UNIT - III
DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE

Dynamic Programming

Concept

It is a technique for solving problems with overlapping sub problems. The smaller sub problems are solved only once and recording the result in a table from which the solution to the original problem is obtained.

General Method

- It is an algorithm design method that can be used when solution to a problem can be viewed as the result of sequence of decisions.
- Enumerate all decision sequences and then pick out the best.
- Optimal sequence of decision is obtained.
- Many decision sequences may be generated.

Example

Consider a Fibonacci series of n numbers.

0, 1, 1, 2, 3, 5, 8……

- Initial condition for Fibonacci series

$$F(0) = 0$$
$$F(1) = 1$$

- Computation of $F(n)$ is expressed in terms of its smaller and overlapping sub problems.

$$F(n) = F(n-1) + F(n-2) \text{ for } n >= 2$$

- It can simply fill elements of a one dimensional array with $(n + 1)$ consecutive elements of $F(n)$ by starting in view of initial conditions with 0 and 1.

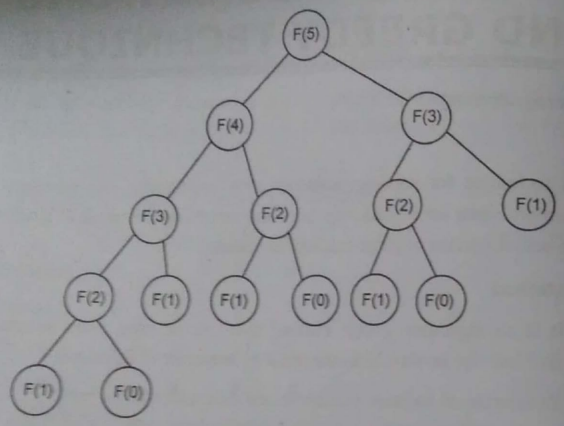


Fig. 3.1.Tree of recursive calls for computing the Fibonacci number for n =5

Algorithm for Computing Fibonacci series of 'n' nos

Algorithm Fib (*n*)

// Computes the n^{th} Fibonacci no iteratively by using its definition.

// Input –A non negative integer *n*.

// Output – The n^{th} Fibonnaci no.

```
F[0] = 0
F[1] = 1
For (i = 2; i <= n; i ++)
    F[i] = F[i – 1] + F[i – 2]
return F[n];
```


Principle of Optimality

- It is the basic principle of dynamic programming, which was developed by Richard Bellman.
- The optimal path has the property that whatever the initial conditions and control variables (choices) over some initial period.
- The control (or decision variables) chosen over the remaining period must be optimal for the remaining problem, with the state resulting from the early decisions taken to be the initial condition.

3.1. COIN CHANGING PROBLEM

Concept

In this problem our goal is to make change for an amount using least number of coins from the available denominations.

Procedure

- We will find the minimum no. of coins required for the amount p where p = 1, 2, ..., A where A is the amount for which we are making the change.
- For every p we will first set min. no. of coins required i.e., min = INFINITY
- This means we initially don't know how many coins will be needed.
- Then we check each denomination coins and see if it can be used to get the solution.
- If its possible then we update the min and coin variables. where, min = minimum no. of coins required for making change for amount p
- coin = first index of the coin in the solution else we move on

Formula

To solve this problem we will use the following formula

$$C[p] = \begin{cases} 0 & \text{if } p = 0 \\ \min_{i: d_i \le p} \{1 + C[p - d_i]\} & \text{if } p > 0 \end{cases}$$

$C[p]$ denotes the minimum number of coins required to make change for an amount p using given denomination coins $d[i]$ where selected denomination is not greater than the amount p.

Algorithm

if d[i] <= p then
if 1 + C[p - d[i]] < min then
min = 1 + C[p - d[i]]
coin = i

Example

Minimum number of coins

To find the min. no. of coins for amount Rs. 6 we have to take the value from C[p] array.

So, minimum coins required to make change for amount Rs. 6 = C[6] = 2.

