Visvesvaraya Technological University (VTU)

Subject Code:21CS42

Subject : DESIGN AND ANALYSIS OF ALGOROTHM

Created By:

Hanumanthu

Dep. Of CSE

Visit Our Official Website It is Available @ any time

http://searchcreators.org/

MODULE-04

Dynamic Programming

Chapter-I

4.1. General Method with Example

- Dynamic Programming is invented by U.S Mathematician Richard Bellman in 1950.
- Dynamic Programming is a technique for solving Problems with overlapping subproblems.
- In this method each sub-problem is solved only once.
- Dynamic Programming Is typically applied for optimization problem.
- For each given problem we may get any number of solutions we seek for optimum solution.

Example of Dynamic Programming

1. Multistage Graph
2. 0/1 Knapsack Problem
3. All Pairs shortest path algorithm
4. Traveling Salesman Problem
5. Reliability Design
6. Optimal Binary Search Trees

4.2. Multistage Graph

- A multistage graph G = (V, El which is a directed graph.
- In this graph all the vertices are partitioned into the k stages where **$k \geq 2$.**
- In multistage graph problem we have to find the shortest path from source to sink.
- The cost of each path is calculated by using the weight given along that edge.
- The cost of a path from source (denoted by S) to sink (denoted by T) is the sum of the costs of edges on the path.
- In multistage graph problem we have to find the path from S to T. There is set of vertices in each stage.
- The multistage graph can be solved using forward and backward approach.

Let us solve multistage problem for both the approaches with the help of some example.

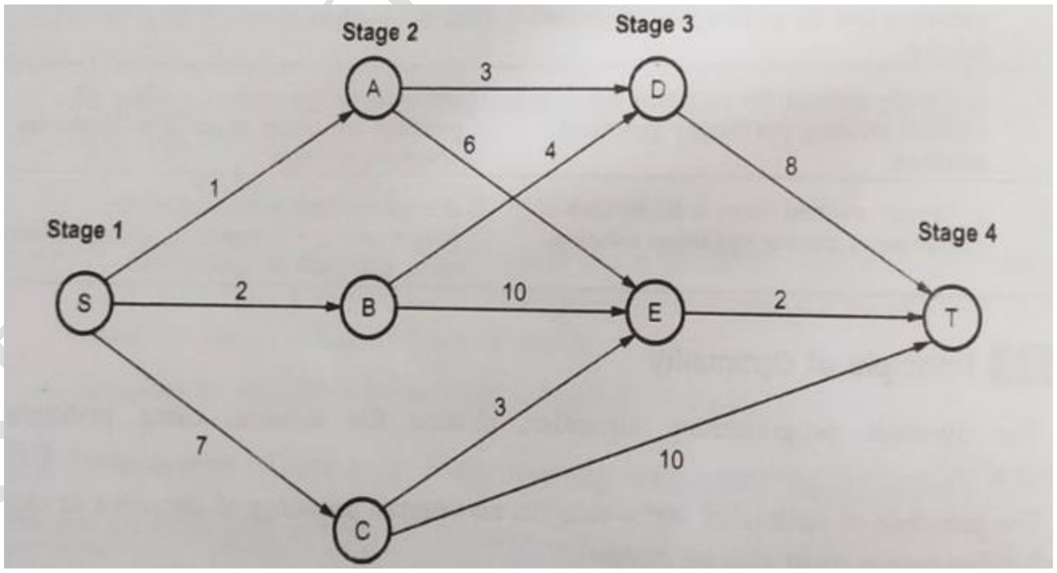


Fig: Multistage Graph

Consider the graph G as shown in the Fig

There is single vertex in stage 1, then 3 vertices in stage 2, then 2 vertices in stage and only one vertex in stage 4 (this is a target stage).

Backward approach d (S, T) = min {1+ d (A, T), 2 + d (B, T), 7 + d (C, T)} …. (1)

We will now compute d (A, T), d (B, T) and d (C, T).

d (A, 1) = min {3 + d (D, T), 6 + d (E, 1)} …… (2)

d (B, T) = min {4 + d (D, T), 10 + d (E, T)} …… (3)

d (C, T) = min {3 + d (E, T), d (C, T)} ………. (4)

Now let us compute d(D, T) and d(E, T).

d (D, T) = 8

d (E, T) = 2 **backward vertex = E**

Let us put these values in equations (2), (3) and (4).

d (A, T) = min {3 + 8, 6 + 2}

d (A, T) = 8 **A - E -T**

d (B, T) = min {4 + 8, 10 + 2}

d (B, T) = 12 **A - D -T**

d (C, T) = min {3 + 2, 10}

d (C, T) = 5 **C - E – T**

d (S, T) = min {1 + d (A, T), 2 + d (B, T), 7 + d (C, T)}

= min {1 +8,2 + 12,7+5}

= min {9, 14, 12}

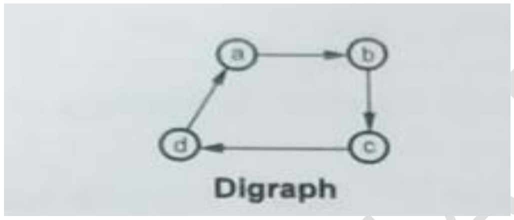
d (S, T) = 9 **S-A-E-T**

Chapter-II

4.3. Transitive Closure: Warshall's Algorithm.

1. Digraph: The graph in which all the edges are directed then it is called **digraph or directed graph.**

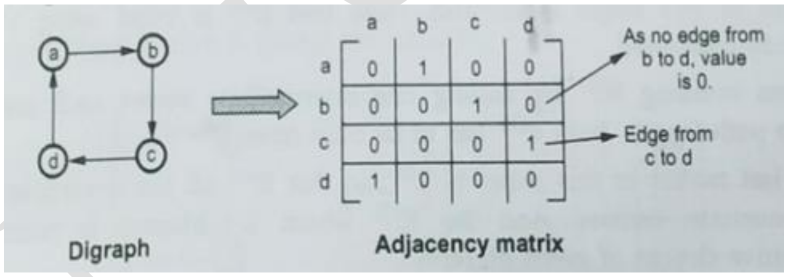
Example:



Digraph

2. Adjacency matrix: It is a representation of a graph by using matrix. If there exists an edge between the vertices V, and V, directing from V_i, to V_j, then entry in adjacency matrix in i^{th} row and j^{th} column is 1.

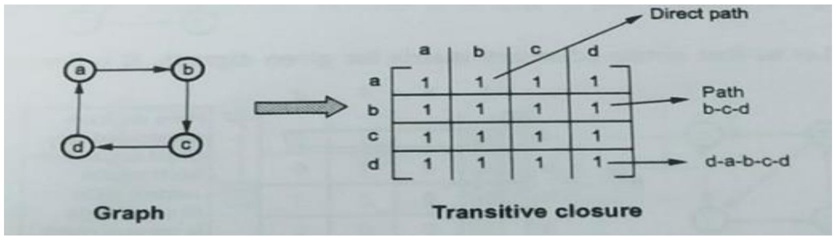
Example:



$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 \\ c & 0 & 0 & 0 & 1 \\ d & 1 & 0 & 0 & 0 \end{array}$$

3. Transitive closure: Transitive closure is basically a Boolean matrix (matrix with 0 and 1 values) in which the existence of directed paths of arbitrary lengths between vertices is mentioned.

Example:



- Transitive closure the transitive closure can be generated with Depth First Search (DFS) or with Breadth First Search (BFS).
- This traversing can be done any vertex. While computing transitive closure we have to start with some vertex and have to find all the edges which are reachable to every other vertex.
- The reachable edges for all the vertices has to be obtained.

Basic Concept

- While computing the transitive closure, we have to traverse the graph several times.
- To avoid this repeated traversing through graph a new algorithm is discovered by S. Warshall which is called Warshall's algorithm.
- Warshall's algorithm constructs the transitive closure of given digraph with n vertices through a series of n-by-n Boolean matrices.

The computations in Warshall's algorithm are given by following sequence,

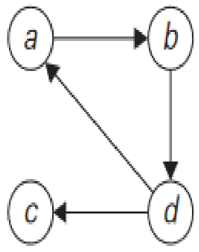
$$\boxed{R^{(0)}, \ldots, R^{(k-1)}, \ldots, R^{(k)}, \ldots R^{(n)}}$$

Thus, the key idea in Warshall's algorithm is building of Boolean matrices.

Procedure to be followed:

- Start with computation of R $^{(0).}$
- Construct R $^{(1)}$ in which first vertex is used as intermediate vertex and a path length of two edges is allowed. Note that R $^{(1)}$ is build using R $^{(1)}$ which is already computed.
- Go on building R$^{(k)}$ by adding one intermediate vertex each time and with more path length. Each R$^{(k)}$ has to be built from R$^{(k-1)}$.
- The last matrix in this series is R$^{(n)}$, in this R$^{(n)}$ all the n vertices are used a intermediate vertices. And the R$^{(n)}$ which is obtained is nothing but the transitive closure of given digraph.

```
ALGORITHM Warshall (A [1..n, 1..n])
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
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)
```



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

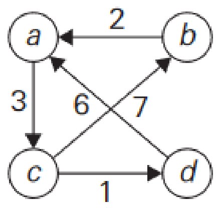
1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

Fig: Application of Warshall's algorithm to the digraph shown.

New 1's are in bold.



