].$$

We conclude:

$$R(k)[i, j] = R(k-1) [i, j] \text{ OR } (R(k-1) [i, k] \text{ AND } R(k-1) [k, j])$$

NOTE:

- If an element rij is 1 in R(k-1), it remains 1 in R(k)
- If an element rij is 0 in R(k-1), it has to be changed to 1 in R(k) if and only if the element in its row I and column k and the element in its column j and row k are both 1's in R(k-1)

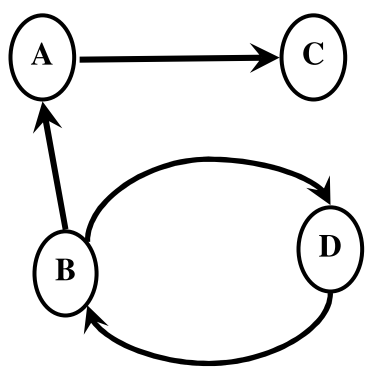
Algorithm:

Algorithm Warshall(A[1..n, 1..n])
// Computes transitive closure matrix
// Input: Adjacency matrix A
// Output: Transitive closure matrix R

```
R(0) ← A
for k ← 1 to n do
        for i ← 1 to n do
                for j ← 1 to n do
                        R(k)[i, j] ← R(k-1)[i, j] OR (R(k-1)[i, k] AND R(k-1)[k, j] )
return R(n)
```

Example:

Find Transitive closure for the given digraph using Warshall's algorithm.



Solution:

R(0) =

	A	B	C	D
A	0	0	1	0
B	1	0	0	1
C	0	0	0	0
D	0	1	0	0

<table>
<tr>
<td>R(0)</td>
<td>k = 1
Vertex 1 can be intermediate node</td>
<td>$\begin{array}{c|cccc|} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{B} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{C} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{D} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array}$

R1[2,3]
= R0[2,3] OR
R0[2,1] AND R0[1,3]
= 0 OR (1 AND 1)
= 1</td>
<td>$\begin{array}{c|cccc|} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{B} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{C} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{D} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array}$</td>
</tr>
<tr>
<td>R(1)</td>
<td>k = 2
Vertex {1,2 } can be intermediate nodes</td>
<td>$\begin{array}{c|cccc|} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{B} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{C} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{D} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array}$

R2[4,1]
= R1[4,1] OR
R1[4,2] AND R1[2,1]
= 0 OR (1 AND 1)
= 1

R2[4,3]
= R1[4,3] OR
R1[4,2] AND R1[2,3]
= 0 OR (1 AND 1)
= 1

R2[4,4]
= R1[4,4] OR
R1[4,2] AND R1[2,4]
= 0 OR (1 AND 1)
= 1</td>
<td>$\begin{array}{c|cccc|} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{B} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{C} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{D} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{array}$</td>
</tr>
<tr>
<td>R(2)</td>
<td>k = 3
Vertex {1,2,3 } can be intermediate nodes</td>
<td>$\begin{array}{c|cccc|} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{B} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{C} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{D} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{array}$</td>
<td>$\begin{array}{c|cccc|} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{B} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{C} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{D} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{array}$

NO CHANGE</td>
</tr>
</table>

<table>
<tr>
<td>R(3)</td>
<td>k = 4
Vertex
{1,2,3,4 }
can be
intermediate
nodes</td>
<td>$\begin{array}{c|cccc|} & A & B & C & D \\ \hline A & 0 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 0 \\ D & 1 & 1 & 1 & 1 \end{array}$

R4[2,2]
= R3[2,2] OR
R3[2,4] AND R3[4,2]
= 0 OR (1 AND 1)
= 1</td>
<td>$\begin{array}{c|cccc|} & A & B & C & D \\ \hline A & 0 & 0 & 1 & 0 \\ B & 1 & 1 & 1 & 1 \\ C & 0 & 0 & 0 & 0 \\ D & 1 & 1 & 1 & 1 \end{array}$</td>
</tr>
<tr>
<td>R(4)</td>
<td></td>
<td>$\begin{array}{c|cccc|} & A & B & C & D \\ \hline A & 0 & 0 & 1 & 0 \\ B & 1 & 1 & 1 & 1 \\ C & 0 & 0 & 0 & 0 \\ D & 1 & 1 & 1 & 1 \end{array}$</td>
<td>TRANSITIVE CLOSURE
for the given graph</td>
</tr>
</table>

Efficiency:

- Time efficiency is $\Theta(n^3)$
- Space efficiency: Requires extra space for separate matrices for recording intermediate results of the algorithm.

