Unit 3	Dynamic Programming	
A	Overview, Difference between dynamic programming and divide and conquer, All pair shortest path problems: Floyd-Warshall Algorithm	CO1,CO2,CO3, CO4
B	Applications and analysis: Matrix Chain Multiplication,	CO1, CO2, CO3, CO4
C	Applications and analysis: Longest Common sub-sequence, 0/1 Knapsack Problem	CO1,CO2,CO3

Dynamic Programming

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decisionsequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations,* that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulationsof some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of theoptimal solution.
- Perform a trace back step in which the solution itself is constructed.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamicprogramming produces all possible sub-problems at most once, one of which guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered

The divide and conquer principle solve a large problem, by breaking it up into smallerproblems which can be solved independently. In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems. Care is to be taken to avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity.In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.
Two difficulties may arise in any application of dynamic programming:

1. It may not always be possible to combine the solutions of smaller problems toform the solution of a larger one.
2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

MULTI STAGE GRAPHS

A multistage graph G = (V, E) is a directed graph in which the vertices are partitioned into k ≥ 2 disjoint sets V_i, 1 ≤ i ≤ k. In addition, if <u, v> is an edge in E,then u ∈ V_i and v ∈ V_{i+1} for some i, 1 ≤ i < k.

Let the vertex 's' is the source, and 't' the sink. Let c (i, j) be the cost of edge <i, j>. The cost of a path from 's' to 't' is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from 's' to 't'. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from 's' to't' starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k – 2 decisions. The i^{th} decision involves determining which vertex in v_{i+1}, 1 ≤ i ≤ k - 2, is to be on the path. Let c (i, j) be the cost of the path from source to destination. Then using the forward approach, we obtain:

$$cost(i, j) = \min_{\substack{l \in V_{i+1} \\ <j, l> \in E}} \{c(j, l) + cost(i + 1, l)\}$$

ALGORITHM:

Algorithm Fgraph (G, k, n, p)

```
// The input is a k-stage graph G = (V, E) with n vertices
// indexed in order or stages. E is a set of edges and c [i, j]
// is the cost of (i, j). p [1 : k] is a minimum cost path.
{
        cost [n] := 0.0;
        for j:= n - 1 to 1 step – 1 do
        {                                                   // compute cost [j]
                let r be a vertex such that (j, r) is an edgeof G and c
                [j, r] + cost [r] is minimum; cost [j] := c [j, r] + cost [r];
                d [j] := r:
        }
        p [1] := 1; p [k]  := n;                            // Find a minimum cost path.for j :=
        2 to k - 1 do p [j] := d [p [j - 1]];
}
```

The multistage graph problem can also be solved using the backward approach. Let bp(i, j) be a minimum cost path from vertex s to j vertex in V_i. Let Bcost(i, j) be the cost of bp(i, j). From the backward approach we obtain:

Bcost (i, j) = min { Bcost (i –1, l) + c (l, j)}l ∈ V_{i-1}
<l, j> ∈ E

Algorithm Bgraph (G, k, n, p)

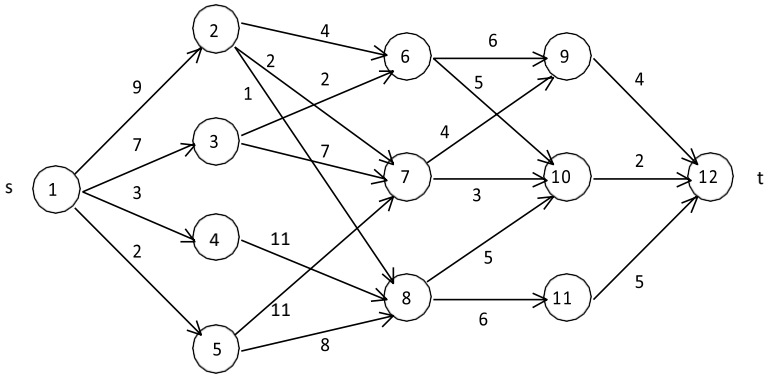
```
// Same function as Fgraph
{
        Bcost [1] := 0.0;
        for j := 2 to n do
        {                                                   // Compute Bcost [j].
                Let r be such that (r, j) is an edge of G and
                Bcost [r] + c [r, j] is minimum;Bcost [j] := Bcost
                [r] + c [r, j];
                D [j] := r;
        }                                                   //find a minimum cost path
        p [1] := 1; p [k] := n;
        for j:= k - 1 to 2 do p [j] := d [p [j + 1]];
}
```

Complexity Analysis:

The complexity analysis of the algorithm is fairly straightforward. Here, if G has |E| edges, then the time for the first for loop is Θ (|V| +|E|).