(a)

$$W = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

(b)

$$D = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

(c)

Fig: (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

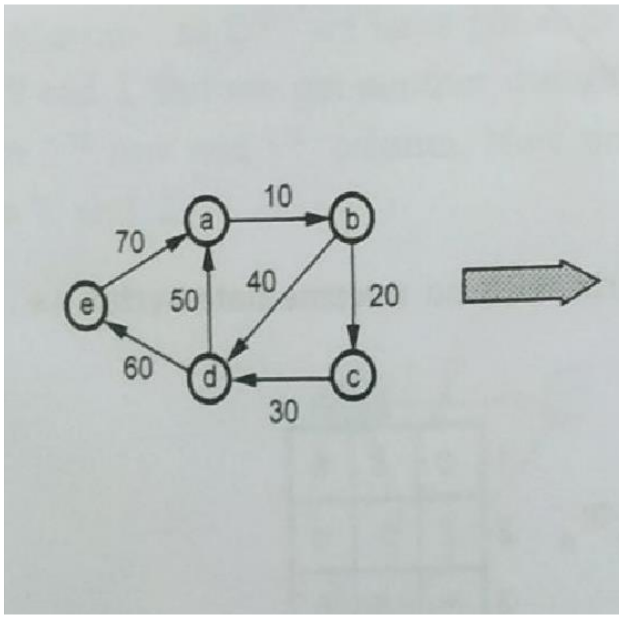
Chapter-III

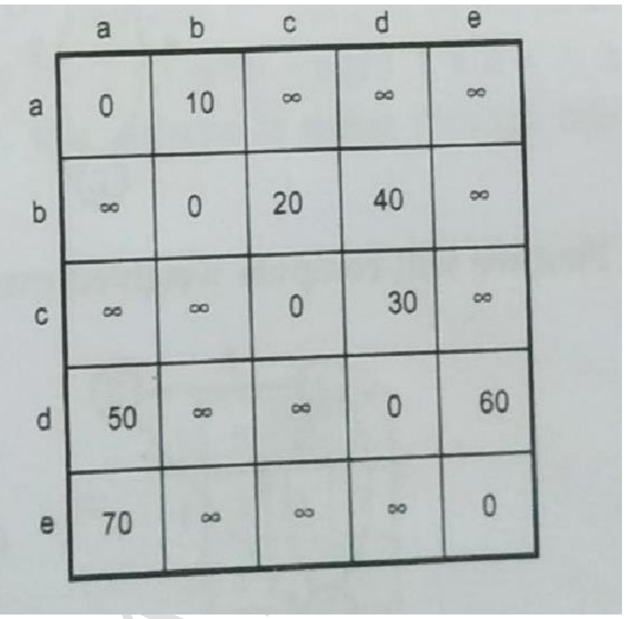
4.4. All Pairs Shortest Paths: Floyd's Algorithm

- Floyd's algorithm is for finding the shortest path between every pair of vertices of a graph.
- This algorithm is invented by R. Floyd and hence is the name.
- Before getting introduced with this algorithm, let us revise few concepts related with graph.

Weighted graph: The weighted graph is a graph in which weights or distances are given along the edges.

The weighted graph can be represented by weighted matrix as follows,





	a	b	c	d	e
a	0	10	∞	∞	∞
b	∞	0	20	40	∞
c	∞	∞	0	30	∞
d	50	∞	∞	0	60
e	70	∞	∞	∞	0

Here w[i][j] = 0 if i=j

w[i][j] = ∞ if there is no edge (direct edge) between i and j.

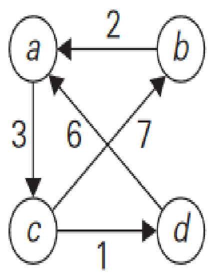
W[i][j] = Weight of edge.

Basic concept of Floyd's Algorithm

- The Floyd's algorithm is for computing shortest path between every pair of vertices of graph.
- The graph may contain negative edges but it should not contain negative cycles.
- The Floyd's algorithm requires a weighted graph.
- Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n-by-n matrices:

$$\boxed{D^{(0)}, D^{(1)}, \ldots D^{(k-1)}, \ldots D^{(n)}}$$

```
ALGORITHM Floyd (W [1...n, 1...n])
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
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
```



$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \mathbf{5} & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \mathbf{9} & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

Fig: Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

4.5. Knapsack Problem

The knapsack problem can be defined as follows : If there are n items with the weights $w_1, w_2, \ldots w_n$ and values (profit associated with each item) $v_1, v_2, \ldots v_n$ and capacity of knapsack to be W, then find the most valuable subset of the items that fit into the knapsack.

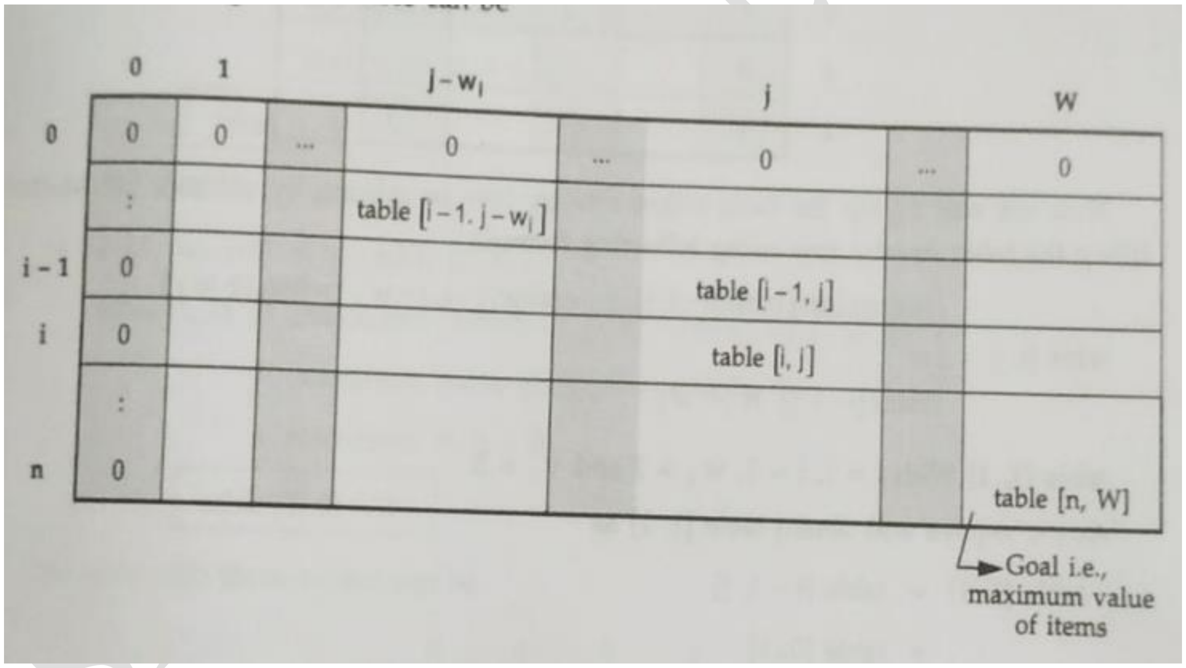
To solve this problem using dynamic programming we will write the recurrence relation as :

table [1, j] = maximum {table [i j], v_i +table [i -1, j - w_i]} if j > w_i

or

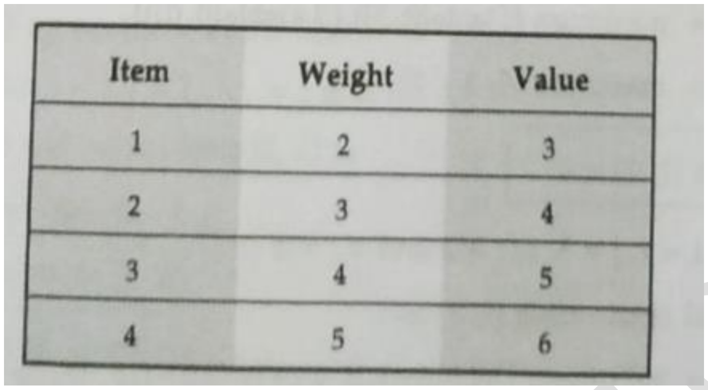
table [i- 1, j] if j < w_i

The initial stage of the table can be



The table [n, W] is a goal i.e., it's gives the total items sum of all the selected items for the knapsack.

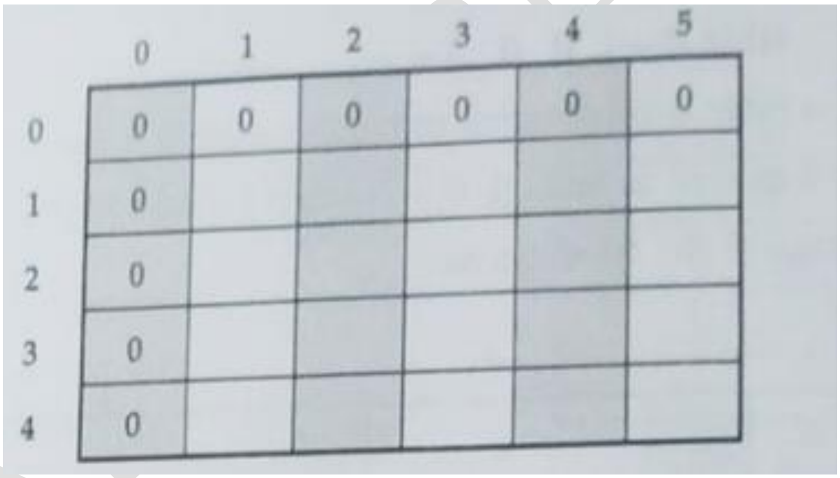
Example: For the given instance of problem obtain the optimal solution for the knapsack problem.