Floyd's Algorithm to find
-ALL PAIRS SHORTEST PATHS

Some useful definitions:

- **Weighted Graph:** Each edge has a weight (associated numerical value). Edge weights may represent costs, distance/lengths, capacities, etc. depending on the problem.
- **Weight matrix:** W(i,j) is
  - 0 if i=j
  - ∞ if no edge b/n i and j.
  - "weight of edge" if edge b/n i and j.

Problem statement:
Given a weighted graph G(V, Ew), the all-pairs shortest paths problem is to find the shortest path between every pair of vertices (vi, vj) Є V.

Solution:
A number of algorithms are known for solving All pairs shortest path problem

- **Matrix multiplication based algorithm**
- **Dijkstra's algorithm**
- **Bellman-Ford algorithm**
- **Floyd's algorithm**

Underlying idea of Floyd's algorithm:

- Let W denote the initial weight matrix.
- Let D(k) [i, j] denote cost of shortest path from i to j whose intermediate vertices are a subset of {1,2,…,k}.
- **Recursive Definition**
  Case 1:
  A shortest path from *vi* to *vj* restricted to using only vertices from {*v*1,*v*2,…,*vk*} as intermediate vertices does not use *vk*. Then
  $D(k)[i, j] = D(k\text{-}1)[i, j]$.
  Case 2:
  A shortest path from *vi* to *vj* restricted to using only vertices from {*v*1,*v*2,…,*vk*} as intermediate vertices do use vk. Then
  $D(k)[i, j] = D(k\text{-}1)[i, k] + D(k\text{-}1)[k, j]$.

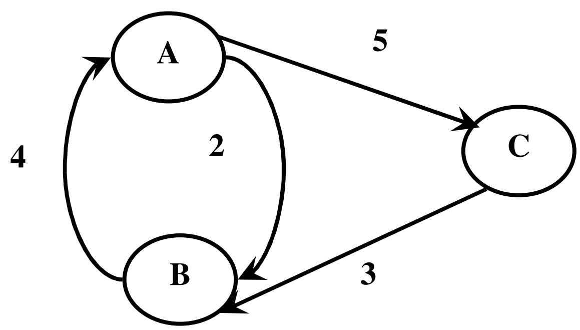
We conclude:
$D(k)[i, j] = \min \{ D(k\text{-}1)[i, j], D(k\text{-}1)[i, k] + D(k\text{-}1)[k, j] \}$

Algorithm:

```
Algorithm Floyd(W[1..n, 1..n])
// Implements Floyd's algorithm
// Input: Weight matrix W
// Output: Distance matrix of shortest paths' length
D ← W
for k ← 1 to n do
        for i ← 1 to n do
        for j ← 1 to n do
                D [ i, j] ← min { D [ i, j],  D [ i, k] +  D [ k, j]
return D
```

Example:

Find All pairs shortest paths for the given weighted connected graph using Floyd's algorithm.



Solution:

$$D(0) = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \left|\begin{array}{ccc} 0 & 2 & 5 \\ 4 & 0 & \infty \\ \infty & 3 & 0 \end{array}\right| \end{array}$$