EXAMPLE 1:

Find the minimum cost path from s to t in the multistage graph of five stages shownbelow. Do this first using forward approach and then using backward approach.



FORWARD APPROACH:

We use the following equation to find the minimum cost path from s to t:cost (i, j) = min {c

(j, l) + cost (i + 1, l)}
l ∈ V_{i+1}

<j, l> ∈ E

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3), c (1, 4) + cost (2,4),
c (1, 5) + cost (2, 5)}
= min {9 + cost (2, 2), 7 + cost (2, 3), 3 + cost (2, 4), 2 + cost (2,5)}

Now first starting with,

cost (2, 2) = min{c (2, 6) + cost (3, 6), c (2, 7) + cost (3, 7), c (2, 8) + cost (3,8)}
= min {4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3,8)}

---

cost (3, 6) = min {c (6, 9) + cost (4, 9), c (6, 10) + cost (4, 10)}
= min {6 + cost (4, 9), 5 + cost (4, 10)}

cost (4, 9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0) = 4

cost (4, 10) = min {c (10, 12) + cost (5, 12)} = 2

Therefore, cost (3, 6) = min {6 + 4, 5 + 2} = 7

---

cost (3, 7) = min {c (7, 9) + cost (4, 9) , c (7, 10) + cost (4, 10)}
= min {4 + cost (4, 9), 3 + cost (4, 10)}

cost (4, 9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0} = 4

Cost (4, 10) = min {c (10, 2) + cost (5, 12)} = min {2 + 0} =2

Therefore, cost (3, 7) = min {4 + 4, 3 + 2} = min {8, 5} = 5

---

cost (3, 8) = min {c (8, 10) + cost (4, 10), c (8, 11) + cost (4,11)}
= min {5 + cost (4, 10), 6 + cost (4 + 11)}

cost (4, 11) = min {c (11, 12) + cost (5, 12)} =5

Therefore, cost (3, 8) = min {5 + 2, 6 + 5} = min {7, 11} = 7

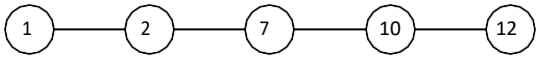
Therefore, cost (2, 2) = min {4 + 7, 2 + 5, 1 + 7} = min {11, 7, 8} = 7

Therefore, cost (2, 3) = min {c (3, 6) + cost (3, 6), c (3, 7) + cost (3,7)}
= min {2 + cost (3, 6), 7 + cost (3, 7)}
= min {2 + 7, 7 + 5} = min {9, 12} = 9

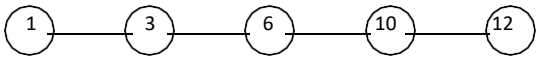
cost (2, 4) = min {c (4, 8) + cost (3, 8)} = min {11 + 7} = 18
cost (2, 5) = min {c (5, 7) + cost (3, 7), c (5, 8) + cost (3, 8)}
= min {11 + 5, 8 + 7} = min {16, 15} = 15

Therefore, cost (1, 1) = min {9 + 7, 7 + 9, 3 + 18, 2 +15}
= min {16, 16, 21, 17} = 16

The minimum cost path is 16.

The path is 

or



BACKWARD APPROACH:

We use the following equation to find the minimum cost path from t tos:Bcost (i, J) = min

{Bcost (i – 1, l) + c (l,J)}
l ∈ V_{i-1}
<l, j> ∈ E

Bcost (5, 12) = min {Bcost (4, 9) + c (9, 12), Bcost (4, 10) + c (10, 12),
Bcost (4, 11) + c (11, 12)}
= min {Bcost (4, 9) + 4, Bcost (4, 10) + 2, Bcost (4, 11) +5}

Bcost (4, 9) = min {Bcost (3, 6) + c (6, 9), Bcost (3, 7) + c (7,9)}
= min {Bcost (3, 6) + 6, Bcost (3, 7) + 4}

Bcost (3, 6) = min {Bcost (2, 2) + c (2, 6), Bcost (2, 3) + c (3,6)}
= min {Bcost (2, 2) + 4, Bcost (2, 3) + 2}