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

The capacity of knapsack is W = 5

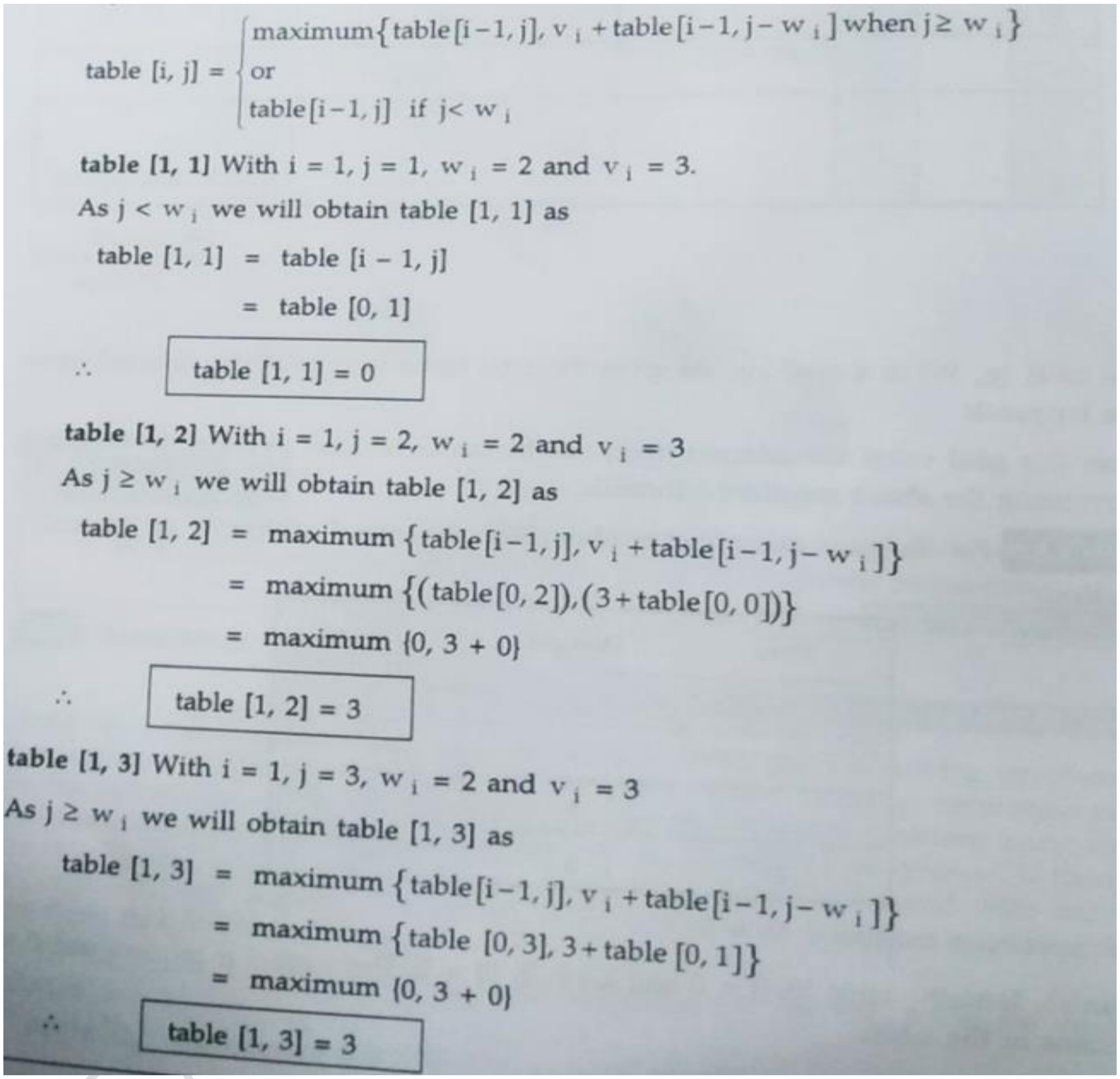
Solution: Initially, table [0, j] = 0 and table [i,0] = 0. There are 0 to n rows and 0 to W columns in the table.



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

Now we will fill up the table either row by row or column by column.

Let us start filling the table row by row using following formula:



$$\text{table}[i, j] = \begin{cases} \text{maximum}\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i] \text{ when } j \geq w_i\} \\ \text{or} \\ \text{table}[i-1, j] \text{ if } j < w_i \end{cases}$$

table [1, 1] With $i = 1, j = 1, w_i = 2$ and $v_i = 3$.

As $j < w_i$ we will obtain table [1, 1] as

$$\begin{aligned} \text{table}[1, 1] &= \text{table}[i-1, j] \\ &= \text{table}[0, 1] \end{aligned}$$

$\therefore$ $\boxed{\text{table}[1, 1] = 0}$

table [1, 2] With $i = 1, j = 2, w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 2] as

$$\begin{aligned} \text{table}[1, 2] &= \text{maximum}\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum}\{(\text{table}[0, 2]), (3 + \text{table}[0, 0])\} \\ &= \text{maximum}\{0, 3 + 0\} \end{aligned}$$

$\therefore$ $\boxed{\text{table}[1, 2] = 3}$

table [1, 3] With $i = 1, j = 3, w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 3] as

$$\begin{aligned} \text{table}[1, 3] &= \text{maximum}\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum}\{\text{table}[0, 3], 3 + \text{table}[0, 1]\} \\ &= \text{maximum}\{0, 3 + 0\} \end{aligned}$$

$\therefore$ $\boxed{\text{table}[1, 3] = 3}$

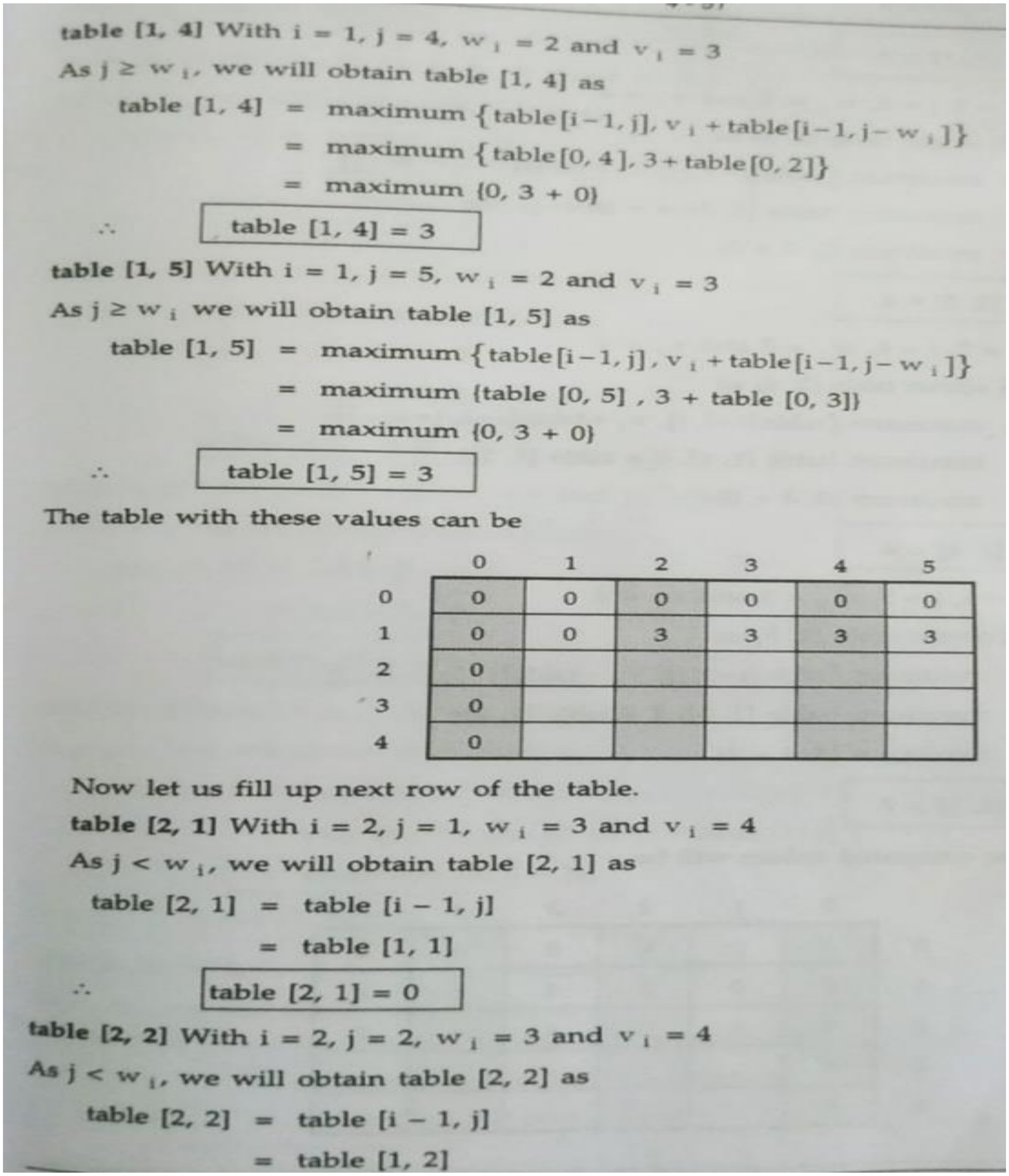


table [1, 4] With $i = 1, j = 4, w_i = 2$ and $v_i = 3$

As $j \geq w_i$, we will obtain table [1, 4] as

$$\begin{aligned} \text{table}[1, 4] &= \text{maximum}\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum}\{\text{table}[0, 4], 3 + \text{table}[0, 2]\} \\ &= \text{maximum}\{0, 3 + 0\} \end{aligned}$$

$\therefore$ $\boxed{\text{table}[1, 4] = 3}$

table [1, 5] With $i = 1, j = 5, w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 5] as

$$\begin{aligned} \text{table}[1, 5] &= \text{maximum}\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum}\{\text{table}[0, 5], 3 + \text{table}[0, 3]\} \\ &= \text{maximum}\{0, 3 + 0\} \end{aligned}$$

$\therefore$ $\boxed{\text{table}[1, 5] = 3}$

The table with these values can be

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

Now let us fill up next row of the table.