<table>
<tr>
<td>D(0)</td>
<td>k = 1
Vertex 1 can be intermediate node</td>
<td>$$\begin{array}{c|ccc|} & A & B & C \\ A & 0 & 2 & 5 \\ B & 4 & 0 & \infty \\ C & \infty & 3 & 0 \end{array}$$
D1[2,3]
= min { D0 [2,3],
D0 [2,1] + D0 [1,3] }
= min { ∞, (4 + 5) }
= 9</td>
<td>$$\begin{array}{c|ccc|} & A & B & C \\ A & 0 & 2 & 5 \\ B & 4 & 0 & 9 \\ C & \infty & 3 & 0 \end{array}$$</td>
</tr>
<tr>
<td>D(1)</td>
<td>k = 2
Vertex 1,2 can be intermediate nodes</td>
<td>$$\begin{array}{c|ccc|} & A & B & C \\ A & 0 & 2 & 5 \\ B & 4 & 0 & 9 \\ C & \infty & 3 & 0 \end{array}$$
D2[3,1]
= min { D1 [3,1],
D1 [3,2] + D1 [2,1] }
= min { ∞, (4 + 3) }
= 7</td>
<td>$$\begin{array}{c|ccc|} & A & B & C \\ A & 0 & 2 & 5 \\ B & 4 & 0 & 9 \\ C & 7 & 3 & 0 \end{array}$$</td>
</tr>
<tr>
<td>D(2)</td>
<td>k = 3
Vertex 1,2,3 can be intermediate nodes</td>
<td>$$\begin{array}{c|ccc|} & A & B & C \\ A & 0 & 2 & 5 \\ B & 4 & 0 & 9 \\ C & 7 & 3 & 0 \end{array}$$</td>
<td>$$\begin{array}{c|ccc|} & A & B & C \\ A & 0 & 2 & 5 \\ B & 4 & 0 & 9 \\ C & 7 & 3 & 0 \end{array}$$
NO Change</td>
</tr>
<tr>
<td>D(3)</td>
<td></td>
<td>$$\begin{array}{c|ccc|} & A & B & C \\ A & 0 & 2 & 5 \\ B & 4 & 0 & 9 \\ C & 7 & 3 & 0 \end{array}$$</td>
<td>ALL PAIRS SHORTEST PATHS for the given graph</td>
</tr>
</table>

0/1 Knapsack Problem
Memory function

Definition:

Given a set of n items of known weights w1,…,wn and values v1,…,vn and a knapsack of capacity W, the problem is to find the most valuable subset of the items that fit into the knapsack.
Knapsack problem is an OPTIMIZATION PROBLEM

Dynamic programming approach to solve knapsack problem

Step 1:
Identify the smaller sub-problems. If items are labeled 1..n, then a sub-problem would be to find an optimal solution for Sk = {items labeled 1, 2, .. k}

Step 2:
Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Initial conditions:

$$V[0, j] = 0 \quad \text{for } j \geq 0$$
$$V[i, 0] = 0 \quad \text{for } i \geq 0$$

Recursive step:

$$V[i, j] = \begin{cases} \max \{ V[i-1, j], vi + V[i-1, j - wi] \} & \text{if } j - wi \geq 0 \\ V[i-1, j] & \text{if } j - wi < 0 \end{cases}$$

Step 3:
Bottom up computation using iteration

Question:
Apply bottom-up dynamic programming algorithm to the following instance of the knapsack problem Capacity W= 5

Item #	Weight (Kg)	Value (Rs.)
1	2	3
2	3	4
3	4	5
4	5	6

Solution:
Using dynamic programming approach, we have:

<table>
<tr><th>Step</th><th>Calculation</th><th>Table</th></tr>
<tr><td>1</td><td>Initial conditions:
$V[0, j] = 0$ for $j \geq 0$
$V[i, 0] = 0$ for $i \geq 0$</td><td>
<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
</table></td></tr>
<tr><td>2</td><td>W1 = 2,
Available knapsack capacity = 1
W1 > WA, CASE 1 holds:
$V[i, j] = V[i-1, j]$
$V[1,1] = V[0, 1] = 0$</td><td>
<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
</table></td></tr>
<tr><td>3</td><td>W1 = 2,
Available knapsack capacity = 2
W1 = WA, CASE 2 holds:
$V[i, j] = \max \{ V[i-1, j],$
$vi + V[i-1, j - wi] \}$
$V[1,2] = \max \{ V[0, 2],$
$3 + V[0, 0] \}$
$= \max \{ 0, 3 + 0 \} = 3$</td><td>
<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>3</td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
</table></td></tr>
<tr><td>4</td><td>W1 = 2,
Available knapsack capacity =
3,4,5
W1 < WA, CASE 2 holds:
$V[i, j] = \max \{ V[i-1, j],$
$vi + V[i-1, j - wi] \}$
$V[1,3] = \max \{ V[0, 3],$
$3 + V[0, 1] \}$
$= \max \{ 0, 3 + 0 \} = 3$</td><td>
<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>3</td><td>3</td><td>3</td><td>3</td></tr>
<tr><td>2</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
</table></td></tr>
<tr><td>5</td><td>W2 = 3,
Available knapsack capacity = 1
W2 >WA, CASE 1 holds:
$V[i, j] = V[i-1, j]$
$V[2,1] = V[1, 1] = 0$</td><td>
<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>3</td><td>3</td><td>3</td><td>3</td></tr>
<tr><td>2</td><td>0</td><td>0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
</table></td></tr>
</table>

6

W2 = 3,
Available knapsack capacity = 2
W2 >WA, CASE 1 holds:
V[i, j] = V[i-1, j]
V[2,2] = V[1, 2] = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

7

W2 = 3,
Available knapsack capacity = 3
W2 = WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[2,3] = max { V[1, 3],
4 +V[1, 0] }
= max { 3, 4 + 0 } = 4

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

8

W2 = 3,
Available knapsack capacity = 4
W2 < WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[2,4] = max { V[1, 4],
4 +V[1, 1] }
= max { 3, 4 + 0 } = 4

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

9

W2 = 3,
Available knapsack capacity = 5
W2 < WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[2,5] = max { V[1, 5],
4 +V[1, 2] }
= max { 3, 4 + 3 } = 7

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

10

W3 = 4,
Available knapsack capacity =
1,2,3
W3 > WA, CASE 1 holds:
V[i, j] = V[i-1, j]

V[i,j]	j=0	1	2	3	4	5
i=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		
4	0					

11

W3 = 4,
Available knapsack capacity = 4
W3 = WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[3,4] = max { V[2, 4],
5 +V[2, 0] }
= max { 4, 5 + 0 } = 5

V[i,j]	j=0	1	2	3	4	5
i=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	
4	0					

12

W3 = 4,
Available knapsack capacity = 5
W3 < WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[3,5] = max { V[2, 5],
5 +V[2, 1] }
= max { 7, 5 + 0 } = 7

V[i,j]	j=0	1	2	3	4	5
i=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					

13

W4 = 5,
Available knapsack capacity =
1,2,3,4
W4 < WA, CASE 1 holds:
V[i, j] = V[i-1, j]

V[i,j]	j=0	1	2	3	4	5
i=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	

14

W4 = 5,
Available knapsack capacity = 5
W4 = WA, CASE 2 holds:
V[i, j] = max { V[i-1, j],
vi +V[i-1, j - wi] }
V[4,5] = max { V[3, 5],
6 +V[3, 0] }
= max { 7, 6 + 0 } = 7

V[i,j]	j=0	1	2	3	4	5
i=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

Maximal value is V [4, 5] = 7/-

What is the composition of the optimal subset?

The composition of the optimal subset if found by tracing back the computations for the entries in the table.

<table>
<tr><th>Step</th><th>Table</th><th>Remarks</th></tr>
<tr><td>1</td><td>
<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>3</td><td>3</td><td>3</td><td>3</td></tr>
<tr><td>2</td><td>0</td><td>0</td><td>3</td><td>4</td><td>4</td><td>7</td></tr>
<tr><td>3</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr>
<tr><td>4</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr>
</table>
</td><td>V[4, 5] = V[3, 5]

→ ITEM 4 NOT included in the subset</td></tr>
<tr><td>2</td><td>
<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>3</td><td>3</td><td>3</td><td>3</td></tr>
<tr><td>2</td><td>0</td><td>0</td><td>3</td><td>4</td><td>4</td><td>7</td></tr>
<tr><td>3</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr>
<tr><td>4</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr>
</table>
</td><td>V[3, 5] = V[2, 5]