Bcost (2, 2) = min {Bcost (1, 1) + c (1, 2)} = min {0 + 9} = 9

Bcost (2, 3) = min {Bcost (1, 1) + c (1, 3)} = min {0 + 7} = 7

Bcost (3, 6) = min {9 + 4, 7 + 2} = min {13, 9} = 9

Bcost (3, 7) = min {Bcost (2, 2) + c (2, 7), Bcost (2, 3) + c (3,7),
Bcost (2, 5) + c (5, 7)}

Bcost (2, 5) = min {Bcost (1, 1) + c (1, 5)} = 2

Bcost (3, 7) = min {9 + 2, 7 + 7, 2 + 11} = min {11, 14, 13} = 11

Bcost (4, 9) = min {9 + 6, 11 + 4} = min {15, 15} = 15

Bcost (4, 10) = min {Bcost (3, 6) + c (6, 10), Bcost (3, 7) + c (7,10),
Bcost (3, 8) + c (8, 10)}

Bcost (3, 8) = min {Bcost (2, 2) + c (2, 8), Bcost (2, 4) + c (4,8),
Bcost (2, 5) + c (5, 8)}
Bcost (2, 4) = min {Bcost (1, 1) + c (1, 4)} = 3

Bcost (3, 8) = min {9 + 1, 3 + 11, 2 + 8} = min {10, 14, 10} = 10

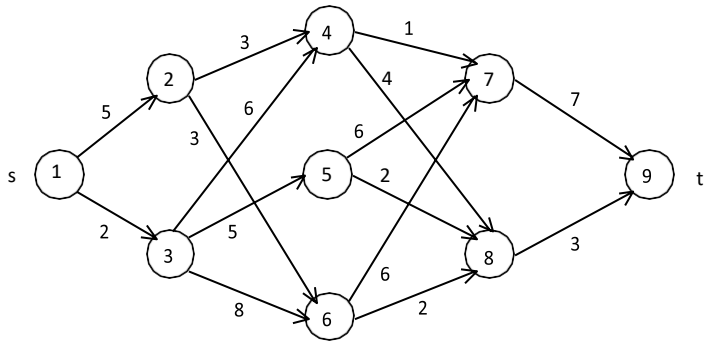
Bcost (4, 10) = min {9 + 5, 11 + 3, 10 + 5} = min {14, 14, 15) = 14

Bcost (4, 11) = min {Bcost (3, 8) + c (8, 11)} = min {Bcost (3, 8) +6}
= min {10 + 6} = 16

Bcost (5, 12) = min {15 + 4, 14 + 2, 16 + 5} = min {19, 16, 21} = 16.

EXAMPLE 2:

Find the minimum cost path from s to t in the multistage graph of five stages shownbelow. Do this first using forward approach and then using backward approach.



SOLUTION:

FORWARD APPROACH:

cost (i, J) = min {c (j, l) + cost (i + 1, l)}
$l \in V_{i+1}$
$\langle J, l \rangle \in E$

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3)}
= min {5 + cost (2, 2), 2 + cost (2, 3)}

cost (2, 2) = min {c (2, 4) + cost (3, 4), c (2, 6) + cost (3, 6)}
= min {3+ cost (3, 4), 3 + cost (3, 6)}

cost (3, 4) = min {c (4, 7) + cost (4, 7), c (4, 8) + cost (4, 8)}
= min {(1 + cost (4, 7), 4 + cost (4, 8)}

cost (4, 7) = min {c (7, 9) + cost (5, 9)} = min {7 + 0) = 7

cost (4, 8) = min {c (8, 9) + cost (5, 9)} = 3

Therefore, cost (3, 4) = min {8, 7} = 7

cost (3, 6) = min {c (6, 7) + cost (4, 7), c (6, 8) + cost (4, 8)}
= min {6 + cost (4, 7), 2 + cost (4, 8)} = min {6 + 7, 2 + 3} = 5

Therefore, cost (2, 2) = min {10, 8} = 8

cost (2, 3) = min {c (3, 4) + cost (3, 4), c (3, 5) + cost (3, 5), c (3, 6) + cost (3,6)}

cost (3, 5) = min {c (5, 7) + cost (4, 7), c (5, 8) + cost (4, 8)}= min {6 + 7, 2 +3} = 5