Solution

	0	1	2	3	4	5	6
C[p]	0	1	1	2	2	1	2
	0	1	2	3	4	5	6
S[p]	0	1	2	1	2	3	1

Coins in the optimal solution

To know the coins selected to make the change we will use the $S[p]$ array

Step 1: *Set a = A*

Step 2: *If a > 0 then*

Print d[S[a]]

else STOP

Step 3: *Set a = a - d[S[a]]*

Repeat step 2

Initially, a = 6

is a > 0

i.e., 6 > 0

YES

Print d[S[a]] = d[S[6]] = d[1]

i.e, Print 1

Set a = a - d[S[a]] = 6 - 1 = 5

So, the first coin is 1

Now, a = 5

is a > 0

i.e., 5 > 0

YES

Print d[S[a]] = d[S[5]] = d[3]

i.e, Print 5

Set a = a - d[S[a]] = 5 - 5 = 0

So, the second coin is 5

And now a = 0

so, we STOP

Therefore, to make change of amount Rs. 6 the shopkeeper will need minimum 2 coins and the coins will be Rs. 1 and Rs. 5

Time Complexity

Time complexity of this algorithm is $O(nA)$ where n is the total number of different denomination of the coins and A is the amount for which we are making change.

3.2. COMPUTING A BINOMIAL COEFFICIENT

Concept

The binomial coefficient denoted $C(n, k)$ or $\binom{n}{k}$ is the number of combinations (subsets) of k elements from an n-element set $(0 \le k \le n)$.

The name "Binomial coefficient" comes from the participation of these numbers in the binomial formula.

Binomial Coefficient Formula

$$(a+b)^n = C(n,0)\,a^n + C(n,1)\,a^{n-1}b + \ldots\ldots + C(n,i)\,a^{n-i}b^i + \ldots\ldots C(n,n)\,b^n.$$

- Binomial coefficient has several properties.
- The 2 important properties are

1. $C(n, k) = C(n-1, k-1) + C(n-1), k)$ for $n > k > 0$

2. $C(n, 0) = C(n, n) = 1$

3.2.1. PROCEDURE FOR COMPUTING BINOMIAL COEFFICIENT

- We record the values of the binomial coefficient in a table of $n + 1$ rows and $k + 1$ columns numbered from 0 to n and from 0 to k.
- Compute the value of $C(n, k)$ row by row starting with row 0 and ending with row n.
- Each row i $(0 \le i \le n)$ is filled left to right.

Put 1 for 0^{th} row and 0^{th} column using the formula $\boxed{C(n, 0) = 1}$

- Put 1 for all the diagonals.

$C(i, i) = 1$ for $\boxed{0 \le i \le k.}$

- Compute other entries using the following formula

$\boxed{C(n, K) = C(n-1, k-1) + C(n-1, k) \text{ for } n > k > 0.}$

ie adding the contents of cell in the preceding row and the previous column and in the preceding row and the same column.

3.2.2. TABLE FOR COMPUTING THE BINOMIAL COEFFICIENT

- Table contains 0 to n rows and 0 to k columns.
- 0^{th} row.

  $C(0, 0) = 1$

- I row.

  $C(1, 0) = 1$ using the formula $\boxed{C(n, 0) = 1}$

  $C(1, 1) = 1$ using the formula $\boxed{C(n, n) = 1}$

- II row

  $C(2, 0) = \boxed{1}$

  $C(2, 1) = C(1, 0) + C(1, 1)$

  $= 1 + 1$

  $= \boxed{2}$

  $C(2, 2) = 1$

- III row

  $C(3, 0) = 1$

  $C(3, 1) = C(2, 0) + C(2, 1) = 1 + 2 = \boxed{3}$

  $C(3, 2) = C(2, 1) + C(2, 2) = 2 + 1 = \boxed{3}$

  $C(3, 3) = \boxed{1}$

∴ All the values are specified in the table.