→ ITEM 3 NOT included in the subset</td></tr>
<tr><td>3</td><td>
<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>3</td><td>3</td><td>3</td><td>3</td></tr>
<tr><td>2</td><td>0</td><td>0</td><td>3</td><td>4</td><td>4</td><td>7</td></tr>
<tr><td>3</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr>
<tr><td>4</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr>
</table>
</td><td>V[2, 5] ≠ V[1, 5]

→ ITEM 2 included in the subset</td></tr>
<tr><td>4</td><td>Since item 2 is included in the knapsack:
Weight of item 2 is 3kg, therefore, remaining capacity of the knapsack is
(5 - 3 =) 2kg

<table>
<tr><th>V[i,j]</th><th>j=0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>3</td><td>3</td><td>3</td><td>3</td></tr>
<tr><td>2</td><td>0</td><td>0</td><td>3</td><td>4</td><td>4</td><td>7</td></tr>
<tr><td>3</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr>
<tr><td>4</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr>
</table>
</td><td>V[1, 2] ≠ V[0, 2]

→ ITEM 1 included in the subset</td></tr>
<tr><td>5</td><td>Since item 1 is included in the knapsack:
Weight of item 1 is 2kg, therefore, remaining capacity of the knapsack is
(2 - 2 =) 0 kg.</td><td>Optimal subset: { item 1, item 2 }

Total weight is: 5kg (2kg + 3kg)
Total profit is: 7/- (3/- + 4/-)</td></tr>
</table>

Efficiency:

- Running time of Knapsack problem using dynamic programming algorithm is: O(n * W)
- Time needed to find the composition of an optimal solution is: O(n + W)

Memory function

- Memory function combines the strength of top-down and bottom-up approaches
- It solves ONLY sub-problems that are necessary and does it ONLY ONCE.

The method:

- Uses top-down manner.
- Maintains table as in bottom-up approach.
- Initially, all the table entries are initialized with special “null” symbol to indicate that they have not yet been calculated.
- Whenever a new value needs to be calculated, the method checks the corresponding entry in the table first:
- If entry is NOT “null”, it is simply retrieved from the table.
- Otherwise, it is computed by the recursive call whose result is then recorded in the table.

Algorithm:

```
Algorithm MFKnap( i, j )
if V[ i, j] < 0
        if j < Weights[ i ]
                value ← MFKnap( i-1, j )
        else
                value ← max {MFKnap( i-1, j ),
                             Values[i] + MFKnap( i-1, j - Weights[i] )}
        V[ i, j ] ← value
return V[ i, j]
```

Example:

Apply memory function method to the following instance of the knapsack problem
Capacity W= 5

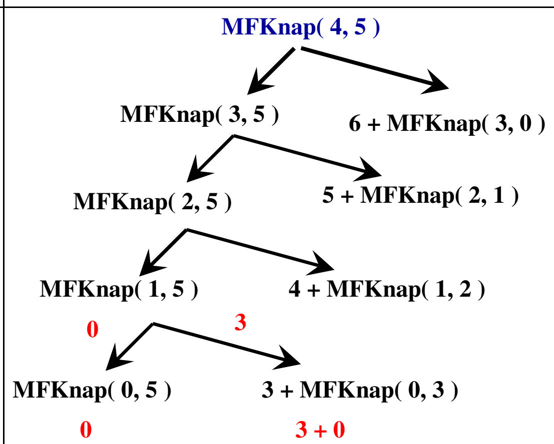
Item #	Weight (Kg)	Value (Rs.)
1	2	3
2	3	4
3	4	5
4	5	6

Solution:
Using memory function approach, we have:

	Computation	Remarks
1	Initially, all the table entries are initialized with special “null” symbol to indicate that they have not yet been calculated. Here null is indicated with -1 value.	(table below)

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	-1	-1	-1	-1	-1
2	0	-1	-1	-1	-1	-1
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