Therefore, cost (2, 3) = min {13, 10, 13} = 10

cost (1, 1) = min {5 + 8, 2 + 10} = min {13, 12} = 12

BACKWARD APPROACH:

$$Bcost(i, J) = \min_{\substack{l \in V_{i-1} \\ <l, j> \in E}} \{Bcost(i-1, l) + c(l, J)\}$$

$$Bcost(5, 9) = \min \{Bcost(4, 7) + c(7, 9), Bcost(4, 8) + c(8,9)\}$$
$$= \min \{Bcost(4, 7) + 7, Bcost(4, 8) + 3\}$$

$$Bcost(4, 7) = \min \{Bcost(3, 4) + c(4, 7), Bcost(3, 5) + c(5, 7), Bcost(3, 6) + c(6, 7)\}$$
$$= \min \{Bcost(3, 4) + 1, Bcost(3, 5) + 6, Bcost(3, 6) + 6\}$$

$$Bcost(3, 4) = \min \{Bcost(2, 2) + c(2, 4), Bcost(2, 3) + c(3,4)\}$$
$$= \min \{Bcost(2, 2) + 3, Bcost(2, 3) + 6\}$$

$Bcost(2, 2) = \min \{Bcost(1, 1) + c(1, 2)\} = \min \{0 + 5\} = 5$

$Bcost(2, 3) = \min (Bcost(1, 1) + c(1, 3)\} = \min \{0 + 2\} = 2$

Therefore, $Bcost(3, 4) = \min \{5 + 3, 2 + 6\} = \min \{8, 8\} = 8$

$Bcost(3, 5) = \min \{Bcost(2, 3) + c(3, 5)\} = \min \{2 + 5\} = 7$

$Bcost(3, 6) = \min \{Bcost(2, 2) + c(2, 6), Bcost(2, 3) + c(3,6)\}$
$= \min \{5 + 5, 2 + 8\} = 10$

Therefore, $Bcost(4, 7) = \min \{8 + 1, 7 + 6, 10 + 6\} = 9$

$Bcost(4, 8) = \min \{Bcost(3, 4) + c(4, 8), Bcost(3, 5) + c(5,8), Bcost(3, 6) + c(6, 8)\}$
$= \min \{8 + 4, 7 + 2, 10 + 2\} = 9$

Therefore, $Bcost(5, 9) = \min \{9 + 7, 9 + 3\} = 12$

All pairs shortest paths

In the all pairs shortest path problem, we are to find a shortest path between everypair of vertices in a directed graph G. That is, for every pair of vertices (i, j), we areto find a shortest path from i to j as well as one from j to i. These two paths are thesame when G is undirected.

When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.

The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires O (n^2) time, the matrix A can be obtained in O (n^3) time.

The dynamic programming solution, called Floyd's algorithm, runs in O (n^3) time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).

The shortest i to j path in G, i ≠ j originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from kto j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let A^k (i, j) represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:

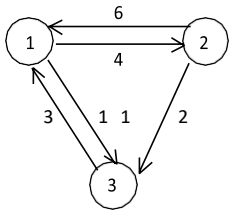
$$A^k(i, j) = \{\min_{1 \le k \le n} \{\min \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$$

Algorithm All Paths (Cost, A, n)
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path fromvertex
// i to vertex j. cost [i, i] = 0.0, for 1 ≤ i ≤ n.
{
for i := 1 to n do
for j:= 1 to n do
A [i, j] := cost [i, j]; // copy cost into A.for k
:= 1 to n do
for i := 1 to n do
for j := 1 to n do
A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
}

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. Therefore, the algorithm has a complexity of O (n^3).

Example 1:

Given a weighted digraph G = (V, E) with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are nocycles with zero or negative cost.



Cost adjacency matrix (A^0) = $\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$

General formula: $\min_{1 \le k \le n} \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$

Solve the problem for different values of k = 1, 2 and 3

Step 1: Solving the equation for, k = 1;

$A^1(1, 1) = \min \{(A^o (1, 1) + A^o (1, 1)), c (1, 1)\} = \min \{0 + 0, 0\} = 0$

$A^1(1, 2) = \min \{(A^o (1, 1) + A^o (1, 2)), c (1, 2)\} = \min \{(0 + 4), 4\} = 4$

$A^1(1, 3) = \min \{(A^o (1, 1) + A^o (1, 3)), c (1, 3)\} = \min \{(0 + 11), 11\} = 11$

$A^1(2, 1) = \min \{(A^o (2, 1) + A^o (1, 1)), c (2, 1)\} = \min \{(6 + 0), 6\} = 6$