	0	1	2	3	……$k-1$	K
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
⋮						
K	1					
⋮						
$n-1$	1					$C(n-1, k-1)$ + $C(n-1, k)$
n	1					

Fig. 3.2. Table for computing the binomial coefficient

3.2.3. ALGORITHM FOR BINOMIAL COEFFICIENT

```
Algorithm binomial (n, k)
{
  // Computes C (n, k) by the dynamic programming algorithm
  // Input: A pair of non negative integers n ≥ k ≥ 0
  // Output: The value of C (n, k)
   for i = 0 to n do
     for j = 0 to min (i, k) do
       if (j = 0 or j = k)
          C [i, j] = 1
       else
          C [i,j] = C [i – 1, j – 1] + C [i – 1, j]
return C [n, k]
```


3.2.4. TIME COMPLEXITY FOR BINOMIAL COEFFICIENT ALGORITHM

- Algorithm's basic operation is addition.
- Let A (n, k) be the total number of addition made by the algorithm.
- It requires just one addition
- In the table, the first $k + 1$ rows of the table form a triangle, while the remaining n-k rows form a rectangle.
- The recurrence relation of binomial coefficient A(n, k) into 2 parts.

$$A(n,k) = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1$$

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

$$= \frac{(k-1)\,k}{2} + k\cdot(n-k) \Rightarrow \frac{k^2-k}{2} + kn - k^2$$

$$= 0\,(n \cdot k) \Rightarrow \frac{k^2-k+2kn-2k^2}{2} \Rightarrow \frac{-k^2+2kn-k}{2}$$

$$\therefore A(n,k) = 0\,(n \cdot k) \qquad -k(k+1)+2kn$$

3.2.5. MERITS

- Computing a binomial coefficient is a standard example of applying dynamic programming to an optimizing problem.

3.3. FLOYD'S ALGORITHM

Basic Concept

Given a weighted graph, the all pairs shortest path problem is used to find the distance from each vertex to all other vertices.

The graph may be either directed or undirected graph.

3.3.1. PROCEDURE FOR FLOYD'S ALGORITHM

- Let G = (V, E) be a directed graph with n vertices.
- Let cost of adjacency matrix for G, such that cost $(i, i) = 0$, $1 \le i \le n$.
- Cost (i, j) is the length of edge (i, j) if $(i, j) \in$ E (G).

- Cost $(i, j) = \alpha$ if $i \neq j$ and $(i, j) \notin E(G)$
- Examine a shortest path 'i' to 'j' if $i \neq j$.
- The path originates at vertex i and goes through some intermediate vertices and terminate at vertex j.
- It computes the distance matrix of a weighted graph with n vertices through a series of n by n matrices.
- The series of n by n matrices are $D^{(0)}, D^{(1)} \ldots D^{(k-1)}, D^{(k)} \ldots D^{(n)}$.
- The element $D_{ij}^{(k)}$ in the i^{th} row and j^{th} column of matrix D^k $(k = 0,1,2,\ldots n)$ is equal to the length of the shortest path among all paths from i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any numbered not higher k.
- The initial matrix $D^{(0)}$ does not allow any intermediate vertices in its path, which is nothing but the weight matrix of the graph.
- The final matrix $D(n)$ contains the length of the shortest path among all paths that use all n vertices as intermediate.
- All the elements of each matrix D^k is computed from its predecessor D^{k-1} in the series.

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

- $D_{ij}^{(k)}$ is equal to the length of the shortest path among all the paths from i^{th} vertex to j^{th} vertex with their intermediate vertices not higher than k.
- The path is given by

V_i, a list of intermediate vertices each,
V_j numbered no higher than k, V_j.

- All paths are partitioned into 2 disjoint subsets.
  1. First subset contains path that do not use k^{th} vertex as intermediate.
  2. Second subset contains path that do not use k^{th} vertex as intermediate
- The paths of the first subset have their intermediate vertices numbered not higher than $(k-1)$, then the shortest length is $D_{ij}^{(K-1)}$.