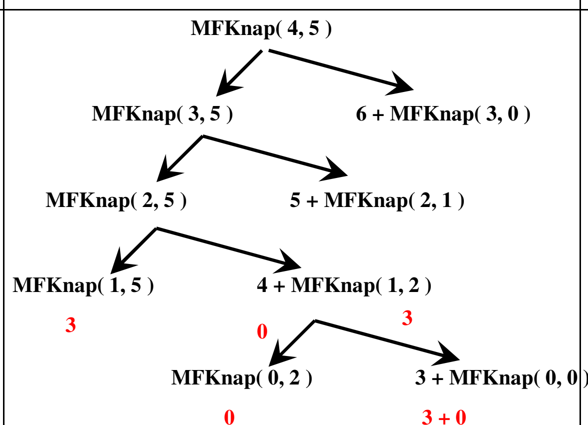
2



V[1, 5] = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	-1	-1	-1	-1	3
2	0	-1	-1	-1	-1	-1
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

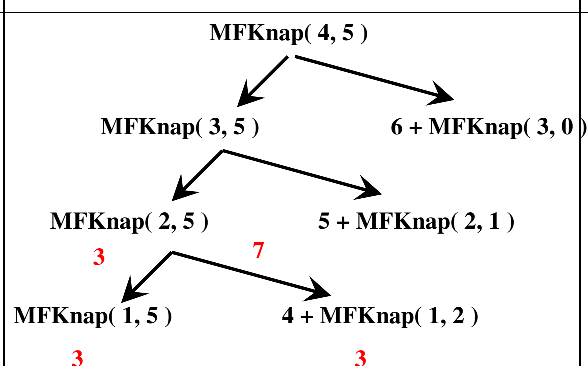
3



V[1, 2] = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	-1	3	-1	-1	3
2	0	-1	-1	-1	-1	-1
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

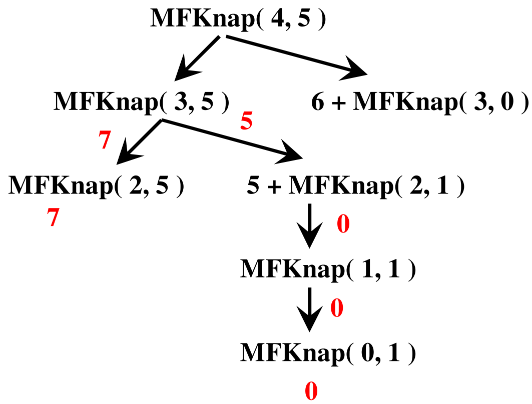
4



V[2, 5] = 7

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	-1	3	-1	-1	3
2	0	-1	-1	-1	-1	7
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

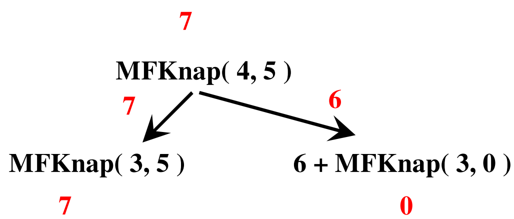
5



V[2, 1] = 0
V[3, 5] = 7

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	**0**	**3**	**-1**	**-1**	**3**
2	0	**0**	**-1**	**-1**	**-1**	**7**
3	0	**-1**	**-1**	**-1**	**-1**	**7**
4	0	**-1**	**-1**	**-1**	**-1**	**-1**

6



V[4, 5] = 7

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	**0**	**3**	**-1**	**-1**	**3**
2	0	**0**	**-1**	**-1**	**-1**	**7**
3	0	**-1**	**-1**	**-1**	**-1**	**7**
4	0	**-1**	**-1**	**-1**	**-1**	**7**

The composition of the optimal subset if found by **tracing back the computations for the entries in the table as done with the early knapsack problem**

Conclusion:
Optimal subset: { item 1, item 2 }

Total weight is: 5kg (2kg + 3kg)
Total profit is: 7/- (3/- + 4/-)

Efficiency:

- Time efficiency same as bottom up algorithm: O(n * W) + O(n + W)
- Just a constant factor gain by using memory function
- Less space efficient than a space efficient version of a bottom-up algorithm