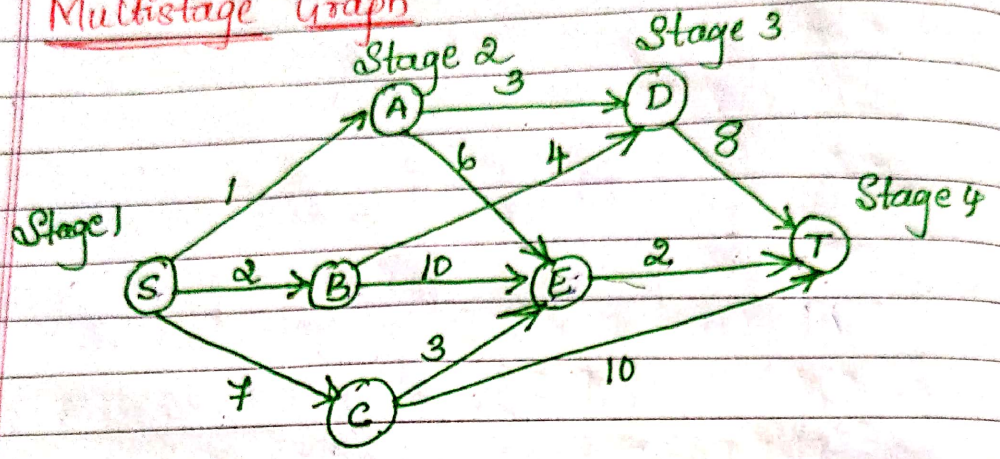
Module - 4

Dynamic Programming

General Method :-

Multistage Graph



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

we will compute $d(A,T)$, $d(B,T)$, $d(C,T)$

$d(A,T) = \min\{3+d(D,T), 6+d(E,T)\}$ → (2)

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

$d(C,T) = \min\{3+d(E,T), d(C,T)\}$ → (4)

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

$d(D,T) = 8$

$d(E,T) = 2$

Let us put these values in eqn (2) (3) and (4)

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

$d(A,T) = \min\{11, 8\}$

$\boxed{d(A,T) = 8}$ A - E - T

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

$d(B,T) = \min\{12, 12\}$

$\boxed{d(B,T) = 12}$ B - D - T

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

$d(C,T) = \min\{5, 10\}$

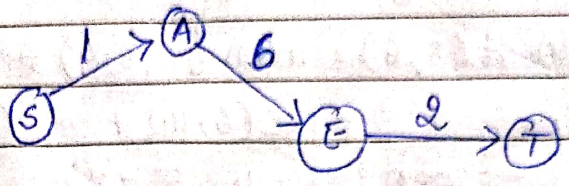
$\boxed{d(C,T) = 5}$ C - E - T

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

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

$= \min\{9, 14, 12\}$

$\boxed{d(S,T) = 9}$ S–A–E–T



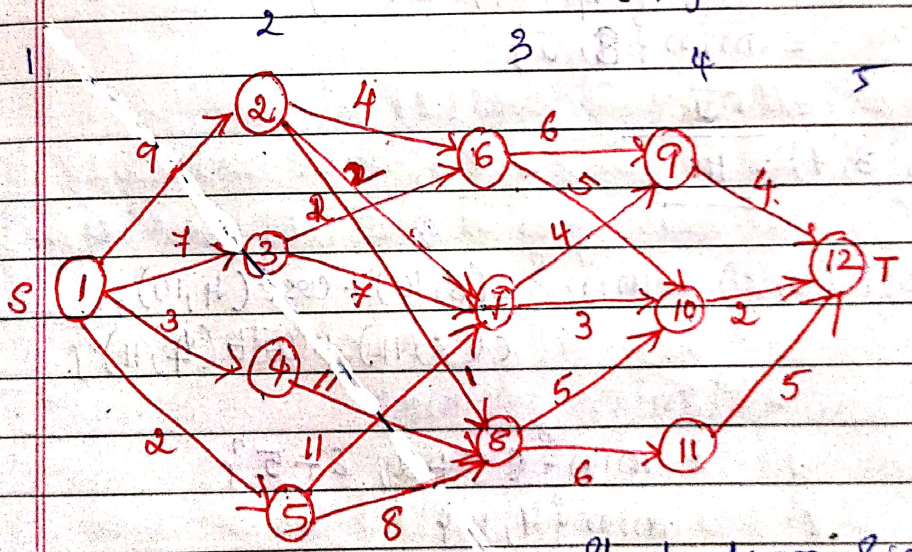
{Backward Approach}

Final Formula

Forward Approach:

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

$D(i,j) = l$



Start from first vertex

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

stage

$Cost(5,12) = 0$

stage

Step 1: $Cost(4,9) = \min\{c(9,12) + cost(5,12)\}$

$= \min\{4+0\}$

$= 4$

$Cost(4,10) = \min\{C(10,12) + cost(5,12)\}$
$= \min\{2+0\} = 2$ $D(4,10) = 12$

$Cost(4,11) = \min\{C(11,12) + cost(5,12)\}$
$= \min\{5+0\} = 5$ $D(4,11) = 12$

Step 2:- $Cost(3,6) = \min\{C(6,9) + cost(4,9)$
$C(6,10) + cost(4,10)\}$
$= \min\{6+4, 5+2\}$
$= \min\{10, 7\}$
$= 7$

$D(3,6) = 10$

$Cost(3,7) = \min\{C(7,9) + cost(4,9), C(7,10)$
$+ cos(4,10)\}$
$= \min\{4+4, 3+2\}$
$= \min\{8, 5\}$
$= 5$

$D(3,7) = 10$

$Cost(3,8) = \min\{C(8,10) + cost(4,10),$
$C(8,11) + cost(4,11)\}$

$= \min\{5+2, 6+5\}$
$= \min\{7, 11\}$
$= 7$

$D(3,8) = 10$

Step 3:-
$Cost(2,2) = \min\{C(2,6) + cost(3,6),$
$C(2,7) + cost(3,7), C(2,8) + cost($
$= \min\{4+7, 2+5, 1+7\}$
$= \min\{11, 7, 8\}$
$= 7$

$D(2,2) = 7$

PAGE NO.
DATE / /

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

$D(2,3) = 6$

$Cost(2,4) = \min\{ c(4,8) + cost(3,8) \}$
$= \min\{ 11+7 \}$
$= 18$

$D(2,4) = 8$

$Cost(2,5) = \min\{ c(5,7) + cost(3,7),$
$c(5,8) + cost(3,8) \}$
$= \min\{ 11+5, 8+7 \}$
$= \min\{ 16, 15 \}$
$= 15$

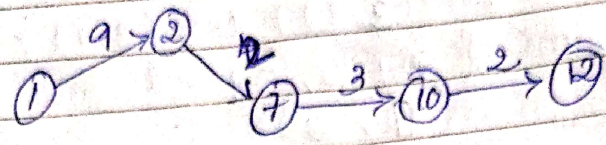
$D(2,5) = 8$

$Cost(1,1) = \min\{ c(1,2) + cost(2,2), c(1,3) +$
$cos(2,3), c(1,4) + cos(2,4),$
$c(1,5) + cost(2,5) \}$
$= \min\{ (9+7, 7+9, 3+8, 2+15) \}$
$= 16$