$A^1(2, 2) = \min \{(A^o (2, 1) + A^o (1, 2)), c (2, 2)\} = \min \{(6 + 4), 0)\} = 0$

$A^1(2, 3) = \min \{(A^o (2, 1) + A^o (1, 3)), c (2, 3)\} = \min \{(6 + 11), 2\} = 2$

$A^1(3, 1) = \min \{(A^o (3, 1) + A^o (1, 1)), c (3, 1)\} = \min \{(3 + 0), 3\} = 3$

$A^1(3, 2) = \min \{(A^o (3, 1) + A^o (1, 2)), c (3, 2)\} = \min \{(3 + 4), ?\} = 7$

$A^1(3, 3) = \min \{(A^o (3, 1) + A^o (1, 3)), c (3, 3)\} = \min \{(3 + 11), 0\} = 0$

$$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 2: Solving the equation for, K = 2;

$A^2(1, 1) = \min \{(A^1(1, 2) + A^1 (2, 1), c (1, 1)\} = \min \{(4 + 6), 0\} = 0$

$A^2(1, 2) = \min \{(A^1(1, 2) + A^1 (2, 2), c (1, 2)\} = \min \{(4 + 0), 4\} = 4$

$A^2(1, 3) = \min \{(A^1(1, 2) + A^1 (2, 3), c (1, 3)\} = \min \{(4 + 2), 11\} = 6$

$A^2(2, 1) = \min \{(A (2, 2) + A (2, 1), c (2, 1)\} = \min \{(0 + 6), 6\} = 6$

$A^2(2, 2) = \min \{(A (2, 2) + A (2, 2), c (2, 2)\} = \min \{(0 + 0), 0\} = 0$

$A^2(2, 3) = \min \{(A (2, 2) + A (2, 3), c (2, 3)\} = \min \{(0 + 2), 2\} = 2$

$A^2(3, 1) = \min \{(A (3, 2) + A (2, 1), c (3, 1)\} = \min \{(7 + 6), 3\} = 3$

$A^2(3, 2) = \min \{(A (3, 2) + A (2, 2), c (3, 2)\} = \min \{(7 + 0), 7\} = 7$

$A^2(3, 3) = \min \{(A (3, 2) + A (2, 3), c (3, 3)\} = \min \{(7 + 2), 0\} = 0$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 3: Solving the equation for, k = 3;

$A^3(1, 1) = \min \{A^2(1, 3) + A^2(3, 1), c (1, 1)\} = \min \{(6 + 3), 0\} = 0$

$A^3(1, 2) = \min \{A^2(1, 3) + A^2(3, 2), c (1, 2)\} = \min \{(6 + 7), 4\} = 4$

$A^3(1, 3) = \min \{A^2(1, 3) + A^2(3, 3), c (1, 3)\} = \min \{(6 + 0), 6\} = 6$

$A^3(2, 1) = \min \{A^2(2, 3) + A^2(3, 1), c (2, 1)\} = \min \{(2 + 3), 6\} = 5$

$A^3(2, 2) = \min \{A^2(2, 3) + A^2(3, 2), c (2, 2)\} = \min \{(2 + 7), 0\} = 0$

$A^3(2, 3) = \min \{A^2(2, 3) + A^2(3, 3), c (2, 3)\} = \min \{(2 + 0), 2\} = 2$

$A^3(3, 1) = \min \{A^2(3, 3) + A^2(3, 1), c (3, 1)\} = \min \{(0 + 3), 3\} = 3$

$A^3(3, 2) = \min \{A^2(3, 3) + A^2(3, 2), c (3, 2)\} = \min \{(0 + 7), 7\} = 7$

$A^3(3, 3) = \min \{A^2(3, 3) + A^2(3, 3), c(3, 3)\} = \min \{(0 + 0), 0\} = 0$

$$A^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

TRAVELLING SALESPERSON PROBLEM

Let G = (V, E) be a directed graph with edge costs C_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \infty$ if < i, j> ∉ E. Let |V| = n and assume n > 1. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and endsat vertex 1.

Let g (i, S) be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function g (1, V − {1}) is the length ofan optimal salesperson tour. From the principal of optimality it follows that:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{--} \quad 1$$

Generalizing equation 1, we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \quad \text{--} \quad 2$$

The Equation can be solved for g (1, V − 1}) if we know g (k, V − {1, k}) for allchoices of k.

The Equation can be solved for g (1, V – 1}) if we know g (k, V – {1, k}) for allchoices of k

Complexity Analysis:

For each value of |S| there are n − 1 choices for i. The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$.