table [2, 1] With $i = 2, j = 1, w_i = 3$ and $v_i = 4$

As $j < w_i$, we will obtain table [2, 1] as

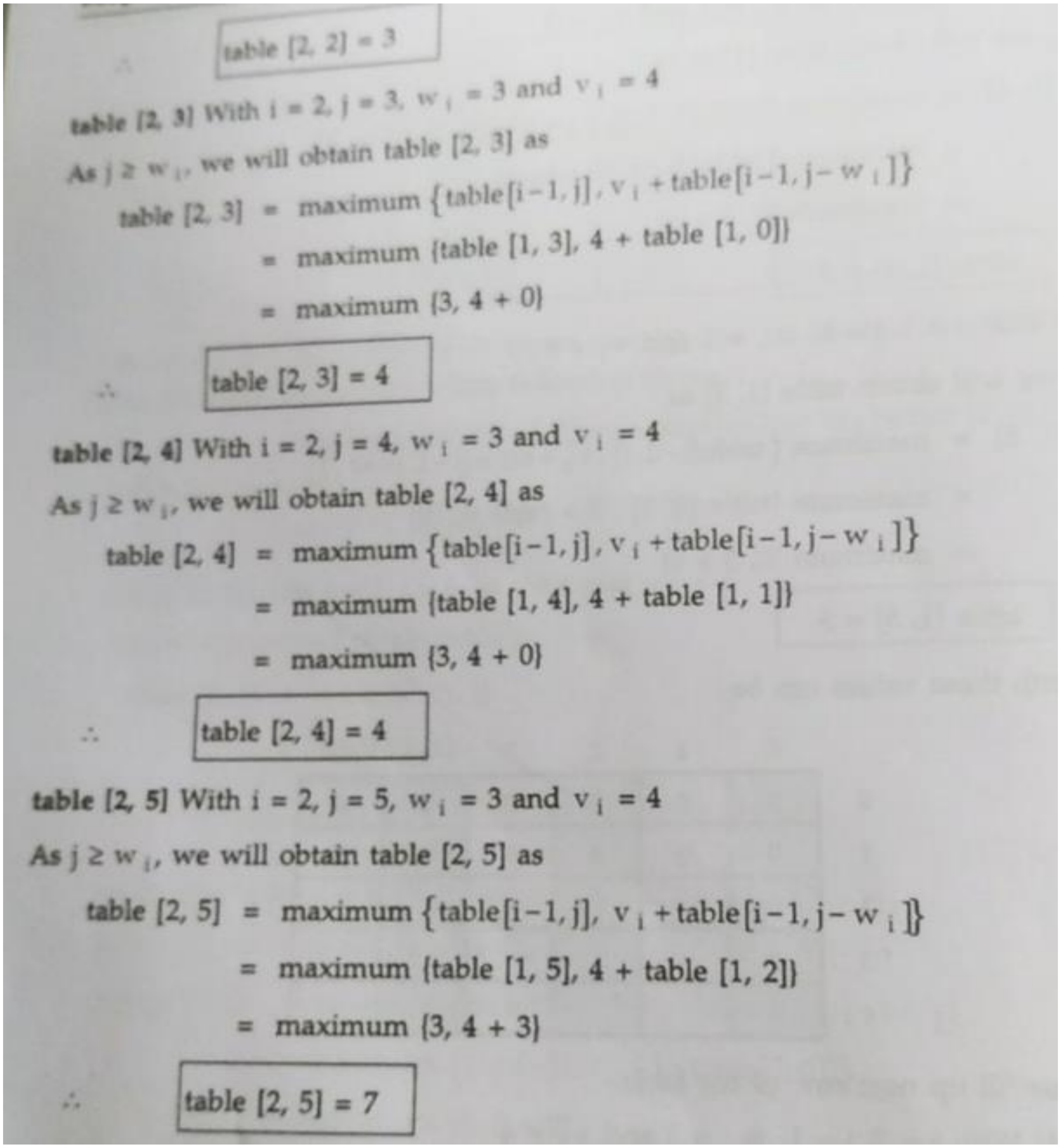
$$\begin{aligned} \text{table}[2, 1] &= \text{table}[i - 1, j] \\ &= \text{table}[1, 1] \end{aligned}$$

$\therefore$ $\boxed{\text{table}[2, 1] = 0}$

table [2, 2] With $i = 2, j = 2, w_i = 3$ and $v_i = 4$

As $j < w_i$, we will obtain table [2, 2] as

$$\begin{aligned} \text{table}[2, 2] &= \text{table}[i - 1, j] \\ &= \text{table}[1, 2] \end{aligned}$$



$\therefore$ $\boxed{\text{table } [2, 2] = 3}$

table [2, 3] With $i = 2$, $j = 3$, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 3] as

$$\begin{aligned} \text{table } [2, 3] &= \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum } \{\text{table } [1, 3], 4 + \text{table } [1, 0]\} \\ &= \text{maximum } \{3, 4 + 0\} \end{aligned}$$

$\therefore$ $\boxed{\text{table } [2, 3] = 4}$

table [2, 4] With $i = 2$, $j = 4$, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 4] as

$$\begin{aligned} \text{table } [2, 4] &= \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum } \{\text{table } [1, 4], 4 + \text{table } [1, 1]\} \\ &= \text{maximum } \{3, 4 + 0\} \end{aligned}$$

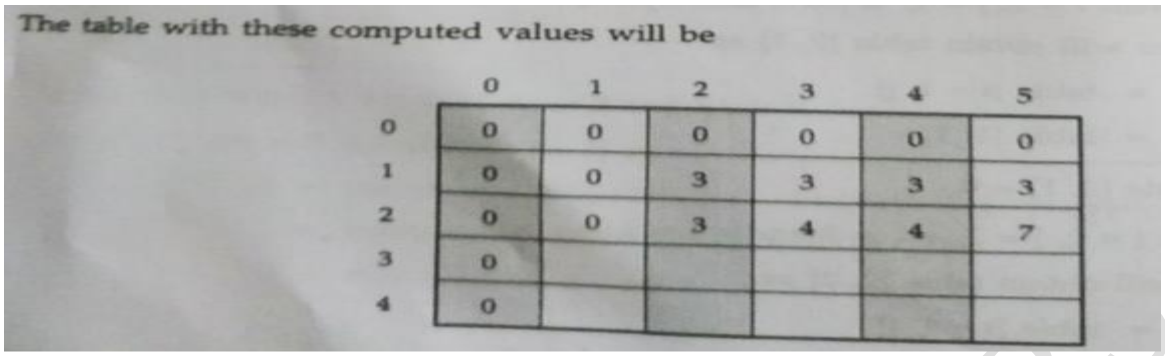
$\therefore$ $\boxed{\text{table } [2, 4] = 4}$

table [2, 5] With $i = 2$, $j = 5$, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 5] as

$$\begin{aligned} \text{table } [2, 5] &= \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum } \{\text{table } [1, 5], 4 + \text{table } [1, 2]\} \\ &= \text{maximum } \{3, 4 + 3\} \end{aligned}$$

$\therefore$ $\boxed{\text{table } [2, 5] = 7}$



The table with these computed values will be

	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					
4	0					

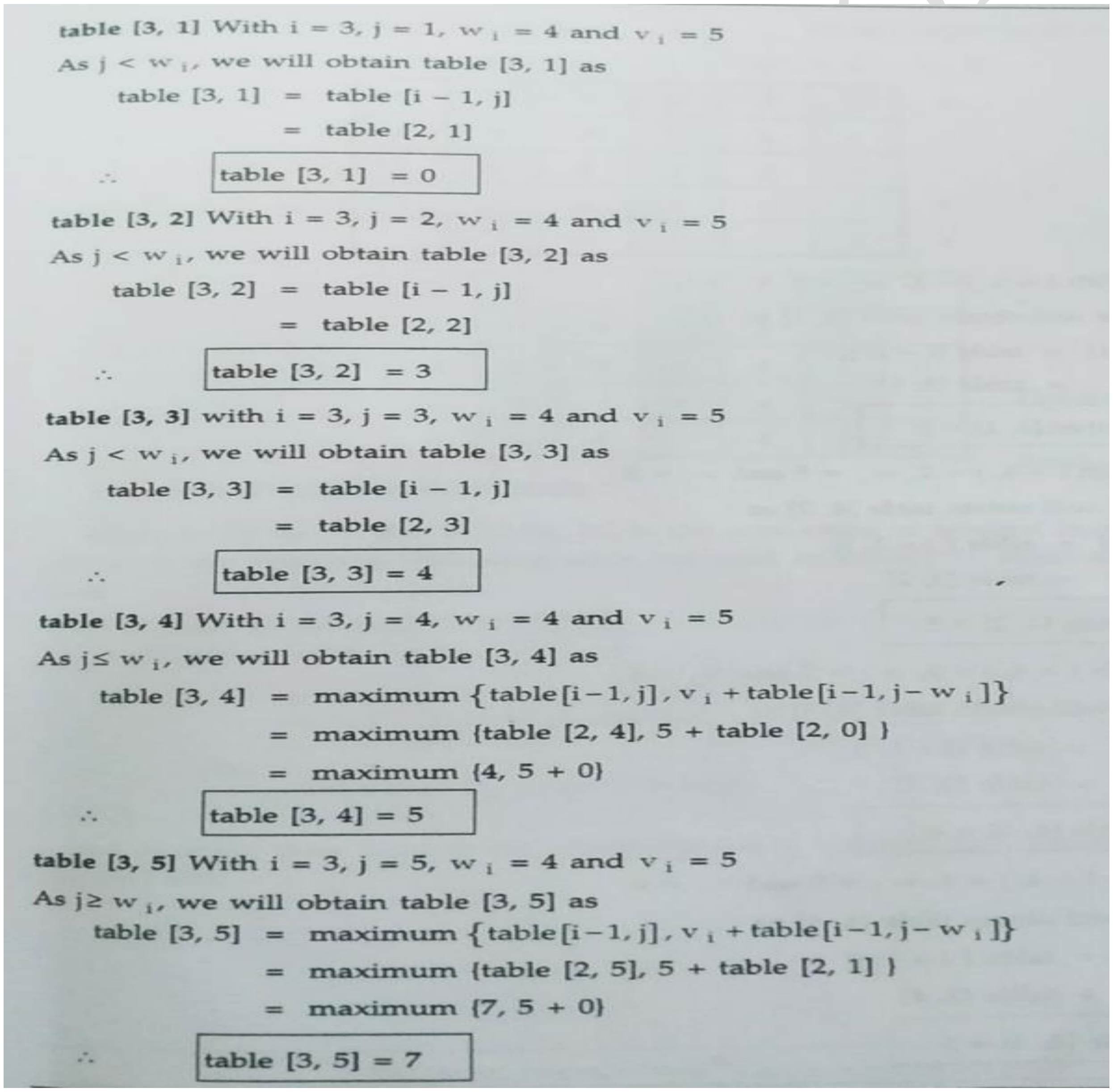


table [3, 1] With $i = 3, j = 1, w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 1] as

$$\begin{aligned} \text{table }[3, 1] &= \text{table }[i-1, j] \\ &= \text{table }[2, 1] \end{aligned}$$