$D(1,1) = 2$ or 3

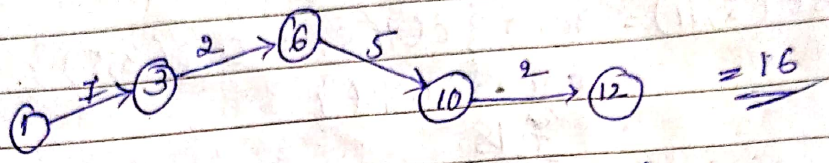
$D(1,1) = 2$	$D(1,1) = 3$
$D(2,2) = 7$	$D(2,3) = 6$
$D(3,7) = 10$	$D(3,6) = 10$
$D(4,10) = 12$	$D(4,10) = 12$
$1-2-7-10-12 = 16$	$1-3-6-10-12 = 16$

1-2-7-10-12



= 16

1-3-6-10-12



= 16

Algorithm Forwardgraph(G, k, n, p)

```
{
  cost[n] := 0;
  for j := n-1 to 1 do
  {
    // Compute cost[j]
    let 'l' be a vertex, such that <j, l> is
    an Edge of G and c[j, l] + cost[j+1, l] is
    minimum.
    cost[j] := c[j, l] + cost[l];
    d[j] = l;
  }
  // Find a minimum cost shortest path
    p[1] := 1;
    p[k] := n;
    for j := 2 to k-1 do
      p[j] := d[p[j-1]];
  }
```

* G is multistage graph, k is multistage graph contains k stages, n -> n number of vertices. p - minimum cost shortest path for the

PAGE NO.
DATE / /

given graph in multistage graph

Cost[n] = 0 we need to start from the sink vertex
Initially this is taken as 0.

for J := n-1 to 1 do
↳ cost of calculating cost of each and every node.

P[1] := 1
P[k] := n } → starts from source vertex 1 to sink vertex n

Algorithm backwardgraph(G, k, n, P)
{
  bcost[1] := 0;
  for J := 2 to n do
  {
    // Compute bcost[J]
    let 'l' be a vertex, such that <l, J> is an edge of G and bcost[l] + c[l, J] is minimum
    Cost[J] := bcost[l] + c[l, J];
    d[J] = l;
  }
// Find a minimum cost shortest path
  P[1] := 1;
  P[k] := n;
  for J := k-1 to 2 do
    p[J] := d[p[J+1]];
}

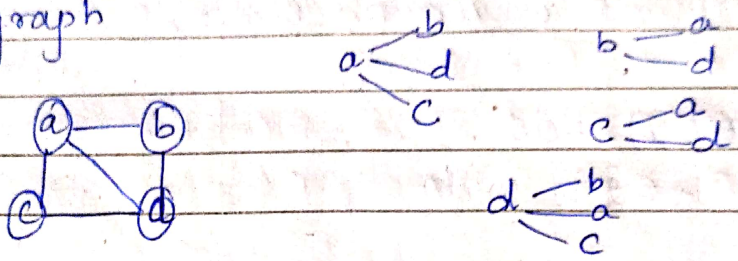
Transitive closure :-

Warshall's Algorithm :-

→ Named after Stephen Warshall, who discovered this algorithm

→ To determine Transitive closure of a Directed graph or all paths in a directed graph using adjacency matrix

→ To find the existence of path between all the pair of vertices in a given weighted connected graph



→ Applicable to both directed and undirected weighted graph

→ To determine transitive closure of a directed graph or all paths in a directed graph using adjacency matrix

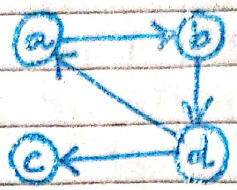
→ Generate transitive closure of a diagraph with the help of DFS & BFS.

→ Application : Data flow and controlflow dependencies, redundancy checking Inheritance testing in object oriented software.

Transitive Closure :-

→ Transitive closure of a directed graph with n vertices can be defined as n x n boolean matrix $T = \{t_{ij}\}$ in which element in the i^{th} row and j^{th} column is 1 if there exist a non trivial path from i^{th} vertex to j^{th} vertex,

Otherwise $t_{ij} = 0$.

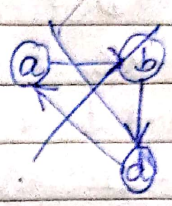


$A =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

T_2

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

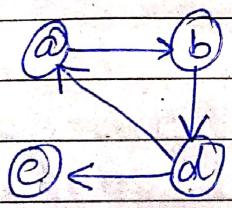


Warshall's Algorithm

→ Named after Stephen Warshall who discovered this algorithm

→ To determine transitive closure of a directed graph or all paths in a directed graph using adjacency matrix

used to find path.



- To chuk whether there is an existence of path between every pair of vertices.

Step 1 :- Generate Adjacency Matrix

$R^{(0)}_2$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Step 2 : Consider path through Vertex a

$R^{(0)}_2$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

$b \to b$ = $b \to a$ & $a \to b$ = 0 & 1 = 0
$b \to c$ = $b \to a$ & $a \to c$ = 0 & 0 = 0
$b \to d$ = 1
$c \to b$ = $c \to a$ & $a \to b$ = 0 & 1 = 0
$c \to c$ = $c \to a$ & $a \to c$ = 0 & 0 = 0
$c \to d$ = $c \to a$ & $a \to d$ = 0 & 0 = 0
$d \to b$ = $d \to a$ & $a \to b$ = 1 & 1 = 1
~~$d \to c$ = $d \to a$ & $a \to c$ = 1 & 0 = 0~~
$d \to d$ = $d \to a$ & $a \to d$ = 1 & 0 = 0

Step 3 : Consider path through vertex b.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$a \to a$ = $a \to b$ & $b \to a$ = 1 & 0 = 0
$a \to c$ = $a \to b$ & $b \to c$ = 1 & 0 = 0
$a \to d$ = $a \to b$ & $b \to d$ = 1 & 1 = 1
$c \to a$ = $c \to b$ & $b \to a$ = 0 & 0 = 0
$c \to c$ = $c \to b$ & $b \to c$ = 0 & 0 = 0

$c \to d = c \to b \;\&\; b \to d = 0 \;\&\; 1 = 0$

$d \to d = d \to b \;\&\; b \to d = 1 \;\&\; 1 = 1$

Step 4:- Consider path through vertex c

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$a \to a = a \to c \;\&\; c \to a = 0 \;\&\; 0 = 0$

$b \to a = b \to c \;\&\; c \to a = 0 \;\&\; 0 = 0$

$b \to b = b \to c \;\&\; c \to b = 0 \;\&\; 0 = 0$

Step 5:- Consider path through vertex d.