Hence, the total number of g (i, S)'s to be computed before computing g (1, V − {1}) is:

$$\sum_{k=0}^{n-1} (n-1)\binom{n-2}{k}$$

To calculate this sum, we use the binominal theorem:

$$(n-1)\left[\binom{(n-2)}{0} + \binom{(n-2)}{1} + \binom{(n-2)}{2} + \cdots + \binom{(n-2)}{(n-2)}\right]$$

According to the binominal theorem:

$$\left[\binom{(n-2)}{0} + \binom{(n-2)}{1} + \binom{(n-2)}{2} + \cdots + \binom{(n-2)}{(n-2)}\right] = 2^{n-2}$$

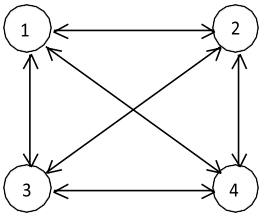
Therefore,

$$\sum_{k=0}^{n-1} (n-1)\binom{n-2}{k} = (n-1)2^{n-2}$$

This is Φ (n 2^{n-2}), so there are exponential number of calculate. Calculating one g (i, S) require finding the minimum of at most n quantities. Therefore, the entire algorithm is Φ (n^2 2^{n-2}). This is better than enumerating all n! different tours to find the best one. So, we have traded on exponential growth for a much smaller exponential growth. The most serious drawback of this dynamic programming solution is the space needed, which is O (n 2^n). This is too large even for modest values of n.

Example 1:

For the following graph find minimum cost tour for the traveling salespersonproblem:



The cost adjacency matrix = $\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, K\})\} \quad - \quad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\} \quad - \quad (2)$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \le i \le n$. So,

$g(2, \Phi) = C_{21} = 5$

$g(3, \Phi) = C_{31} = 6$

$g(4, \Phi) = C_{41} = 8$

Using equation – (2) we obtain:

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}, c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$

$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$
$= \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$

$g(3, \{4\}) = \min \{c_{34} + g(4, \Phi)\} = 12 + 8 = 20$

$g(4, \{3\}) = \min \{c_{43} + g(3, \Phi)\} = 9 + 6 = 15$

Therefore, $g(2, \{3, 4\}) = \min \{9 + 20, 10 + 15\} = \min \{29, 25\} = 25$

$g(3, \{2, 4\}) = \min \{(c_{32} + g(2, \{4\}), (c_{34} + g(4, \{2\})\}$

$g(2, \{4\}) = \min \{c_{24} + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min \{c_{42} + g(2, \Phi)\} = 8 + 5 = 13$

Therefore, $g(3, \{2, 4\}) = \min \{13 + 18, 12 + 13\} = \min \{41, 25\} = 25$

$g(4, \{2, 3\}) = \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$

$g(2, \{3\}) = \min \{c_{23} + g(3, \Phi\} = 9 + 6 = 15$

$g(3, \{2\}) = \min \{c_{32} + g(2, \Phi\} = 13 + 5 = 18$

Therefore, $g(4, \{2, 3\}) = \min \{8 + 15, 9 + 18\} = \min \{23, 27\} = 23$

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$
$= \min \{10 + 25, 15 + 25, 20 + 23\} = \min \{35, 40, 43\} = 35$

The optimal tour for the graph has length = 35

The optimal tour is: 1, 2, 4, 3, 1.

Matrix Chain Multiplication

Given the dimension of a sequence of matrices in an array **arr[]**, where the dimension of the **ith** matrix is **(arr[i-1] * arr[i])**, the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum. When two matrices of size **m*n** and **n*p** when multiplied, they generate a matrix of size **m*p** and the number of multiplications performed are **m*n*p**.

Examples:

Input: *arr[] = {2, 1, 3, 4}*

Output: *20*

Explanation: *There are 3 matrices of dimensions 2×1, 1×3, and 3×4,*
Let the input 3 matrices be M1, M2, and M3. There are two ways to multiply ((M1 x M2) x M3) and (M1 x (M2 x M3)),
Please note that the result of M1 x M2 is a 2 x 3 matrix and result of (M2 x M3) is a 1 x 4 matrix.
((M1 x M2) x M3) requires (2 x 1 x 3) + (0) + (2 x 3 x 4) = 30
(M1 x (M2 x M3)) requires (0) + (1 x 3 x 4) + (2 x 1 x 4) = 20
The minimum of these two is 20.