- The path of the second subset is given by

V_i, vertices numbered $\leq k-1$, V_k
Vj vertices numbered $\leq k-1$, V_j

- That is, each path in the second subset that is made up of a path from V_i to V_k with intermediate vertex numbered not higher than $k-1$ and a path from V_k to V_j with each intermediate vertex numbered not higher than $k-1$.

$\therefore$ Path from V_i to V_k with intermediate vertex not more than $k-1$

$$= \boxed{D_{(i,K)}^{K-1}}$$

- Path from V_K to V_j with intermediate vertex not more than $k-1$

$$= \boxed{D_{(K,j)}^{K-1}}$$

Length of shortest path = $\boxed{D_{(i,K)}^{K-1} + D_{(K,j)}^{K-1}}$

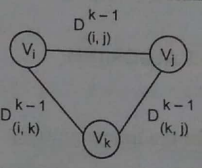


Fig. 3.3. Underlying idea of all pair shortest path problem

Length of the shortest path

- By taking the length of the shortest paths in both subsets, the length of the shortest path is given as

$$D_{(i,j)}^{(K)} = \min \left\{ D_{(i,j)}^{(k-1)}, D_{(i,K)}^{(K-1)} + D_{(K-j)}^{(K-1)} \right\} k \geq 1$$

$$D^{(0)}(i,j) = W(i,j)$$

$W(i,j)$ = Weight matrix

- The element in i^{th} row and j^{th} col of the current distance matrix $D^{(k-1)}$ is replaced by the sum of elements in the same row i and k^{th} column and in

the same column j and k^{th} row if the latter sum is smaller than its current value.

3.3.2. ALGORITHM FOR FLOYD'S PROBLEM

Algorithm Floyd's (W[1...n, 1 ...n])

// Implements Floyd's algorithm for all the all pairs shortest path, problem.

//The weight matrix W of a graph – Input.

// Output – The distance matrix of the shortest paths lengths.

$D = W$

for $k = 1$ to n do

for $i = 1$ to n do

for $j = 1$ to n do

$D[i, j] = \min \{ D[i,j], D[i, k] + D[k,j]\}$

return D.

3.3.3. EXAMPLE FOR FLOYD'S ALGORITHM

Example 1

Find all pair shortest path for the following graph.

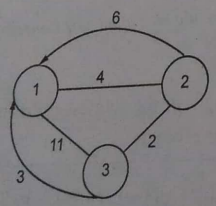


Fig. 3.4 Example of Floyd's algorithm.

1. Find weighted matrix of a given graph is

$$W = D^{(0)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \alpha & 0 \end{bmatrix}$$

Length of the shortest path with no intermediate vertices is obtained.

2. The boxed row and column in used to generate the matrix $D^{(1)}$

$$\begin{bmatrix} \boxed{0} & \boxed{4} & \boxed{11} \\ 6 & 0 & 2 \\ 3 & \alpha & 0 \end{bmatrix}$$

3. Lengths of the shortest path with intermediated vertices numbered not higher than 1.

One new path is obtained.

(1) $3 \rightarrow 2 = 3 \rightarrow 1 \rightarrow 2 = 3 + 4 = \boxed{7}$

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

4. The boxed row and column in $D^{(1)}$ is used to generate the elements of $D^{(2)}$

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

5. Length of the shortest path with intermediate vertices numbered not higher than 2.

The new path $1 \rightarrow 2 \rightarrow 3 = 4 + 2 = 6$ is obtained.

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

6. The boxed row and column in $D^{(2)}$ is used to generate the elements of $D^{(3)}$

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

7. Lengths of the shortest path with intermediate numbered not higher than 3, the new path $2 \rightarrow 3 \rightarrow 1 = 2 + 3 = \boxed{5}$ is obtained.

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

Example 2

Find all pair shortest path for the following graph

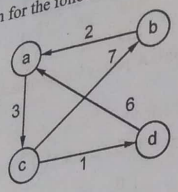


Fig. 3.5.