$\therefore$ $\boxed{\text{table }[3, 1] = 0}$

table [3, 2] With $i = 3, j = 2, w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 2] as

$$\begin{aligned} \text{table }[3, 2] &= \text{table }[i-1, j] \\ &= \text{table }[2, 2] \end{aligned}$$

$\therefore$ $\boxed{\text{table }[3, 2] = 3}$

table [3, 3] with $i = 3, j = 3, w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 3] as

$$\begin{aligned} \text{table }[3, 3] &= \text{table }[i-1, j] \\ &= \text{table }[2, 3] \end{aligned}$$

$\therefore$ $\boxed{\text{table }[3, 3] = 4}$

table [3, 4] With $i = 3, j = 4, w_i = 4$ and $v_i = 5$

As $j \le w_i$, we will obtain table [3, 4] as

$$\begin{aligned} \text{table }[3, 4] &= \text{maximum }\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum }\{\text{table }[2, 4], 5 + \text{table }[2, 0]\} \\ &= \text{maximum }\{4, 5 + 0\} \end{aligned}$$

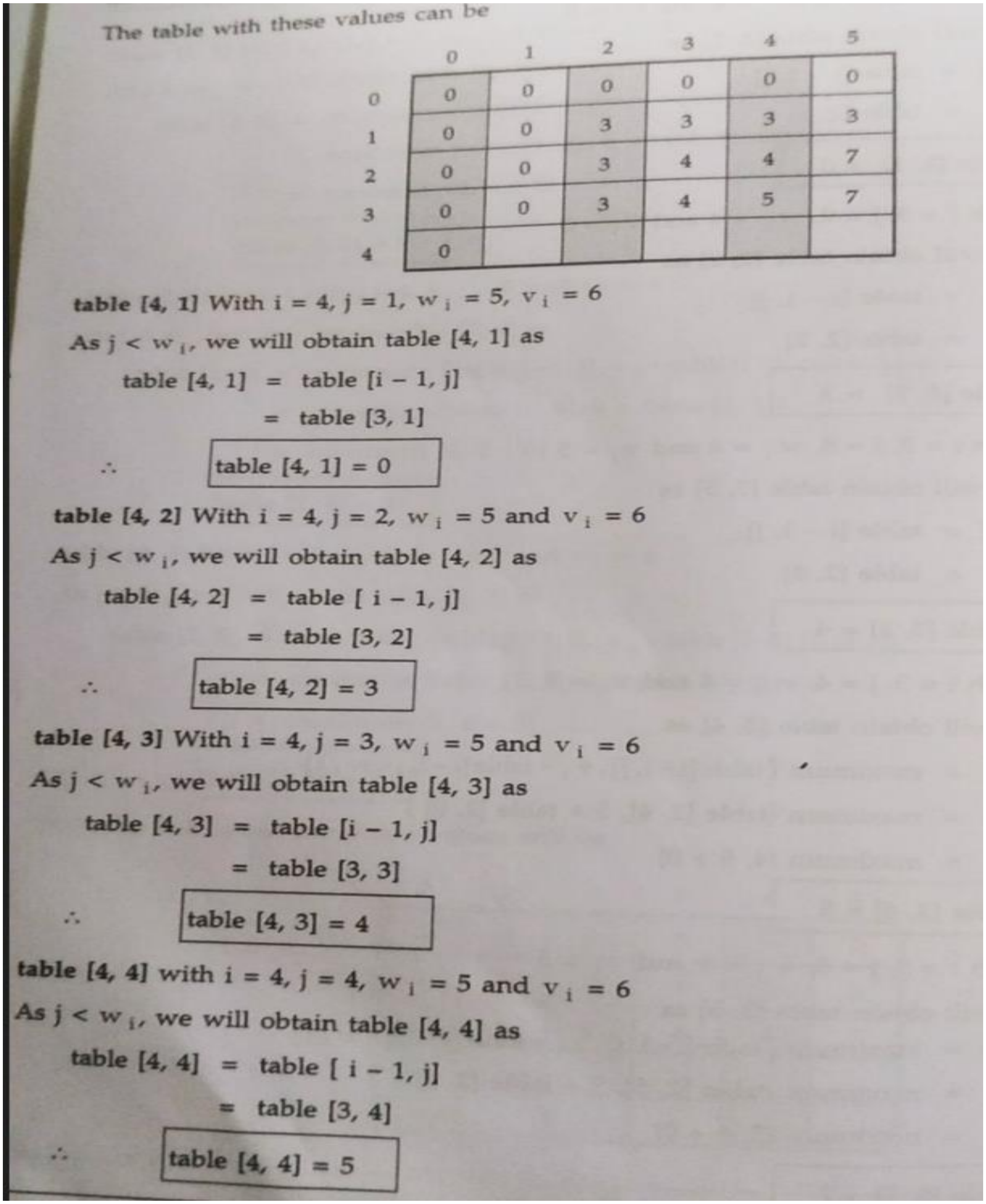
$\therefore$ $\boxed{\text{table }[3, 4] = 5}$

table [3, 5] With $i = 3, j = 5, w_i = 4$ and $v_i = 5$

As $j \ge w_i$, we will obtain table [3, 5] as

$$\begin{aligned} \text{table }[3, 5] &= \text{maximum }\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum }\{\text{table }[2, 5], 5 + \text{table }[2, 1]\} \\ &= \text{maximum }\{7, 5 + 0\} \end{aligned}$$

$\therefore$ $\boxed{\text{table }[3, 5] = 7}$



The table with these values can be

	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					

table [4, 1] With $i = 4, j = 1, w_i = 5, v_i = 6$

As $j < w_i$, we will obtain table [4, 1] as

table [4, 1] = table [i − 1, j]

= table [3, 1]

∴ $\boxed{\text{table } [4, 1] = 0}$

table [4, 2] With $i = 4, j = 2, w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 2] as

table [4, 2] = table [i − 1, j]

= table [3, 2]

∴ $\boxed{\text{table } [4, 2] = 3}$

table [4, 3] With $i = 4, j = 3, w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 3] as

table [4, 3] = table [i − 1, j]

= table [3, 3]

∴ $\boxed{\text{table } [4, 3] = 4}$

table [4, 4] with $i = 4, j = 4, w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 4] as

table [4, 4] = table [i − 1, j]

= table [3, 4]