Input: *arr[] = {1, 2, 3, 4, 3}*

Output: *30*

Explanation: *There are 4 matrices of dimensions 1×2, 2×3, 3×4, 4×3. Let the input 4 matrices be M1, M2, M3 and M4. The minimum number of multiplications are obtained by ((M1M2)M3)M4. The minimum number is 1*2*3 + 1*3*4 + 1*4*3 = 30*

Input: *arr[] = {3, 4}*

Output: *0*

Explanation: *As there is only one matrix so, there is no cost of multiplication.*

Using Recursion:

We can solve the problem using recursion based on the following facts and observations:
*Now, for a given chain of **n** matrices, the first partition can be done in **N-1** ways. For example, sequence of matrices M1, M2, M3 and M4 can be grouped as (M1)(M2 x M3 x M4), (M1x M2) x (M3 x M4) or ((M1 x M2 x M3) x M4) in these 3 ways.*
*For **n** matrices M1, M2, ….. Mn. we can put the first bracket **n-1** ways*
(M1) x (M2 x M3 x M4 …… Mn-1 x Mn)
(M1 x M2) x (M3 x M4……. Mn-1 x Mn)
(M1 x M2 x M3) x (M4 ……. Mn-1 x Mn)
……
……
(M1 x M2 x M3 x M4 …….Mn-1) x (Mn)
*We put the first bracket at different **n-1** places and then recursively call for the two parts. At the end, we return the minimum of all partitions.*
*To write a recursive function we use a range of indexes **[i, j].** And run a loop for **k = i+1** to **j**. For **k = i+1,** we put the first bracket after the first matrix which has dimensions **arr[i] x arr[i+1]** and before the remaining matrices which have dimenstions **arr[i+1] x arr[i+2], arr[i+2] x arr[i+3], ……. arr[j-1] x arr[j]***

- *For every **k**, we make two subproblems : (a) Chain from **i** to **k** (b) Chain from **k** to **j***
- *Each of the subproblems can be further partitioned into smaller subproblems and we can find the total required multiplications by solving for each of the groups.*
- *The base case would be when we have only one matrix left which means when **j** is equal to **i+1**.*

Longest Common sub-sequence

Given two strings, **s1** and **s2**, the task is to find the length of the Longest Common Subsequence. If there is no **common subsequence**, return 0.
A **subsequence** is a string generated from the original string by deleting 0 or more characters and without

changing the relative order of the remaining characters. For example , subsequences of “ABC” are “”, “A”, “B”, “C”, “AB”, “AC”, “BC” and “ABC”.
In general a string of length **n** has **2n** subsequences.
Examples:
Input: *s1 = “ABC”, s2 = “ACD”*

Output: *2*
Explanation: *The longest subsequence which is present in both strings is “AC”.*
Input: *s1 = “AGGTAB”, s2 = “GXTXAYB”*
Output: *4*
Explanation: *The longest common subsequence is “GTAB”.*
Input: *s1 = “ABC”, s2 = “CBA”*
Output: *1*
Explanation: *There are three longest common subsequences of length 1, “A”, “B” and “C”.*

*The idea is to compare the last characters of **s1** and **s2**. While comparing the strings **s1** and **s2** two cases arise:*

1. ***Match** : Make the recursion call for the remaining strings (strings of lengths **m-1** and **n-1**) and add 1 to result.*
2. ***Do not Match** : Make two recursive calls. First for lengths **m-1** and **n**, and second for **m** and **n-1.** Take the maximum of two results.*

***Base case** : If any of the strings become empty, we return 0.*

For example, consider the input strings s1 = “ABX” and s2 = “ACX”.
LCS(“ABX”, “ACX”) = 1 + LCS(“AB”, “AC”) [Last Characters Match]
LCS(“AB”, “AC”) = max(LCS(“A”, “AC”) , LCS(“AB”, “A”)) [Last Characters Do Not Match]
LCS(“A”, “AC”) = max(LCS(“”, “AC”) , LCS(“A”, “A”)) = max(0, 1 + LCS(“”, “”)) = 1
LCS(“AB”, “A”) = max(LCS(“A”, “A”) , LCS(“AB”, “”)) = max(1 + LCS(“”, “”, 0)) = 1
*So overall result is 1 + 1 **= 2***