1. Find the weighted matrix of a given graph is

$$W = D^{(0)} = \begin{bmatrix} 0 & \alpha & 3 & \alpha \\ 2 & 0 & \alpha & \alpha \\ \alpha & 7 & 0 & 1 \\ 6 & \alpha & \alpha & 0 \end{bmatrix}$$

Length of the shortest path with no intermediate vertices is obtained.

2. The boxed row and column is used to generate the matrix $D^{(1)}$

$$D^{(0)} = \begin{bmatrix} \boxed{0} & \boxed{\alpha} & \boxed{3} & \boxed{\alpha} \\ 2 & 0 & \alpha & \alpha \\ \alpha & 7 & 0 & 1 \\ 6 & \alpha & \alpha & 0 \end{bmatrix}$$

3. Length of the shortest paths with intermediate vertices numbered not higher than 1, just *a* is found.

2 new shortest paths are obtained.

They are

1. $b \to a \to c = 2 + 3 = 5$
2. $d \to a \to c = 6 + 3 = 9$

$$D^{(1)} = \begin{bmatrix} 0 & \alpha & 3 & \alpha \\ 2 & 0 & 5 & \alpha \\ \alpha & 7 & 0 & 1 \\ 6 & \alpha & \alpha & 0 \end{bmatrix}$$

4. The boxed row and column in $D^{(1)}$ is used to generate the elements o $D^{(2)}$.

$$D^{(1)} = \begin{bmatrix} 0 & \boxed{\alpha} & 3 & \alpha \\ \boxed{2} & \boxed{0} & \boxed{5} & \boxed{\alpha} \\ \alpha & 7 & 0 & 1 \\ 6 & \boxed{\alpha} & \alpha & 0 \end{bmatrix}$$

5. Length of the shortest paths with intermediate vertices number not higher than 2 ie *a* and *b* is found.

One new shortest path $c \to b \to a \to 7 + 2 = \boxed{9}$

$$D^{(2)} = \begin{bmatrix} 0 & \alpha & 3 & \alpha \\ 2 & 0 & 5 & \alpha \\ 9 & 7 & 0 & 1 \\ 6 & \alpha & 9 & 0 \end{bmatrix}$$

6. The boxed row and column in $D^{(2)}$ is used to generate the element of $D^{(3)}$

$$D^{(2)} = \begin{bmatrix} 0 & \alpha & \boxed{3} & \alpha \\ 2 & 0 & \boxed{5} & \alpha \\ \boxed{9} & \boxed{7} & \boxed{0} & \boxed{1} \\ 6 & \alpha & \boxed{9} & 0 \end{bmatrix}$$

7. The length of the shortest path with intermediate vertices not higher than 3, ie *a*, *b* and *c*.

4 new paths are

1. $a \to c \to b = 3 + 7 = 10$
2. $a \to c \to d = 3 + 1 = 4$
3. $b \to a \to c \to d = 2 + 3 + 1 = 6$
4. $d \to a \to c \to b = 6 + 3 + 7 = 16$

$$D^{(3)} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

8. The boxed row and column in $D^{(3)}$ is used to generate the elements of matrix $D^{(4)}$

$$D^{(3)} = \begin{bmatrix} 0 & 10 & 3 & \boxed{4} \\ 2 & 0 & 5 & \boxed{6} \\ 9 & 7 & 0 & \boxed{1} \\ \boxed{6} & \boxed{16} & \boxed{9} & \boxed{0} \end{bmatrix}$$

9. The lengths of the shortest path with intermediate vertices numbered not greater than 4 ie (a, b, c and d) is found.

One new shortest path $c \to d \to a = 1 + 6 = 7$

$$D^{(4)} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

Example 3

Solve the all pairs shortest path problem for the digraph with the given weight matrix.