∴ $\boxed{\text{table } [4, 4] = 5}$

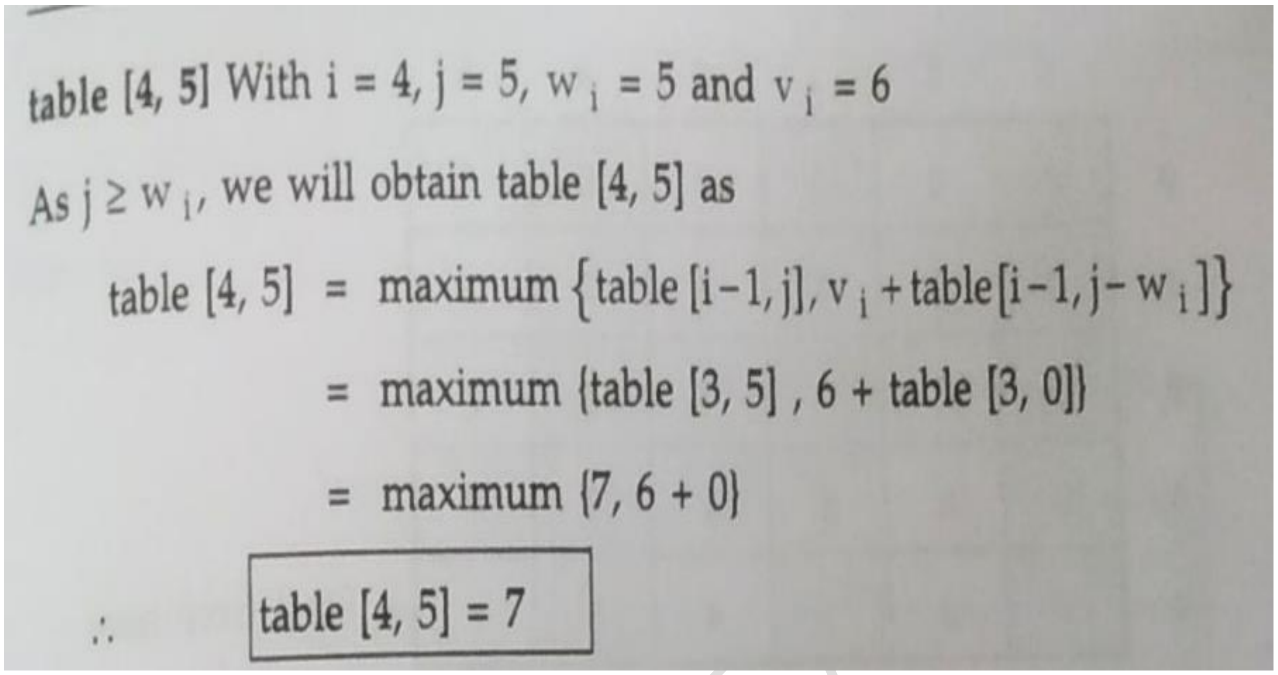
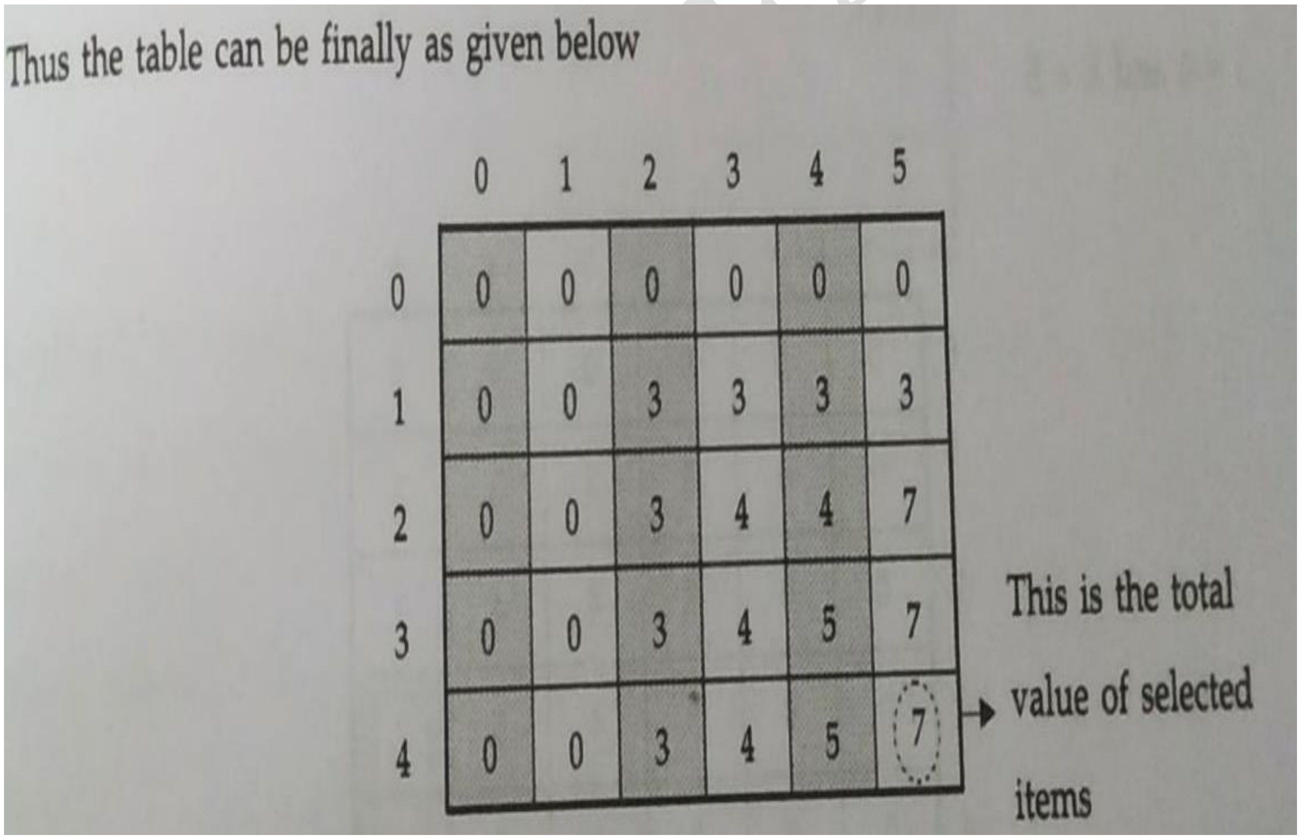


table [4, 5] With $i = 4, j = 5, w_i = 5$ and $v_i = 6$

As $j \geq w_i$, we will obtain table [4, 5] as

$$\begin{aligned} \text{table}[4, 5] &= \text{maximum}\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \\ &= \text{maximum}\{\text{table}[3, 5], 6 + \text{table}[3, 0]\} \\ &= \text{maximum}\{7, 6 + 0\} \end{aligned}$$

$\therefore$ $\boxed{\text{table}[4, 5] = 7}$



Thus the table can be finally as given below

	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

This is the total value of selected items

Algorithm

Algorithm Dynamic Knapsack(n,W,w[ ],v[ ])

//**Problem Description:** This algorithm is for obtaining knapsack solution using dynamic programming.

//**Input:** n is total number of items, W is the capacity of knapsack, w[ ] stores weights of each item and v[ ] stores the values of each item.

//**Output:** Returns the total value of selected items for the knapsack.

```
for(i←0 to n) do
{
    for(j←0 to W) do
    {
        table[i,0]=0
        table[0,j]=0
    }
}for(i←0 to n) do
{
    for(j←0 to W) do
    {
if  (j<w[i]) then
    table [i,j] ←table[i-1,j]
else if(j>=w[i]) then
    table[i,j] ←max(table [i-1,j],(v[i]+table[i-1,j-w[i]]))
}
}
return table[n,W].
```

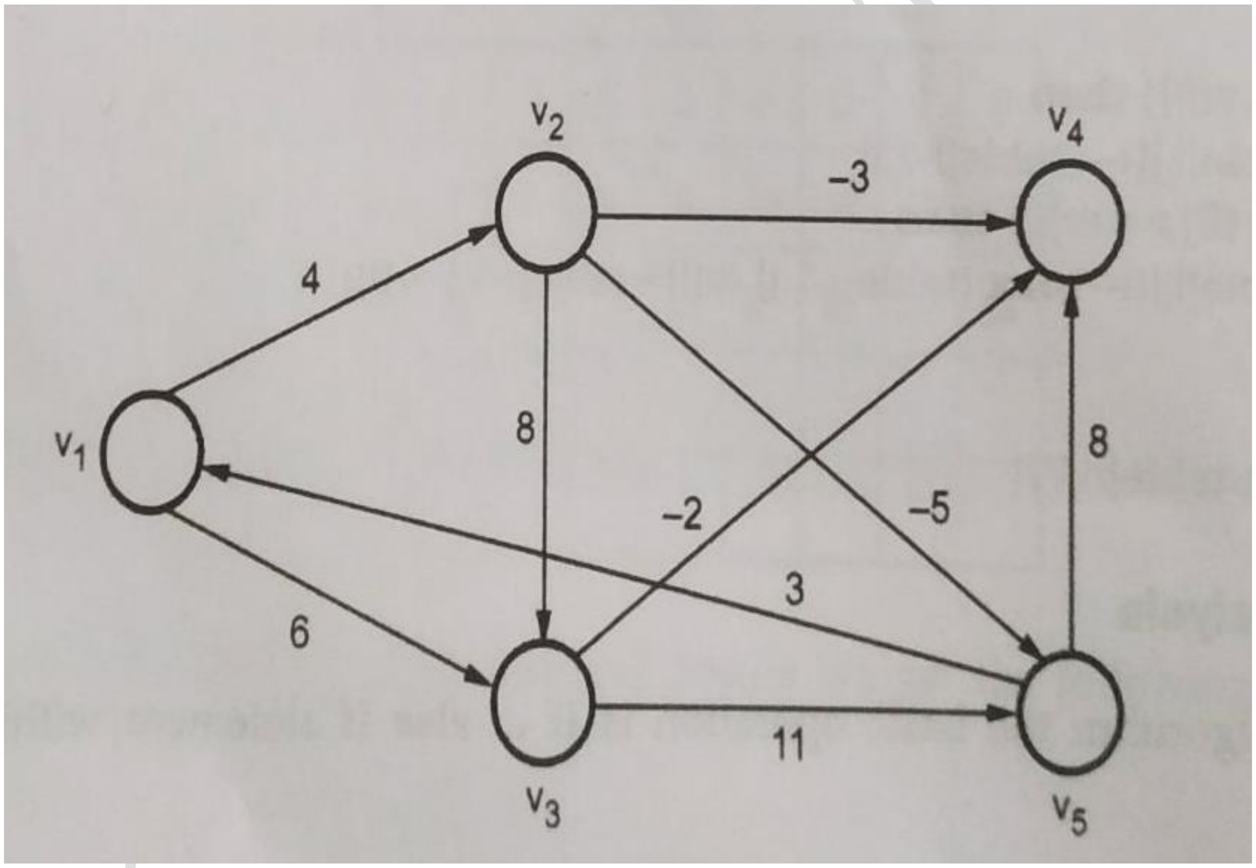
4.6. Bellman-Ford Algorithm

The Bellman-Ford algorithm works for finding single source shortest path. works for **negative edge** weights.

This is a single source shortest path algorithm.

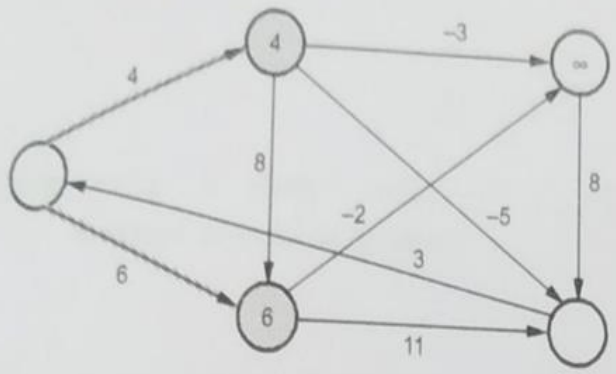
Let us understand this algorithm with the help of some example.