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(4)}$ =

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

$a \to a = a \to d \;\&\; d \to a$

$1 \;\&\; 1 = 1$

$a \to c = a \to d \;\&\; d \to c$

$1 \;\&\; 1 = 1$

$b \to a = b \to d \;\&\; d \to a$

$= 1 \;\&\; 1 = 1$

$b \to b = b \to d \;\&\; d \to b$

$= 1 \;\&\; 1 = 1$

$b \to c = b \to d \;\&\; d \to c$

$= 1 \;\&\; 1 = 1$

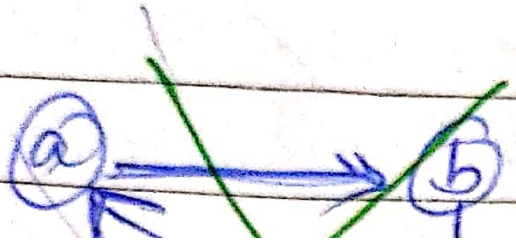
$c \to a = c \to d \;\&\; d \to a = 0 \;\&\; 1 = 0$

$c \to b = c \to d \;\&\; d \to b = 0 \;\&\; 1 = 0$

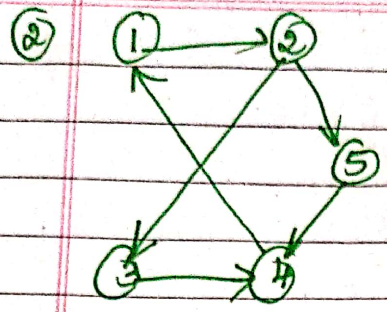
$c \to c = c \to d \;\&\; d \to c = 0 \;\&\; 1 = 0$

$R^{(4)}$:

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1



Adjacency matrix



$R^{(0)}$ =

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

Taking 1 as intermediate

$R^{(0)}$ =

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

$R^{(1)}$ =

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

2→2 : 2→1 & 1→2
0 & 1 = 0

~~2→3 : 2→1 & 1→3~~
~~0 & 0 = 0~~

2→4 : 2→1 & 1→4
0 & 0 = 0

2→5 : 2→1 & 1→5
0 & 0 = 0

3→2 : 3→1 & 1→2
0 & 1 = 0

3→3 : 3→1 & 1→3
0 & 0 = 0

3→4 : ~~3→1 & 1→4~~
~~0 & 0 = 0~~

3→5 : 3→1 & 1→5
: 0 & 0 = 0

4→2 : 4→1 & 1→2
~~0 & 1~~ 1 & 1 = 1

4→3 : 4→1 & 1→3
1 & 0 = 0

4→4 : 4→1 & 1→4
1 & 0 = 0

4→5 : 4→1 & 1→5
1 & 0 = 0

5→2 : 5→1 & 1→2
: 0 & 1 = 0

5→3 : 5→1 & 1→3 : 0 & 0 = 0

5→4 : 1

5→5 : 5→1 & 1→5 : 0 & 0 = 0

Step2 :- $R^{(1)}$ =

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

Taking 2

$R^{(2)}$ =

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

1→1 : 1→2 & 2→1
1 & 0 = 0

1→3 : 1→2 & 2→3
1 & 1 = 1

1→4 : 1→2 & 2→4
: 1 & 0 = 0

1→5 : 1→2 & 2→5
1 & 1 = 1

3→1 : 3→2 & 2→1
0 & 0 = 0

3→3 : 3→2 & 2→3
0 & 1 = 0

3→4 : 3→2 & 2→4 1
: 0 & 0 = 0

3→5 : 3→2 & 2→5
: 0 & 0 = 0

4→3 : 4→2 & 2→3
1 & 1 = 1

4→4 : 4→2 & 2→4
: 1 & 0 = 0

4→5 : 4→2 & 2→5
1 & 1 = 1

5→1 : 5→2 & 2→5
0 & 1 = 0

5→3 : 5→2 & 2→5
0 & 1 = 0

5→4 : 1 5→2 & 2→4
0 & 0 = 0

5→5 : 5→2 & 2→5
0 & 1 = 0

Step 3: $R^{(2)}$ =

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

3 as node

$R^{(3)}$ =

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

1→1 : 1→3 & 3→1 = 1 & 0 = 0

~~1→2 : 1→3 & 3→2 = 1 & 0 = 0~~ 1

1→4 : 1→3 & 3→4 = 1 & 1 = 1

~~1→5 : 1→3 & 3→5 = 1 & 0 = 0~~

2→1 : 2→3 & 3→1 = 1 & 0 = 0

2→2 : 2→3 & 3→2 = 1 & 0 = 0

2→4 : 2→3 & 3→4 = 1 & 1 = 1

~~4→1 : 4→3 & 3→1 =~~

4→4 : 4→3 & 3→4 = 1 & 1 = 1

~~4→5 0~~

5→1 : 5→3 & 3→1 : 0 & 0 = 0

5→2 : 5→3 & 2→3 : 0 & 1 = 0

5→5 : 5→3 & 3→5 : 0 & 0 = 0

Step 4: $R^{(3)}$ =

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

$R^{(4)}$ =

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

4 as node.

1→1 = 1→4 & 4→1 : 1 & 1 = 1

2→1 = 2→4 & 4→1 : 1 & 1 = 1

2→2 = 2→4 & 4→2 : 1 & 1 = 1

3→1 = 3→4 & 4→1 : 1 & 1 = 1

3→2 = 3→4 & 4→2 : 1 & 1 = 1

3→3 = 3→4 & 4→3 : 1 & 1 = 1

3→5 = 3→4 & 4→5 : 1 & 1 = 1

5→1 = 5→4 & 4→1 : 1 & 1 = 1

5→2 = 5→4 & 4→2 : 1 & 1 = 1

5→3 = 5→4 & 4→3 : 1 & 1 = 1

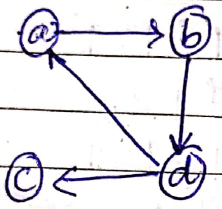
5→5 = 5→4 & 4→5 : 1 & 1 = 1

∴ $R^{(5)}$ =

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

Warshall's algorithm & Time complexity

```
for K=0 to n-1 do
    for i=0 to n-1 do
        for j=0 to n-1 do
(1)         if(p[i,j]=0 and (if(p[i,k]=1 and
                                p[k,j]=1)) then
                p[i,j]=1
            end if
        end for
    end for
end for
```



	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

K=0 p[0,0]=0
i=0 p[0,
j=0

Time Complexity

$$F(n) = \sum_{K=0}^{n-1} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

$$= \sum_{k=0}^{n-1} \sum_{i=0}^{n-1} (n-1-0)+1$$

$$= \sum_{k=0}^{n-1} \sum_{i=0}^{n-1} n$$

$$= \sum_{k=0}^{n-1} n \sum_{i=0}^{n-1} 1$$

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

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

$$= n^2(n-0)+1$$

$$= n^2(n)$$

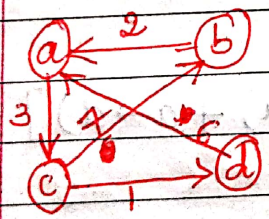
$$= n^3$$

Time Complexity = $\Theta(n^3)$

Floyd - Warshall's Algorithm :-

- To find the shortest distance from each node to all other nodes.
- Floyd's Algorithm is commonly called all pair shortest path problem
- Modified version of warshall's Algorithm.

(1)



Solve the all pair shortest path problem for the given diagraph

Cost Matrix =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Step 1 : Consider shortest path through vertex a.

$R^{(0)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

$R^{(1)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

Diagonal always zero

$(b,c) = \text{Min}[(b \to c), (b \to a + a \to c)]$

$= \text{Min}[\infty, (2+3)]$

$= \text{Min}[\infty, 5]$

$= 5$

$(b,d) = \text{Min}[(b \to d), (b \to a + a \to d)]$

$= \text{Min}[\infty, 2+\infty]$

$= \infty$

$(c,b) = \text{Min}[(c \to b), (c \to a + a \to b)]$

$= \text{Min}[7, \infty + \infty]$

$= 7$

$(c,d) = \text{Min}[(c \to d), (c \to a + a \to d)]$

$= \text{Min}[1, \infty + \infty]$

$= 1$

$(d,b) = \text{Min}[(d \to b), (d \to a + a \to b)]$

$\text{Min}[\infty, 6+\infty]$

$= \infty$

$(d,c) = \text{Min}[(d \to c), (d \to a + a \to c)]$

$= \text{Min}[\infty, 6+3]$

$= 9$

Step 2: Consider the shortest path through b.

