Unit III
3.Dynamic Programming

3.1 General Method

- Dynamic programming is a name, coined by Richard Bellman in 1955.
- Dynamic programming is a technique for solving problems with **overlapping sub problems**.
- Typically, these sub problems arise from a recurrence relating a given problem's solution to solutions of its smaller sub problems.
- Rather than solving overlapping sub problems again and again, dynamic programming suggests solving each of the smaller sub problems only once and recording the results in a table from which a solution to the original problem can then be obtained.
- The Dynamic programming can be used when the solution to a problem can be viewed as the result of **sequence of decisions**.
- Dynamic programming is based on the principle of optimality.
- 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 sub sequences.
- Since the principle of optimality may not hold for some formulations of 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 the optimal 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 dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal.

3.2 0/1 KNAPSACK Problem

- We are given n objects and a knapsack. Each object i has a positive weight wi and a positive value Vi.
- 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 $x_1, x_2, \ldots, x_n$.
- 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 $x_n, x_{n-1}, \ldots x_1$. 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 $x_{n-1}, \ldots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n.
- otherwise, $x_n, \ldots, x_1$ will not be optimal. Hence, the principal of optimality holds.

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

- For arbitrary fi (y), i > 0, this equation generalizes to:
  $F_i(y) = \max\{f_i\text{-}1(y), f_i\text{-}1(y - w_i) + p_i\}$ -- 2
- Equation-2 can be solved for $f_n(m)$ by beginning with the knowledge $f_o(y) = 0$ for all y and $f_i(y) = -\alpha$, $y < 0$. Then $f_1, f_2, \ldots f_n$ can be successively computed using equation–2.
- 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 w_i are real numbers, $f_i(y)$ is needed for real numbers y such that $0 < y < m$.
- f_i cannot be explicitly computed for all y in this range.
- Even when the w_i are integer, the explicit $\Theta(m\ n)$ computation of fn may not be the most efficient computation.
- compute only $f_i(y_j)$, $1 < j < k$.
- Use the ordered set $S^i = \{(f(y_j), y_j) \mid 1 < j < k\}$ to represent $f_i(y)$. Each number of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$.
  $S^0 = \{(0, 0)\}$.
- Compute S^{i+1} from S^i by first computing:
  $S^i_1 = \{(P, W) \mid (P - p_i, W - w_i)\ \eta\ \epsilon\ S^i\}$

- S^{i+1} can be computed by merging the pairs in S^i and S^i_1 together.
- if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j < P_k$ and $W_j > W_k$, then the pair (P_j, Wj) 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).

```
Algorithm DKP(p, w, n, m)
{
    S^0 := {(0, 0)};
    for i := 1 to n − 1 do
    {
        S_1^{i−1} := {(P, W) | (P − p_i, W − w_i) ∈ S^{i−1} and W ≤ m};
        S^i := MergePurge(S^{i−1}, S_1^{i−1});
    }
    (PX, WX) := last pair in S^{n−1};
    (PY, WY) := (P' + p_n, W' + w_n) where W' is the largest W in
        any pair in S^{n−1} such that W + w_n ≤ m;
    // Trace back for x_n, x_{n−1}, ..., x_1.
    if (PX > PY) then x_n := 0;
    else x_n := 1;
    TraceBackFor(x_{n−1}, ..., x_1);
}
```

```
PW = record {float p; float w; }

Algorithm DKnap(p, w, x, n, m)
{
    // pair[ ] is an array of PW's.
    b[0] := 1; pair[1].p := pair[1].w := 0.0; // S^0
    t := 1; h := 1; // Start and end of S^0
    b[1] := next := 2; // Next free spot in pair[ ]
    for i := 1 to n - 1 do
    { // Generate S^i.
        k := t;
        u := Largest(pair, w, t, h, i, m);
        for j := t to u do
        { // Generate S_1^{i-1} and merge.
            pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
                // (pp, ww) is the next element in S_1^{i-1}.
            while ((k ≤ h) and (pair[k].w ≤ ww)) do
            {
                pair[next].p := pair[k].p;
                pair[next].w := pair[k].w;
                next := next + 1; k := k + 1;
            }
            if ((k ≤ h) and (pair[k].w = ww)) then
            {
                if pp < pair[k].p then pp := pair[k].p;
                k := k + 1;
            }
            if pp > pair[next - 1].p then
            {
                pair[next].p := pp; pair[next].w := ww;
                next := next + 1 ;
            }
            while ((k ≤ h) and (pair[k].p ≤ pair[next - 1].p))
                do k := k + 1;
        }
        // Merge in remaining terms from S^{i-1}.
        while (k ≤ h) do
        {
            pair[next].p := pair[k].p; pair[next].w := pair[k].w;
            next := next + 1; k := k + 1;
        }
        // Initialize for S^{i+1}.
        t := h + 1; h := next - 1; b[i + 1] := next;
    }
    TraceBack(p, w, pair, x, m, n);
}
```

Example 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 $m = 6$. For these data we have

$$
\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0),(1,2)\}; S_1^1 = \{(2,3),(3,5)\} \\
S^2 &= \{(0,0),(1,2),(2,3),(3,5)\}; S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\} \\
S^3 &= \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}
\end{aligned}
$$

By applying purge rule

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

Optimal solution (x_1,x_2,x_3)=(1,0,1)

3.3 SINGLE SOURCE SHORTEST PATH-GENERAL WEIGHTS

- Also known as bellman ford algorithm.
- Find the shortest path from a vertex to all other vertices of a weighted graph.

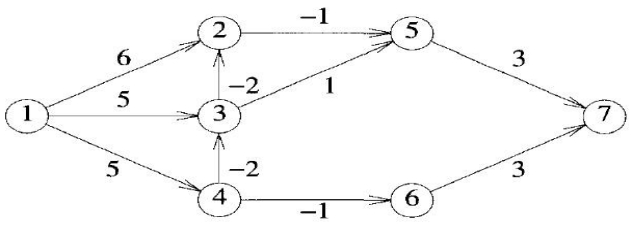
- It is similar to Dijkstra's algorithm but it can work with graphs in which some edges can have negative weights.
- Negative weight edges can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction.
- Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices.
- Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.
- In each of these iterations, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances.

Algorithm:

Algorithm BellmanFord(*list* vertices, *list* edges, *vertex* source) **is**

```
    // This implementation takes in a graph, represented as
    // lists of vertices (represented as integers [0..n-1]) and edges,
    // and fills two arrays (distance and predecessor) holding
    // the shortest path from the source to each vertex
    distance := list of size n
    predecessor := list of size n
    // Step 1: initialize graph
    for each vertex v in vertices do
        distance[v] := inf                  // Initialize the distance to all vertices to
infinity
        predecessor[v] := null              // And having a null predecessor
      distance[source] := 0                 // The distance from the source to itself is,
                                                of course, zero
    // Step 2: relax edges repeatedly
      repeat |V|−1 times:
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v] := distance[u] + w
                predecessor[v] := u
    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            error "Graph contains a negative-weight cycle"
    return distance, predecessor
```

- Bellman–Ford runs in O(V.E)



(a) A directed graph

	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$