Example: Consider following for which the shortest path can be obtained from source vertex v_1.



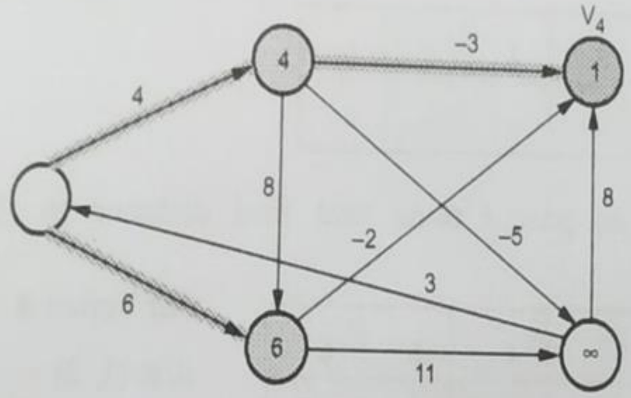
Solution :

Step 1 :



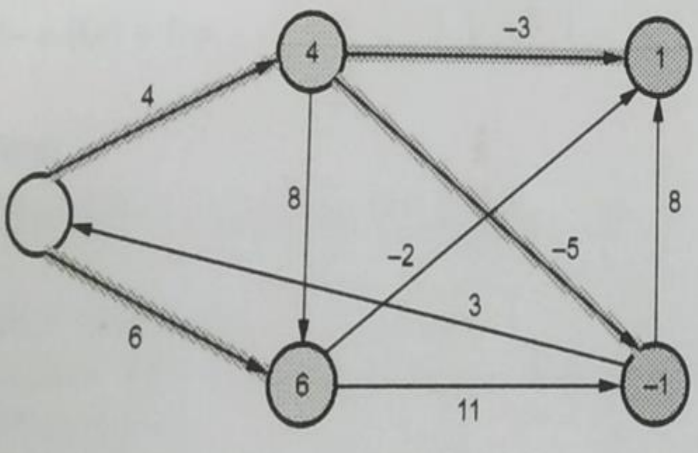
From vertex v_1 we find the distances to v_2 and v_3. The direct edges corresponds to the weights of destination vertices.

Step 2



Modify vertex v_4 by its minimum weight i.e. 1. The path is shown by shaded area.

Step 3 :



Process vertex v_5 by its minimum weight –1. The path is shown by shaded area.

Thus minimum distance of each vertex is obtained.

Algorithm

Algorithm Bellman Ford (vertices, edges, source)

```
{
//Problem Description : This algorithm finds the shortest path using Bellman Ford
method
for (each vertex v)
{
    if (v is source) then
     v distance k ←0
    else
    v. distance ←infinity
    v. prede←Null
    }
    for to toal vertices — 1)
    {
        for (each edge uv)
        {
        U ← uv. source
        V← uv. desination
        if (v. distance > u. distance + uv. weight ) then
        {
            v distance ←u. distance + uv. weight
            v. prede ← u
        }
    } for (each edge uv)
    {
    U← uv. source
    v ← uv. destination
    if (v. distance > u.distance + uv. weight) then
    {
        Write ("Graph has negative edges")
        return False
    }
    }
    } //end of for return True
    }// end of algorithm
```

4.7. Travelling Sales Person Problem

Problem Description

- Let G directed graph denoted by (V, E) where V denotes set of vertices 17: = denotes set of edges.
- The edges are given along with their cost C_{ij} .
- The cost C_{ij} >0 for all i and j. If there is no edge between i and j then $C_{ij} = \infty$.
- A tour for the graph should be such that all the vertices should be visited only and cost of the tour is sum of cost of edges on the tour.
- The traveling salesperson problem is to find the tour of minimum cost.

Dynamic programming is used to solve this **Problem**.

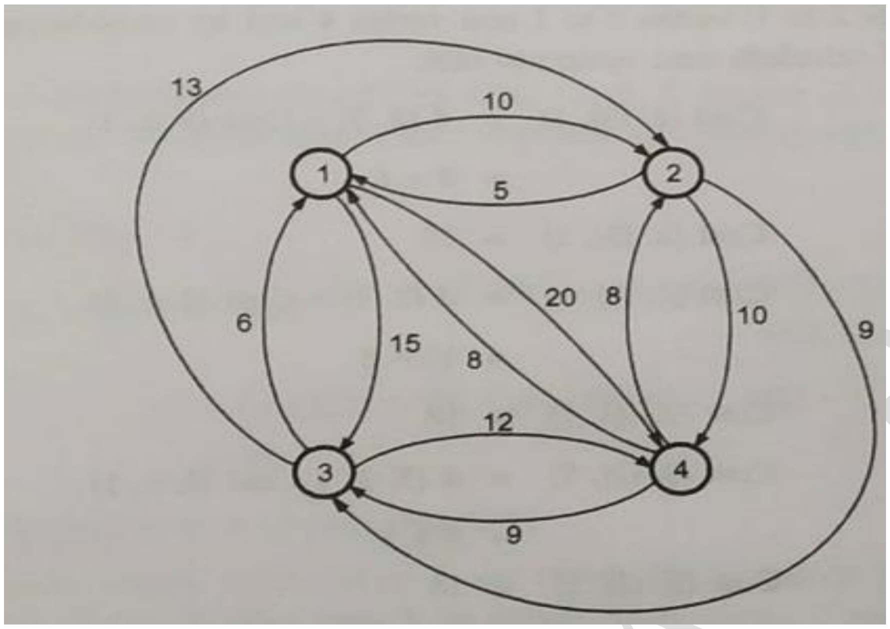
Step 1: Let the function C (1, V- {1}) is the total length of the tour terminating at 1 the objective of TSP problem is that the cost of this tour should be minimum.

Step 2: Let $V_1, V_2 \ldots V_n$, be the sequence of vertices followed in optimal tour. Then $(V_1, V_2 \ldots V_n)$ must be a shortest path from V_1 to V_n., which passes through each vertex exactly once.

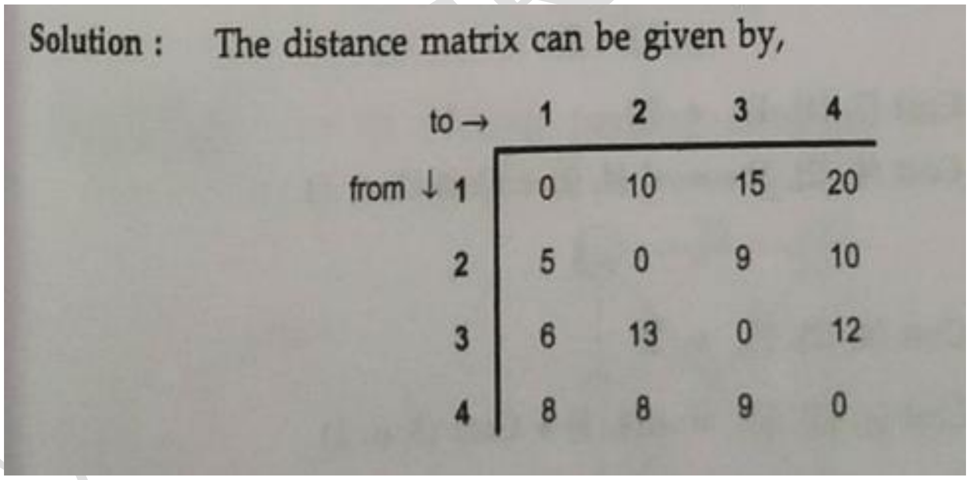
Step 3: Following formula can be used to obtain the optimum cost tour.

Cost (i, S) = min {d [i, j] + Cost (j, S—{j})} where j ∈ S and i ∈ S.

Consider one example to understand solving of TSP using dynamic programming approach



Fig



Solution : The distance matrix can be given by,

from ↓ \ to →	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

First, we will select any arbitrary vertex say select 1.

Now process for intermediate sets with increasing size.

Step 1: Let S = ∅ then,

Cost (2, ∅, 1) = d(2, 1) = 5

Cost (3, ∅, 1) = d(3, 1) = 6

Cost (4, ∅, 1) = d(4, 1) = 8

That means eve have obtained dist (2, 1), dist (3, 1) and dist (4, 1).

Step 2:

Candidate (S) = 1

Apply the formula,

Cost (i, S) = min {d[i, j]+ Cost (j, S-{j}}

Hence from vertex 2 to 1, vertex 3 to 1 and vertex 4 to 1 by considering intermediate path lengths we will calculate total optimum cost.