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

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

$(a,c) = \text{Min}[(a \to c), (a \to b) + (b \to c)]$
$= \text{Min}[(3), \infty + 5]$
$= \underline{3}$

$(a,d) = \text{Min}[(a \to d), (a \to b) + (b \to d)]$
$= \text{Min}[(\infty, \infty + \infty]$
$= \underline{\infty}$

$(c,a) = \text{Min}[(c \to a), (c \to b) + (b \to a)]$
$= \text{Min}[(\infty), 7 + 2]$
$= \underline{9}$

$(c,d) = \text{Min}[(c \to d), (c \to b) + (b \to d)]$
$= \text{Min}[1, (7 + \infty)]$
$= \underline{1}$

$(d,a) = \text{Min}[(d \to a), (d \to b) + (b \to a)]$
$= \text{Min}[6, \infty + 2]$
$= \underline{6}$

$(d,c) = \text{Min}[(d \to c), (d \to b) + (b \to c)]$
$= \text{Min}[9, \infty + 5]$

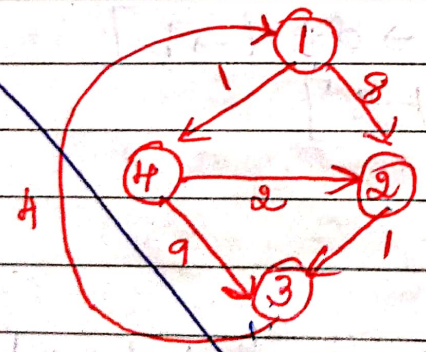
$(C,a) = \text{Min}[(C \to a), C \to d + d \to a]$
$= \text{Min}[9, 6+1]$
$= 7$

$(C,b) = \text{Min}[(C \to b), C \to d + d \to b]$
$= \text{Min}[7, 1+16]$
$= 7$

$\therefore$ Final Matrix =

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

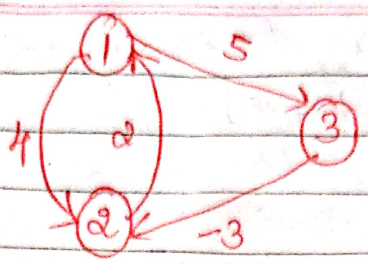
(2)



Cost Matrix =

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	∞	0	∞
4	∞	2	9	0

(2)



$R^{(0)}$ =

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

Step 1: Consider the vertex 1

$R^{(0)}$ =

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

$R^{(1)}$ =

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

$(2 \to 3) = \text{Min}[(2 \to 3), (2 \to 1) + (1 \to 3)]$

$\text{Min}[\infty, 2+5]$

$\text{Min}[\infty, 7]$

7

$(3 \to 2) = \text{Min}[(3 \to 2), (3 \to 1) + (1 \to 2)]$

$= \text{Min}[-3, \infty + 4]$

$= \text{Min}[-3, \infty]$

$= -3$

Step 2 :- Consider vertex 2.

$$R^{(1)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & 7 \\ \infty & -3 & 0 \end{bmatrix} \end{array}$$

$$R^{(2)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{bmatrix} \end{array}$$

$(1 \to 3) = \text{Min}[(1 \to 3), (1 \to 2) + (2 \to 3)]$
$= \text{Min}[5, 4+7]$
$= \text{Min}[5, 11]$
$= 5$

~~$(2 \to 1) = \text{Min}[(2 \to$~~

$(3 \to 1) = \text{Min}[(3 \to 1), (3 \to 2) + (2 \to 1)]$
$= \text{Min}[(\infty), -3+2]$
$= \text{Min}[\infty, -1]$
$= -1$

Step 3 : Consider Vertex 3

$$R^{(3)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{bmatrix} \end{array}$$

$$= \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 2 & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{bmatrix} \end{array}$$

$(1 \to 2) = \text{Min}[(1 \to 2), [(1 \to 3) + (3 \to 2)]$
$= \text{Min}[4, [5 + (-3)]$
$= \text{Min}[2]$
$= 2$

$(2 \to 1) = \min[(2 \to 1), (2 \to 3) + (3 \to 1)]$

$= \min[2, (7 + (-1))]$

$= \min[2, 6]$

$= 2$

Final Matrix =

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

Knapsack Algorithm & Analysis using dynamic programming :- (0/1) Knapsack.

Aim :- To find the optimal solution for the Knapsack problem using dynamic programming

Algorithm : Knapsack (n, m, w, p, v)

// To find the optimal solution for the Knapsack problem using dynamic programming

// Input :- n - Number of objects to be selected
m - Capacity of Knapsack
w - Weights of all the objects
p - Profits of all the objects.

// Output :- v - optimal solution for the number of objects selected with specified remaining capacity

```
for i=0 to n do
    for j=0 to m do
        if (i=0 or j=0)
            V[i,j]=0
        else if (W[i]>j)
            V[i,j] = V[i-1,j]
```

else

$V[i,j] = \max(V[i-1,j], V[i-1, j-w[i]] + P[i])$

end if

end for

end for

① Apply dynamic programming technique to the following instance of knapsack problem with capacity $W = 5$

Item i	w_i weight	P_i Profit
1	2	12$
2	1	10$
3	3	20$
4	2	15$

$n = 4$

$$V[i,j] = \begin{cases} 0 & \text{if } i = j = 0 \\ V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], V[i-1, j-w_i] + p_i) & \text{if } w_i \le j \end{cases}$$

$j \Rightarrow$ remaining capacity

Step 1 :- Construct Adjacency Table

$n+1$ and $m+1$

↳ 5 rows ↳ 6 columns

$j \longrightarrow$

i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

$n = 4$

profit table

Top - down Approach

$i = 0 \quad j = 0$

$V[i, j] = 0$

Step3:- Consider the first object.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

	wi	j	wi > j	i j
Condition 2	2	1	2 > 1 (yes)	V[1,1] = V[i-1, j] = V[0,1] = 0
Condition 3	2	2	2 > 2 (No)	= V[1,2] = Max(v[0,2], v[0, 2-2] + 12)
			2 <= 2 (yes)	= Max(0, 0+12)
				= Max(0,12)
				= 12
	2	3	2 > 3 (No)	= V[1,3] = Max(v[0,3], v[0,1]+12)
			2 <= 3 (yes)	= Max(0, 0+12)
				= Max(0,12)
				= 12
	2	4	2 > 4 (No)	V[1,4] = Max(v[0,4], v[0,2]+12)
			2 <= 4 (yes)	= Max(0, 12)
				= 12
	2	5	2 > 5 (No)	V[1,5] = Max(v[0,5], v[0,3]+12)
			2 <= 5 (yes)	= Max(0, 0+12)
				= 12

Step 4 :-

wi	j	wi > j	
1	1	1 > 1 x 1 <= 1	V[2,1] = Max(V[1,1], V[1,0]+10) = Max(0, 0+10) = 10
1	2	1 > 2 x 1 <= 2 ✓	V[2,2] = Max(V[1,2], V[1,1]+10] = Max(12, 0+10) = 10
1	3	1 > 3 1 <= 3	V[2,3] = Max(V[1,3] V[1,2]+10] = Max(12, 12+10) = (12, 22) = ~~12~~ 22
1	4	1 > 4 1 <= 4	V[2,4] = Max(V[1,4) V[1,3] +10] = Max(12, 12+10) = Max(12, 22) = 22
1	5	1 > 5	V[2,5] = Max(V[1,5) V(1,4)+10] = Max(12, 12+10] = 22