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 2 & \alpha & 1 & 8 \\ 6 & 0 & 3 & 2 & \alpha \\ \alpha & \alpha & 0 & 4 & \alpha \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & \alpha & \alpha & \alpha & 0 \end{bmatrix} \end{array}$$

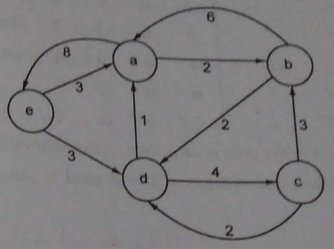


Fig. 3.6.

1. Find the weighted matrix of a given graph.

$$W = D^{(0)} = \begin{bmatrix} 0 & 2 & \alpha & 1 & 8 \\ 6 & 0 & 3 & 2 & \alpha \\ \alpha & \alpha & 0 & 4 & \alpha \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & \alpha & \alpha & \alpha & 0 \end{bmatrix}$$

Length of the shortest path with no intermediate vertices is obtained.

2. The boxed row and column is used to generate the matrix $D^{(1)}$

$$D^{(0)} = \begin{bmatrix} \boxed{0} & \boxed{2} & \boxed{\alpha} & \boxed{1} & \boxed{8} \\ \boxed{6} & 0 & 3 & 2 & \alpha \\ \boxed{\alpha} & \alpha & 0 & 4 & \alpha \\ \boxed{\alpha} & \alpha & 2 & 0 & 3 \\ \boxed{3} & \alpha & \alpha & \alpha & 0 \end{bmatrix}$$

3. Lengths of the shortest paths with intermediate vertices numbered not higher than 1, just a is found.

4 new paths were found.

$b \to a \to d = 6 + 1 = 7$

$b \to a \to e = 6 + 8 = 14$

$e \to a \to b = 3 + 2 = 5$

$e \to a \to d = 3 + 1 = 4$

$$D^{(1)} = \begin{bmatrix} 0 & 2 & \alpha & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \alpha & \alpha & 0 & 4 & \alpha \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & 5 & \alpha & 4 & 0 \end{bmatrix}$$

$b \to d$ direct path is lower than $b \to a \to d$ so the new value is not updated.

4. The boxed row and column in $D^{(1)}$ is used to generate the elements of $D^{(2)}$

$$D^{(1)} = \begin{bmatrix} 0 & \boxed{2} & \alpha & 1 & 8 \\ \boxed{6} & \boxed{0} & 3 & 2 & \alpha \\ \alpha & \alpha & 0 & 4 & \alpha \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & \alpha & \alpha & \alpha & 0 \end{bmatrix}$$

5. The lengths of the shortest path with intermediate vertices numbered not greater than 2 ie a and b.

2 new paths are found.

$a \to b \to c = 2 + 3 = 5$

$e \to a \to b \to c = 3 + 2 + 3 = 8$

$$D^{(2)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \alpha & \alpha & 0 & 4 & \alpha \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

6. The boxed row and column in $D^{(2)}$ is used to generate the elements of matrix $D^{(3)}$

$$D^{(2)} = \begin{bmatrix} 0 & 2 & \boxed{5} & 1 & 8 \\ 6 & 0 & \boxed{3} & 2 & 14 \\ \boxed{\alpha} & \boxed{\alpha} & \boxed{0} & \boxed{4} & \boxed{\alpha} \\ \alpha & \alpha & \boxed{2} & 0 & 3 \\ 3 & 5 & \boxed{8} & 4 & 0 \end{bmatrix}$$

The lengths of the shortest path with intermediate vertices numbered not greater than 3 ie a, b and c.

No new path is found.

$$D^{(3)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \alpha & \alpha & 0 & 4 & \alpha \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

7. The boxed row and column in $D^{(3)}$ is used to generate the elements of $D^{(4)}$.

$$D^{(3)} = \begin{bmatrix} 0 & 2 & 5 & \boxed{1} & 8 \\ 6 & 0 & 3 & \boxed{2} & 14 \\ \alpha & \alpha & 0 & \boxed{4} & \alpha \\ \boxed{\alpha} & \boxed{\alpha} & \boxed{2} & \boxed{0} & \boxed{3} \\ 3 & 5 & 8 & \boxed{4} & 0 \end{bmatrix}$$

8. The length of the shortest path with intermediate vertices numbered not greater than 4 ie (a, b, c and d) is found.

4 new paths are found.