0/1 – KNAPSACK

We are given n objects and a knapsack. Each object i has a positive weight w_i and a positive value V_i. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x_1, x_2, ... , x_n. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order x_n, x_{n-1}, ...x_1. Following a decision on x_n, we may be in one of two possible states: the capacity remaining in $m - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions x_{n-1}, , x_1 must be
optimal with respect to the problem state resulting from the decision on x_n. Otherwise, x_n, , x_1 will not be optimal. Hence, the principal of optimality holds.

$$F_n(m) = \max \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \quad -- \quad 1$$

For arbitrary $f_i(y)$, $i > 0$, this equation generalizes to:

$$F_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \quad -- \quad 2$$

Equation-2 can be solved for $f_n(m)$ by beginning with the knowledge $f_o(y) = 0$ for all y and $f_i(y) = -\infty$, $y < 0$. Then

$f_1, f_2, \ldots f_n$ can be successively computed using equation–2.

When the w_i's are integer, we need to compute $f_i(y)$ for integer y, $0 \leq y \leq m$. Since f_i $(y) = -\infty$ for $y < 0$, these function values need not be computed explicitly. Since each f_i can be computed from $f_i - 1$ in $\Theta(m)$ time, it takes $\Theta(m\ n)$ time to compute f_n. When the wi's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit $\Theta(m\ n)$ computation of f_n may not be the most efficient computation. So, we explore **an alternative method for both cases.**

The $f_i(y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1$ $< y_2 < \ldots . < y_k$, such that $f_i(y_1) < f_i(y_2) < \ldots . . < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; f_i $(y) = f(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y \leq y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) \mid 1 \leq j \leq k\}$ to represent $f_i(y)$. Each number of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 =$
$\{(0, 0)\}$. We can compute S^{i+1} from S_i by first computing:

$$S^i \overline{+} \{(P, W) \mid (P - p, W - w_i) \in S^i\}$$ $_i$

Now, S^{i+1} can be computed by merging the pairs in S^i and $S^i{}_1$ together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Example 1:

Consider the knapsack instance n = 3, $(w_1, w_2, w_3) = (2, 3, 4)$, $(P_1, P_2, P_3) = (1, 2, 5)$ and M = 6.

Solution:

Initially, $f_o(x) = 0$, for all x and $f_i(x) = -\infty$ if $x < 0$. $F_n(M) = \max\{f_{n-1}$

$(M), f_{n-1}(M - w_n) + p_n\}$

$F_3(6) = \max(f_2(6), f_2(6 - 4) + 5\} = \max\{f_2(6), f_2(2) + 5\}$

$F_2(6) = \max(f_1(6), f_1(6 - 3) + 2\} = \max\{f_1(6), f_1(3) + 2\}$

$F_1(6) = \max(f_0(6), f_0(6 - 2) + 1\} = \max\{0, 0 + 1\} = 1$

$F_1(3) = \max(f_0(3), f_0(3 - 2) + 1\} = \max\{0, 0 + 1\} = 1$

Therefore, $F_2(6) = \max(1, 1 + 2\} = 3$

$F_2(2) = \max(f_1(2), f_1(2 - 3) + 2\} = \max\{f_1(2), -\infty + 2\}$

$F_1(2) = \max(f_0(2), f_0(2 - 2) + 1\} = \max\{0, 0 + 1\} = 1$

$F_2(2) = \max\{1, -\infty + 2\} = 1$

Finally, $f_3(6) = \max\{3, 1 + 5\} = 6$

Other Solution:

For the given data we have:

$S^0 = \{(0, 0)\}$; $S^0 = \{(1, 2)\}$

$S^1 = (S^0 \cup S^0) = \{(0, 0), (1,2)\}$

X - 2 = 0 => x = 2.	y – 3 = 0 => y = 3
X - 2 = 1 => x = 3.	y – 3 = 2 => y = 5

$S^{11} = \{(2, 3), (3, 5)\}$

$S^2 = (S^1 \cup S^{11}) = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$

X – 5 = 0 => x = 5.	y – 4 = 0 => y = 4
X – 5 = 1 => x = 6.	y – 4 = 2 => y = 6
X – 5 = 2 => x = 7.	y – 4 = 3 => y = 7
X – 5 = 3 => x = 8.	y – 4 = 5 => y = 9

$S^{21} = \{(5, 4), (6, 6), (7, 7), (8, 9)\}$

$S^3 = (S^2 \cup S^2) = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8,9)\}$

By applying Dominance rule,

$S^3 = (S^2 \cup S^2) = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)\}$

From (6, 6) we can infer that the maximum Profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$