Step 5 :

wi	j	wi > j	
3	1	3 > 1 ✓	V[3,1] = V[2,1] = 10
3	2	3 > 2 ✓	V[3,2] = V[2,2] = 12
3	3	3 > 3 x 3 <= 3	V[3,3] = Max[V[2,3], V[2,0]+20 = Max(22, 0+20) = 22
3	4	3 > 4 3 <= 4 ~~x~~	V[3,4] = Max[V[2,4], V[2,1]+20] = Max[22, 10+20] = ~~Max~~ 30
3	5	3 <= 5	V[3,5] = Max[V[2,5], V[2,2]+20] = Max[22, 12+20] = 32

Step 6

w_i	j	$w_i > j$	
2	1	2>1 ✓	V[4,1] = V[3,1] = 10
2	2	2<=2	V[4,2] = Max(V[3,2], V[3,0]+15) = Max(12,15) = 15
2	3	2<=3	V[4,3] = Max[V[3,3], V[3,1]+15] = Max(22,25) = 25
2	4	2<=4	V[4,4] = Max.(V[3,4], V[3,2]+15) = Max(30, 12+15) = 30
2	5	2<=5	V[4,5] = Max[V(3,5), V(3,3)+15] = Max(32, 22+15) = 37

	j → M					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
n=4 ↑	0	10	15	25	30	37

↳ first found

V[4,5] = 37

= 37 − 15 = 22

(4) = 5 − 2 = 3

(2) V[2,3] = 22 − 10 = 12

Weight 3 − 1 = 2

(1) V[1,2] = 12 − 12 = 0

Weight 2 − 2 = 0

$(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$

Knapsack problem using memory functions in dynamic programming

Algorithm: MFKnapsack(i, j)
// Input: A non-negative integer i indicating the number of the first items being considered and a non-negative integer j indicating the Knapsack capacity.
// Output: The value of an optimal feasible subset of the first i items

```
if (V[i,j] < 0)
    if (w[i] > j)
        value ← f(i-1, j)
    else
        value ← max(f(i-1, j), f(i-1, j-w[i]) + p[i])
    endif
    V[i,j] ← value
end if
```

(1) Apply the memory function method to solve the following instance of Knapsack problem with capacity M = 5.

Item	weight	Profit
1	2	12$
2	1	10$
3	3	20$
4	2	15$

Bottom up approach

Solution $n = 4$

$M = 5$

$(w_1, w_2, w_3, w_4) = (2, 1, 3, 2)$

$(P_1, P_2, P_3, P_4) = (12, 10, 20, 15)$

$j \longrightarrow M$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	–	12	22	–	22
3	0	–	–	22	–	32
4	0	–	–	–	–	37

$V[4,5]$

Profit Table

$N = 4$

$V[i,j] = 0$

~~$V[4,5]$ $i=4$ $j=5$ $w_4=2$ $w_4 \le j$ $P_4=15$~~

1	$V[4,5]$	$i=4$ $j=5$ $w_4=2$ $P_4=15$	$w_4 \le j$	$V[4,5] = \max[V[3,5], V[3,3]+15]$ $\max[32, 22+15] = 37$
2.	$V[3,5]$	$i=3$ $j=5$ $w_3=3$ $P_3=20$	$w_3 \le j$	$V[3,5] = \max[V[2,5], V[2,2]+20]$ $\max\{12, 12+20\} = 32$
3.	$V[3,3]$	$i=3$ $j=3$ $w_3=3$ $P_3=20$	$w_3 \le j$	$V[3,3] = \max[V[2,3], V[2,0]+20] = 22$ $\max[12, 0+20]$
4.	$V[2,5]$	$i=2$ $j=5$ $w_2=1$ $P_2=10$	$w_2 < j$	$V[2,5] = \max[V[1,5], V[1,4]+10] = 22$ $\max[12, 12+10]$
5	$V[2,2]$	$i=2$ $j=2$ $w_2=1$ $P_2=10$	$w_2 < j$	$V[2,2] = \max[V[1,2], V[1,1]+10\} = 12$ $\max\{10, 0+10\}$
6.	$V[2,3]$	$i=2$, $j=3$ $w_2=1$ $P_2=10$	$w_2 < j$	$V[2,3] = \max[V[1,3], V[1,2]+10\} = 22$ $\max[12, 12+10]$
7	$V[2,0]$	$i=2$ $j=0$ $w_2=1$ $P_2=10$		$V[2,0] = 0$

8 V[1,5] i=1 j=5 w_1=2 P_1=12 w_1<j V[1,5] = Max[V[0,5], V[0,3]+12] = 12

9 V[1,4] i=1 j=4 w_1=2 P_1=12 w_1<j V[1,4] = Max[V[0,4], V[0,2]+12] = 12

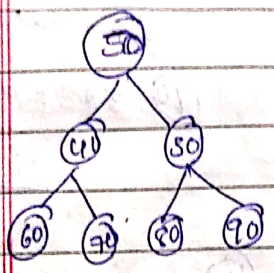
10 V[1,2] i=1 j=2 w_1=2 P_1=12 w_1=j V[1,2] = Max[V[0,2], V[0,0]+12] = 12

11 V[1,1] i=1 j=1 w_1=2 P_1=12 w_1>j V[1,1] = V[0,1] = 0

12 V[1,3] i=1 j=3 w_1=2 P_1=12 w_1<j V[1,3] = Max[V[0,3] V[0,1]+12] = 12

∴ The optimal Solution is 37

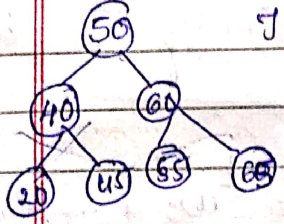
Optimal binary Search tree :-



I want to search for the element 80.

50 – 40 – 60 – 70 less left side, right greater than

50 – 30 – 80 ✓

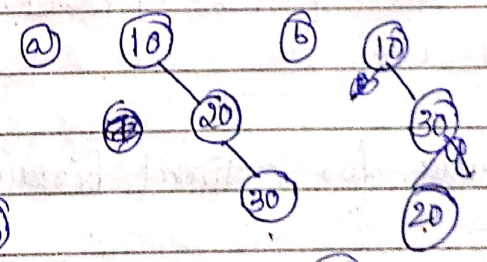


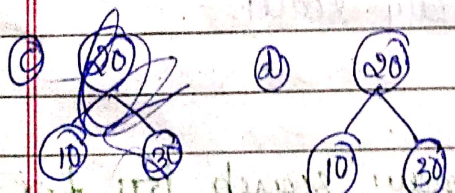
The element to search 65

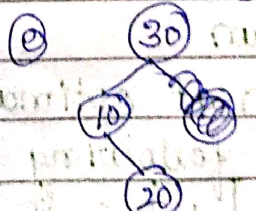
65 key found. This is called binary search tree.