Cost (2, {3}, 1) = d (2, 3) + Cost (3, ∅, 1)

= 9 + 6

Cost (2, {3}, 1) = 15

Cost (2, {4}, 1) = d (2, 4) + Cost (4, ∅, 1)

= 10+8

Cost (2, {4}, 1) = 18

Cost (3, {2}, 1) = d (3, 2) + Cost (2, ∅, 1)

= 3 + 5

Cost (3, {2}, 1) = 8

Cost (3, {4}, 1) = d (3, 4) + Cost (4, ∅, 1)

= 12+8

Cost (3, {4}, 1) = 20

Cost (4, {2}, 1) = d (4, 2) + Cost (2, ∅,, 1) = 8 + 5

Cost (4, {2}, 1) = 13

Cost (4, {3}, 1) = d (4, 3) + Cost (3, ∅,, 1)

=9 + 6

Cost (4, {3}, 1) = 15

Step 3: Consider candidate (S) = 2

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]}

Cost (2, {3, 4}, 1) = 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]}

Cost (3, (2, 41, 1) = 25

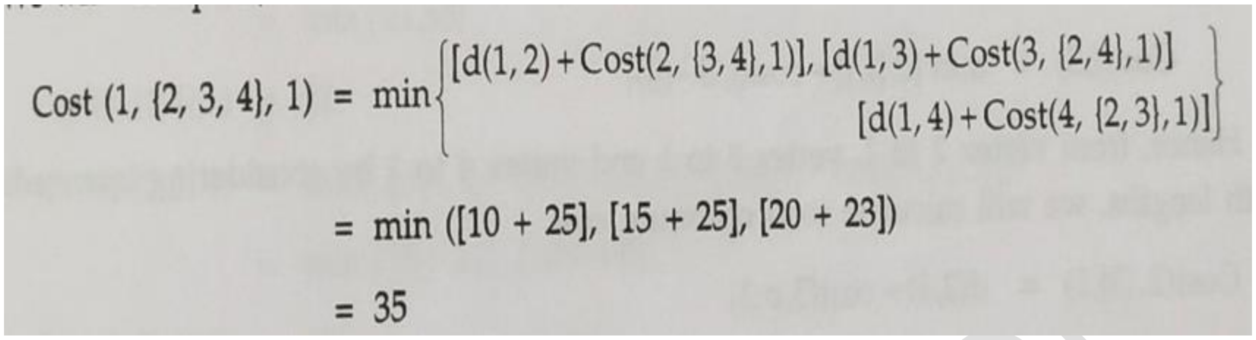
Cost (4, (2, 31, 1) = min {[d(4, 2) + Cost (2, {3}, 1)], [d(4, 3) + Cost (3, {2}, 1)]}

= min {[8 + 15], [9 + 18]}

Cost (4, {2, 3}, 1) = 23

Step 4: Consider candidate (S) = 3. i.e., Cost (1, {2, 3, 4}) but as we have chosen vertex 1 initially the cycle should be completed i.e., starting and ending vertex should be 1.

∴ We will compute,



$$\text{Cost}(1, \{2, 3, 4\}, 1) = \min\left\{\begin{array}{r}[d(1,2) + \text{Cost}(2, \{3,4\}, 1)], [d(1,3) + \text{Cost}(3, \{2,4\}, 1)] \\ [d(1,4) + \text{Cost}(4, \{2,3\}, 1)]\end{array}\right\}$$

$$= \min([10 + 25], [15 + 25], [20 + 23])$$

$$= 35$$

Thus, the optimal tour is of path length 35.

Chapter-IV

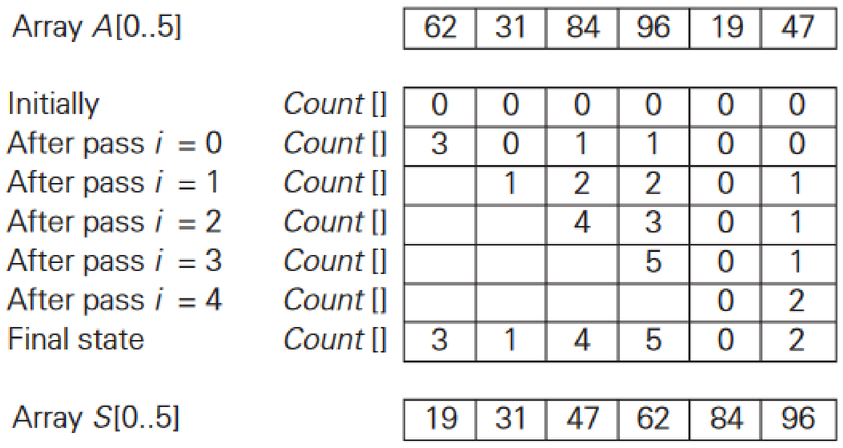
Space-Time Tradeoffs

4.8. Introduction

- Space and time trade-offs in algorithm design are a well-known issue for both theoreticians and practitioners of computing.
- Consider, as an example The problem of computing values of a function at many points in its domain. If it is time that is at a premium, we can precompute the function’s values and store them in a table.
- This is exactly what human computers had to do before the advent of electronic computers, in the process burdening libraries with thick volumes of mathematical tables.
- Though such tables have lost much of their appeal with the widespread use of electronic computers, the underlying idea has proven to be quite useful in the development of several important algorithms for other problems.

4.9. Sorting By Counting

- One rather obvious idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element and record the results in a table.
- These numbers will indicate the positions of the elements in the sorted list: e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array.
- Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list. This algorithm is called comparison counting sort



Array *A*[0..5]		62	31	84	96	19	47
Initially	*Count* []	0	0	0	0	0	0
After pass *i* = 0	*Count* []	3	0	1	1	0	0
After pass *i* = 1	*Count* []		1	2	2	0	1
After pass *i* = 2	*Count* []			4	3	0	1
After pass *i* = 3	*Count* []				5	0	1
After pass *i* = 4	*Count* []					0	2
Final state	*Count* []	3	1	4	5	0	2
Array *S*[0..5]		19	31	47	62	84	96

Fig: Example of sorting by comparison counting

```
ALGORITHM ComparisonCountingSort (A [0..n − 1])
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A’s elements sorted in nondecreasing order
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
                if A[i] < A[j ]
                        Count[j ] ← Count[j ] + 1
                else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```

EXAMPLE Consider sorting the array

13	11	12	13	12	12

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

	D[0..2]			*S*[0..5]					
A [5] = 12	1	**4**	6				12		
A [4] = 12	1	**3**	6			12			
A [3] = 13	1	2	**6**						13
A [2] = 12	1	**2**	5		12				
A [1] = 11	**1**	1	5	11					
A [0] = 13	0	1	**5**					13	

Example of sorting by distribution counting. The distribution values being decremented are shown in bold. ■

```
ALGORITHM DistributionCountingSort(A[0..n − 1], l, u)
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j ] ← 0 //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1 //compute frequencies
for j ← 1 to u − l do D[j ] ← D[j − 1] + D[j ] //reuse for distribution
for i ← n − 1 downto 0 do
j ← A[i] − l
S[D[j ] − 1] ← A[i]
D[j ] ← D[j ] − 1
return S
```

4.10. Input Enhancement in String Matching

- the problem of string matching requires finding an occurrence of a given string of m characters called the **pattern** in a longer string of n characters called the **text.**
- it simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial.
- Since the maximum number of such trials is n − m + 1 and, in the worst case, m comparisons need to be made on each of them, the worst-case efficiency of the brute-force algorithm is in the O(nm) class.
- Several faster algorithms have been discovered.
- Most of them exploit the input-enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text.

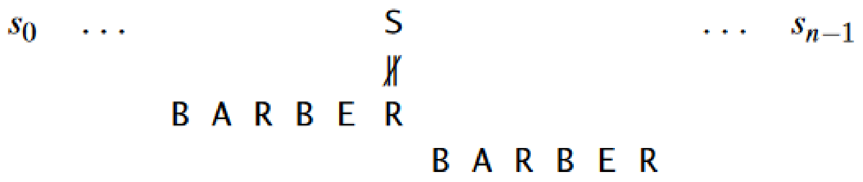
Horspool's Algorithm

Consider, as an example, searching for the pattern BARBER in some text:

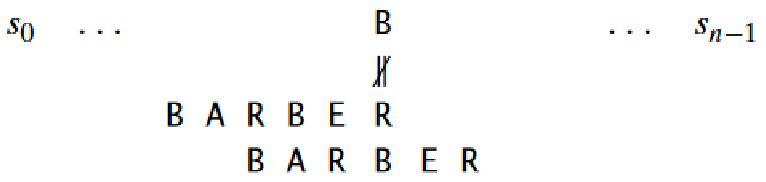
$$\begin{array}{cccccccccc} s_0 & \dots & & & & & & c & \dots & s_{n-1} \\ & & & B & A & R & B & E & R & \end{array}$$

In general, the following four possibilities can occur.

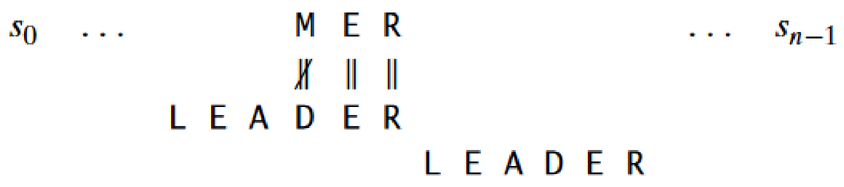
Case 1: If there are no c’s in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length



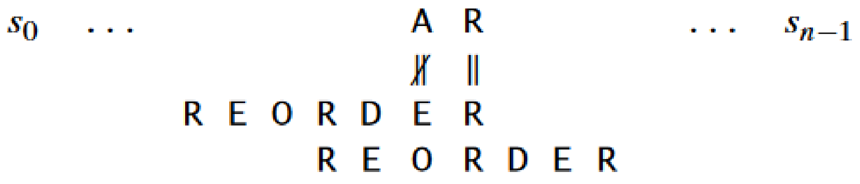
Case 2: If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:



Case 3: If c happens to be the last character in the pattern but there are no c’s among its other m − 1 characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern’s length m:


s_0 . . . M E R . . . s_{n-1}
$\not\Vert$ $\Vert$ $\Vert$
L E A D E R
L E A D E R

Case 4: Finally, if c happens to be the last character in the pattern and there are other c's among its first m − 1 characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first m − 1 characters in the pattern should be aligned with the text's c:



Horspool's algorithm

Step 1: For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2: Align the pattern against the beginning of the text.

Step 3: Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text.

Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched

```
ALGORITHM HorspoolMatching(P [0..m − 1], T [0..n − 1])
//Implements Horspool’s algorithm for string matching
//Input: Pattern P [0..m − 1] and text T [0..n − 1]
//Output: The index of the left end of the █rst matching substring
// or −1 if there are no matches
ShiftTable(P [0..m − 1]) //generate Table of shifts
i ← m − 1 //position of the pattern’s right end
while i ≤ n − 1 do
    k ← 0 //number of matched characters
    while k ≤ m − 1 and P [m − 1 − k] = T [i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T [i]]
return −1
```

Thanking You

Visit Our Official Website

http://searchcreators.org/