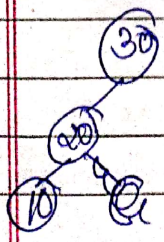
10, 20, 30

3 2 5



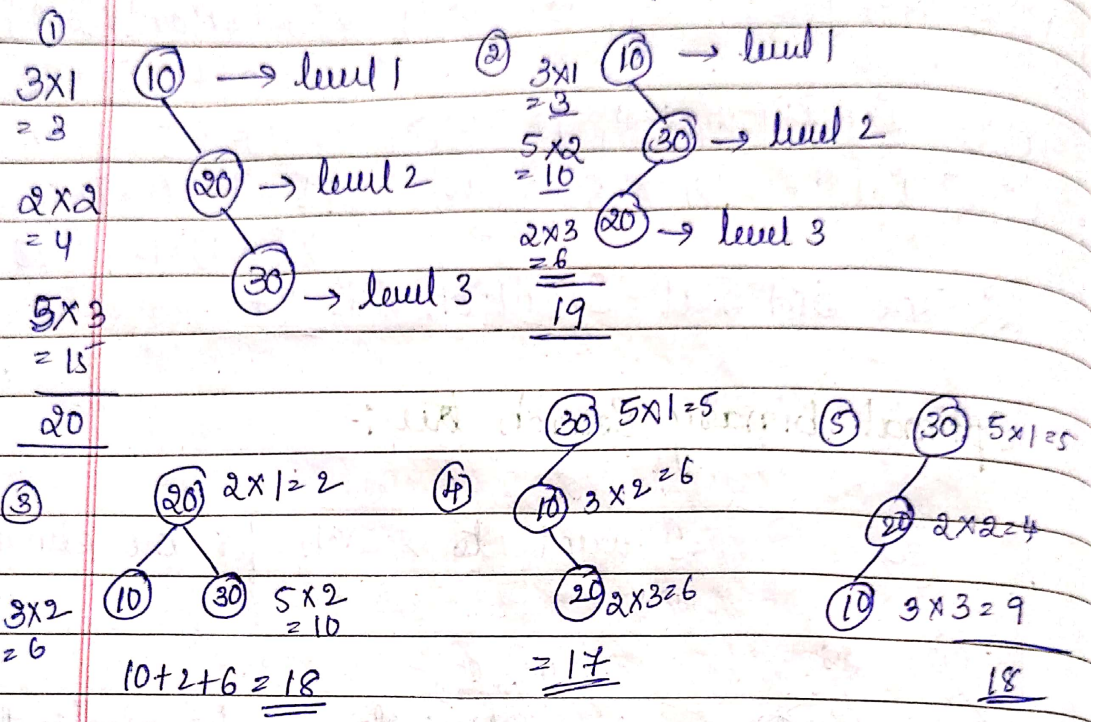






$\frac{2^n C_n}{n+1} = \frac{2^3 C_3}{4} = \frac{6C_3}{4} = 5$

This all are the method to find out the binary search tree. Now we need to find out the optimal binary search tree we have to take the frequencies



which is minimum cost tree ④

Tree ④ we call it as optimal binary search tree.

$$C[i,j] = \min_{i \le k \le j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} w_s \rightarrow \text{frequencies}$$

Formula for optimal binary search

Problem

① Construct optimal binary search tree for the following data

	1	2	3	4
Keys	10	20	30	40
Frequency	4	2	6	3

$$C(i,j) = \min_{i \le k \le j} \{C(i,k-1) + C(k+1,j)\} + \sum_{s=i}^{j} P_s$$

for $1 \le i \le j \le n$

① Construct an optimal binary search tree for given set of keys.

Key	A	B	C	D
Probability	0.1	0.2	0.4	0.3

Initial table will be $C(i, i-1) = 0$ for $1 \le i \le n+1$ & $C(i,i) = P_i$

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

$C(1,0) = 0$
$C(2,1) = 0$
$C(3,2) = 0$
$C(4,3) = 0$
$C(5,4) = 0$

$C(i,i) = P_i$
$C(1,1) = 0.1$
$C(2,2) = 0.2$
$C(3,3) = 0.4$
$C(4,4) = 0.3$

Root table

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

$$C(1,2) = \min \begin{cases} \text{for } k=1 : C[1,0] + C[2,2] + \sum_{s=1}^{2} P_s \\ \text{for } k=2 : C[1,1] + C[3,2] + \sum_{s=1}^{2} P_s \end{cases}$$

$$= \min [0 + 0.2 + 0.3,\ 0.1 + 0 + 0.3]$$
$$= \min [0.5,\ 0.4]$$
$$= 0.4$$

$$C(i,j) = \min_{i \le k \le j} \{C(i,k-1) + C(k+1,j)\} + \sum_{s=i}^{j} P_s$$

for $1 \le i \le j \le n$

(1) Construct an optimal binary search tree for given set of keys.

Key	A	B	C	D
Probability	0.1	0.2	0.4	0.3

Initial table will be $C(i, i-1) = 0$ for $1 \le i \le n+1$ &

$C(i, i) = P_i$

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

$C(1,0) = 0$

$C(2,1) = 0$

$C(3,2) = 0$

$C(4,3) = 0$

$C(5,4) = 0$

$C(i,i) = P_i$

$C(1,1) = 0.1$

$C(2,2) = 0.2$

$C(3,3) = 0.4$

$C(4,4) = 0.3$

Root table

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

$$C(1,2) = \min \begin{cases} \text{for } k=1 : C[1,0] + C[2,2] + \sum_{s=1}^{2} P_s \\ \text{for } k=2 : C[1,1] + C[3,2] + \sum_{s=1}^{2} P_s \end{cases}$$

$$= \min [0 + 0.2 + 0.3,\ 0.1 + 0 + 0.3]$$

$$= \min [0.5,\ 0.4]$$

$$= 0.4$$

$$C(2,3) = \begin{cases} K=2 : C[2,2] + C[3,3] + \sum_{s=2}^{3} P_s \\ K=3 : C[2,2] + C[4,3] + \sum_{s=2}^{3} P_s \end{cases}$$

$= \min [0 + 0.4 + 0.6,\ 0.2 + 0 + 0.6]$

$= \min [1.0,\ 0.8]$

$= \underline{0.8}$

$$C(3,4) = \begin{cases} K=3 : C[3,2] + C[4,4] + \sum_{s=3}^{3} P_s \\ K=4 : C[3,3] + C[5,4] + \sum_{s=3}^{4} P_s \end{cases}$$

$= \min [0 + 0.3 + 0.7,\ 0.4 + 0 + 0.7]$

$= \min [1.0,\ 1.1]$

$= \underline{1.0}$

$$C(1,3) = \min \begin{cases} K=1 : C[1,0] + C[2,3] + \sum_{s=1}^{3} P_s \\ K=2 : C[1,1] + C[3,3] + \sum_{s=1}^{3} P_s \\ K=3 : C[1,2] + C[4,3] + \sum_{s=1}^{3} P_s \end{cases}$$

$= \min [0 + 0.8 + 0.7,\ 0.1 + 0.4 + 0.7,\ 0.4 + 0 + 0.7]$

$= \min [1.5,\ 1.2,\ 1.1]$

$= \underline{1.1}$

$$C(2,4) = \min \begin{cases} K=2 : C[2,1] + C[3,4] + \sum_{s=2}^{4} P_s \\ K=3 : C[2,2] + C[4,4] + \sum_{s=2}^{4} P_s \\ K=4 : C[2,3] + C[5,4] + \sum_{s=2}^{4} P_s \end{cases}$$

$= \min [0 + 1.0 + 0.9,\ 0.2 + 0.3 + 0.9,\ 0.8 + 0 + 0.9]$

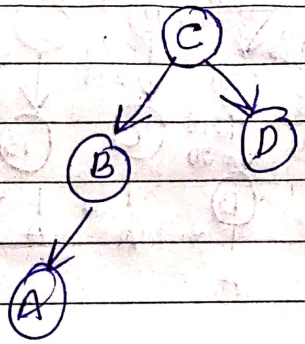
$= \min [1.9,\ 1.4,\ 1.7]$

$= \underline{1.4}$

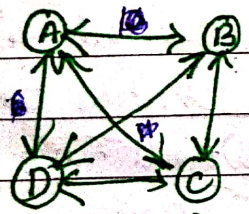
Thus the average number of key comparisons in the optimal tree is equal to 1.7

PAGE NO.
DATE / /

- Since R(1,4) = 3, the root of the optimal tree contains the third key ie C.
- Since its a binary search tree, its left subtree is made up of keys A and B, and its right subtree contains just the key D.
- To find the specific structure of these subtrees
- On the root table since R(1,2) = 2, the root of the optimal tree containing A and B is B, with A being its left child
- Since R(4,4) = 4 the root of this one-node optimal tree is its only key D.
- The below figure represents the optimal bst



Travelling Salesman Problem using dynamic programming



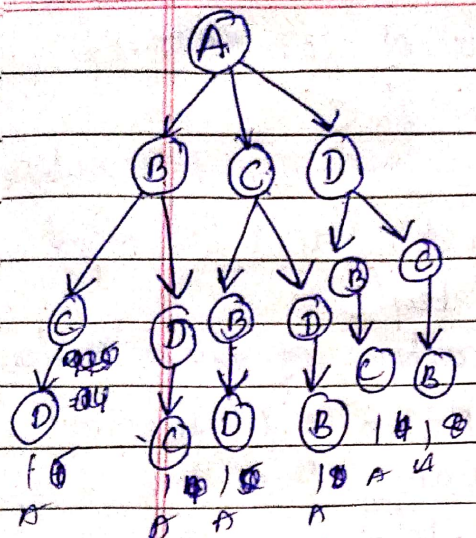
	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0

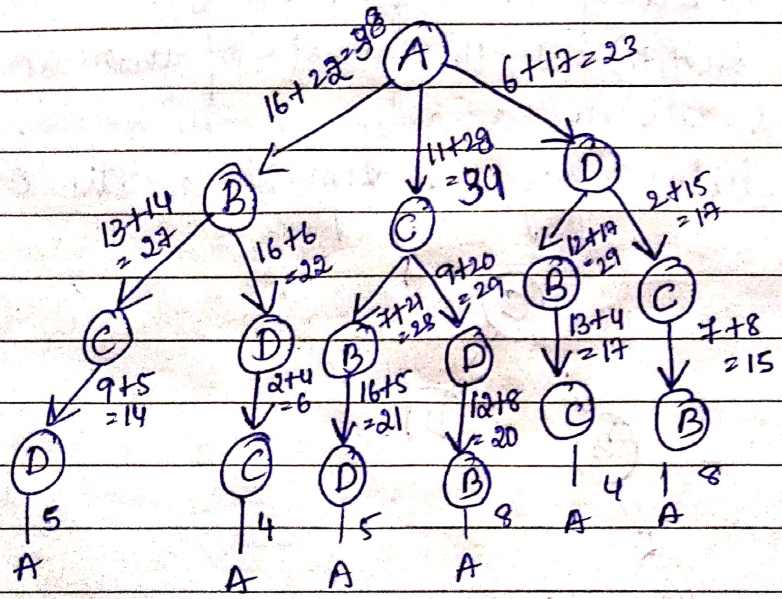
- It should form a cycle.
- Visit all the cities exactly once and come back to its original city

Eg A → B → C → D → A ✓

A → D → C → B → A ✓

A → D → C → B → D → A ✗





Minimum 6+17 = 23

$g(i,S) = \min_{j \in S} [w(i,j) + g(j, \{S-j\})]$

$i \rightarrow$ starting vertex

$S \rightarrow$ ~~Sate~~ Set of Vertices the salesm will visit exactly once

$i = A \quad g(A,\{B,C,D\}) = \min[w(A,B) + g(B,\{C,D\}), w(A,C) + g(C,\{B,D\}), w(A,D) + g(D,\{B,C\})]$

$g(B,\{C,D\}) = 16+22 = 38$

$= w(A,C) + g(C,\{B,D\}) = 11+28 = 39$

$= w(A,D) + g(D,\{B,C\}) = 6+17 = 23$ → min

$g(B,\{C,D\}) = \min \begin{matrix} w(B,C) + g(C,\{D\}) = 13+14 = 27 \\ w(B,D) + g(D,\{C\}) = 16+6 = 22 \end{matrix}$

PAGE NO.
DATE / /

$g(C,\{D\}) = w(C,D) + g(D,\phi)$ (D, A)

$= 9+5 = 14$

$g(x,\phi) = w(x,1)$

$g(D,\{C\}) = w(D,C) + g(C,\phi)$ (C, A)

$= 2+4 = 6$

$g(C,\{B,D\}) = \min[w(C,B) + g(B,\{D\})] = 7+21 = 28$

$\min[w(C,D) + g(D,\{B\})]$

$= 9+20 = 29$

$g(\{B,D\}) = \min[w(B,D) + g(D,\phi)]$

$= 16+5 = 21$

$g(D,\{B\}) = \min[w(D,B) + g(B,\phi)]$

$= 12+8 = 20$

$A \to D \to C \to B = 23$

~~$w(A,D)$~~ A

$g(D,\{B,C\}) = w(D,B) + g(B,\{C\}) = 12+4 = 16$ (17) (29)

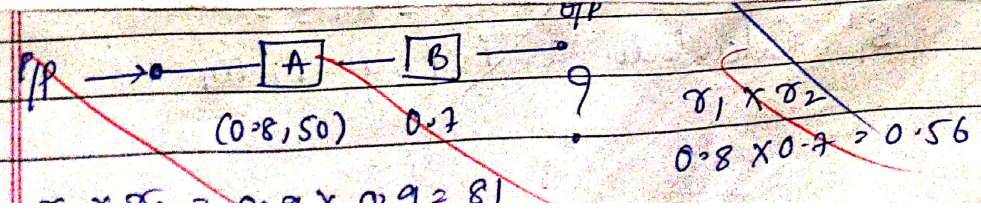
$= w(D,C) + g(C,\{B\}) = 2+15 = 17$

$g(B,\{C\}) = \min[w(B,C) + g(C,\phi)]$

$= 13+4 = 17$

$g(C,\{B\}) = \min[w(C,B) + g(B,\phi)]$

$= 7+8 = 15$



1 Sorting by Counting

As a first example of applying the input-enhancement technique, we discuss its application to the sorting problem. 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*** (Figure 7.1).

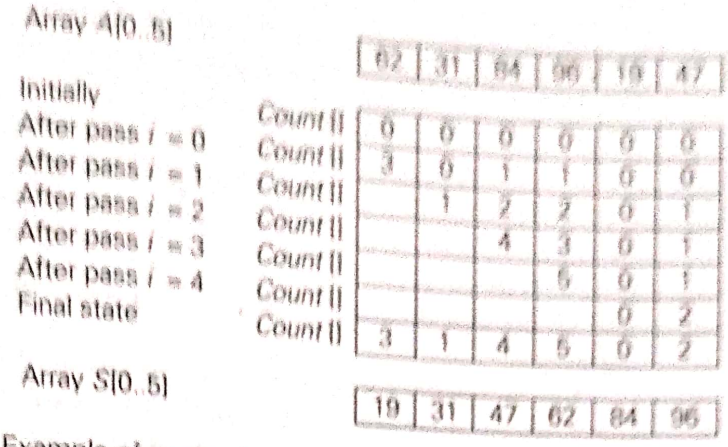


FIGURE 7.1 Example of sorting by comparison counting

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
```

What is the time efficiency of this algorithm? It should be quadratic because the algorithm considers all the different pairs of an n-element array. More formally, the number of times its basic operation, the comparison $A[i] < A[j]$, is executed is equal to the sum we have encountered several times already:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Thus, the algorithm makes the same number of key comparisons as selection sort and in addition uses a linear amount of extra space. On the positive side, the algorithm makes the minimum number of key moves possible, placing each of them directly in their final position in a sorted array.

The counting idea does work productively in a situation in which elements to be sorted belong to a known small set of values. Assume, for example, that we have to sort a list whose values can be either 1 or 2. Rather than applying a general sorting algorithm, we should be able to take advantage of this additional

information about values to be sorted. Indeed, we can scan the list to compute the number of 1's and the number of 2's in it and then, on the second pass, simply make the appropriate number of the first elements equal to 1 and the remaining elements equal to 2. More generally, if element values are integers between some lower bound l and upper bound u, we can compute the frequency of each of those values and store them in array $F[0..u-l]$. Then the first $F[0]$ positions in the sorted list must be filled with l, the next $F[1]$ positions with $l+1$, and so on. All this can be done, of course, only if we can overwrite the given elements.

Let us consider a more realistic situation of sorting a list of items with some other information associated with their keys so that we cannot overwrite the list's elements. Then we can copy elements into a new array $S[0..n-1]$ to hold the sorted list as follows. The elements of A whose values are equal to the lowest possible value l are copied into the first $F[0]$ elements of S, i.e., positions 0 through $F[0]-1$; the elements of value $l+1$ are copied to positions from $F[0]$ to $(F[0]+F[1])-1$, and so on. Since such accumulated sums of frequencies are called a distribution in statistics, the method itself is known as ***distribution counting***.

EXAMPLE Consider sorting the array

13	11	12	13	12	12

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

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

Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to $n-1$, the distribution values must be reduced by 1 to get corresponding element positions.

It is more convenient to process the input array right to left. For the example, the last element is 12, and, since its distribution value is 4, we place this 12 in position $4-1=3$ of the array S that will hold the sorted list. Then we decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array. The entire processing of this example is depicted in Figure 7.2.

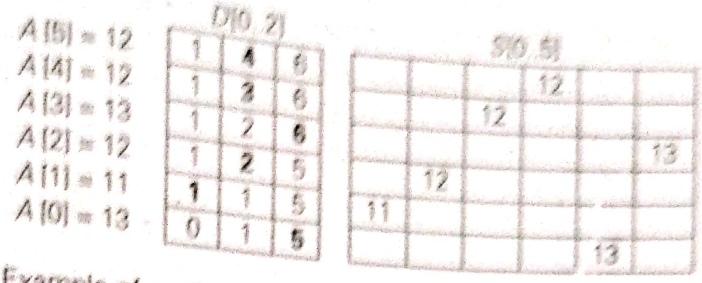


FIGURE 7.2 Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

Here is pseudocode of this algorithm.

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
```

Assuming that the range of array values is fixed, this is obviously a linear algorithm because it makes just two consecutive passes through its input array A. This is a better time-efficiency class than that of the most efficient sorting algorithms—mergesort, quicksort, and heapsort—we have encountered. It is important to remember, however, that this efficiency is obtained by exploiting the specific nature of inputs for which sorting by distribution counting works, in addition to trading space for time.

Exercises 7.1

1. What is prestructuring? What is dynamic programming?
2. What is the time efficiency of the comparison-counting algorithm?
3. Assuming that the set of possible list values is $\{a, b, c, d\}$, sort the following list in alphabetical order by the distribution-counting algorithm:

   $b, c, d, c, b, a, a, b.$

Horspool's Algorithm

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

$s_0 \quad \cdots \quad c \quad \cdots \quad s_{n-1}$

B A R B E R

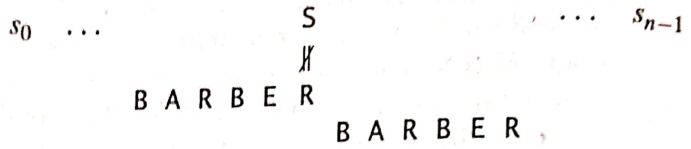
Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired.

If a mismatch occurs, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size

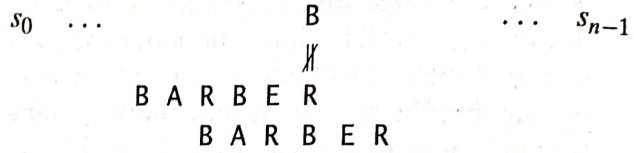
of such a shift by looking at the character c of the text that is aligned against the last character of the pattern. This is the case even if character c itself matches its counterpart in the pattern.

In general, the following four possibilities can occur.

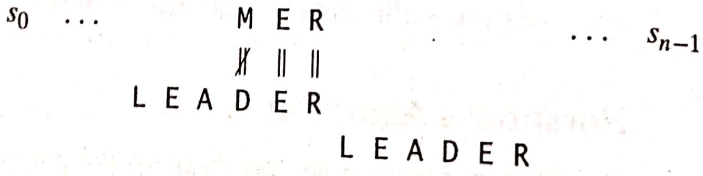
Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):



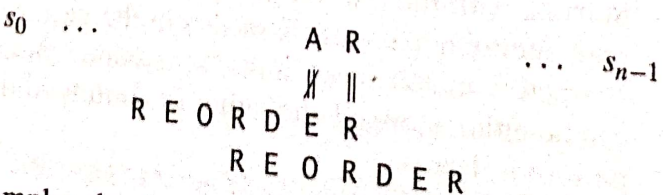
Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:



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:



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:



These examples clearly demonstrate that right-to-left character comparisons can lead to farther shifts of the pattern than the shifts by only one position

always made by the brute-force algorithm. However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority. Fortunately, the idea of input enhancement makes repetitive comparisons unnecessary. We can precompute shift sizes and store them in a table. The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters. (Note that no other information about the text in which eventual searching will be done is required.) The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad (7.1)$$

For example, for the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

Here is a simple algorithm for computing the shift table entries. Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step $m-1$ times: for the jth character of the pattern ($0 \le j \le m-2$), overwrite its entry in the table with $m-1-j$, which is the character's distance to the last character of the pattern. Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence—exactly as we would like it to be.

ALGORITHM *ShiftTable*($P[0..m-1]$)

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

Now, we can summarize the algorithm as follows:

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 (then

stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

Here is pseudocode of Horspool's algorithm.

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 first 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
```

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R               B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                   B A R B E R
```

A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in $O(nm)$ (Problem 4 in this section's exercises). But for random texts, it is in $\Theta(n)$, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm. In fact, as mentioned, it is often at least as efficient as its more sophisticated predecessor discovered by R. Boyer and J. Moore.