$a \to b \to e = 1 + 3 = \boxed{4}$ $\quad a \to d \to c = 1 + 2 = \boxed{3}$

$b \to d \to e = 2 + 3 = \boxed{5}$

$c \to d \to e = 4 + 3 = \boxed{7}$

$$D^{(4)} = \begin{bmatrix} 0 & 2 & 3 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ \alpha & \alpha & 0 & 4 & 7 \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

9. The boxed row and column in $D^{(4)}$ is used to generate the elements of $D^{(5)}$.

$$D^{(4)} = \begin{bmatrix} 0 & 2 & 3 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ \alpha & \alpha & 0 & 4 & 7 \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

10. The length of the shortest path with intermediate vertices numbered not greater than 5, ie (a, b, c, d and e) is found.

5 new paths are generated.

$c \to d \to e \to a = 4 + 3 + 3 =$ **10**

$c \to d \to e \to a \to b = 4 + 3 + 3 + 2 =$ **12**

$d \to e \to a = 3 + 3 =$ **6**

$d \to e \to a \to b = 3 + 3 + 2 =$ **8**

$e \to a \to d \to c \to 3 + 1 + 2 =$ **6**

$$D^{(5)} = \begin{bmatrix} 0 & 2 & 3 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ 10 & 12 & 0 & 4 & 7 \\ 6 & 8 & 2 & 0 & 3 \\ 3 & 5 & 6 & 4 & 0 \end{bmatrix}$$

3.3.4. COMPLEXITY OF FLOYD'S ALGORITHM

Complexity of an algorithm depends on space and time.

Space Complexity

Storage space for weight array = n^2 locations.

Storage space for control variable = 4 locations

i, j, k, n

$\therefore$ Total Space = $n^2 + 4$ locations

Time complexity

For finding all pair shortest path algorithm uses 3 loops.

$\therefore$ Time Complexity $O(n^3)$

3.4. MULTI STAGE GRAPH

Concept

The multistage graph is to find a minimum cost path from source 's' to sink 't' i.e destination.

Problem Description

- A multistage graph G = (V, E) is a directed graph in which the vertices are partitioned into k >= 2 disjoint sets Vi $\mathbf{1 <- i <- k}$
- If (u, v) is an edge in E, then U $\in V_i$ and V $\in V_{i+1}$ for some i, $\mathbf{1 \le i \le k}$
- The sets V_1 and V_K are such that $|V_1| = |V_K| = 1$
- The vertex 's' is the source and 't' is the sink.
- Let c (i, j) be the cost edge (i, j)
- The cost of a path from 's' to 't' is the sum of the cost of the edges on the path.
- Each set V_i defines a stage in the graph
- Every path from S to t starts 1, goes to stage 2, then to stage 3 then the stage 4 and so on and eventually terminates at stage k.

Procedure for Multistage Problems

- Find path from s to t, stage by stage.
- Every s to t path is the result of a sequence of k – 2 decisions.
- The i^{th} decision involves determining which vertex in V_{i+1}, $1 \le i \le k-2$ is to be on the path.
- P(i, j) be a minimum cost path from vertex j in V_i to vertex t.
- Cost (i, j) be the cost of the path.
- Find cost of path using the formula.

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

- Cost $(k-1, j) =$ if $c(j, t) \in E$
- Cost $(k-1, j) = \alpha$ if $(j, t) \notin E$
- Shortest distance between source 's' and sink 't' using following formula.

Cost $(1, s)$ by first computing

Cost $(k-2, j)$ for all $j \in V_{K-2}$
Cost $(k-3, j)$ for all $j \in V_{K-3}$
Cost $(1, s)$

3.4.1. EXAMPLE FOR MULTISTAGE GRAPH

Find the shortest distance between source 's' and sink 't'.

Using 5 stage graph.

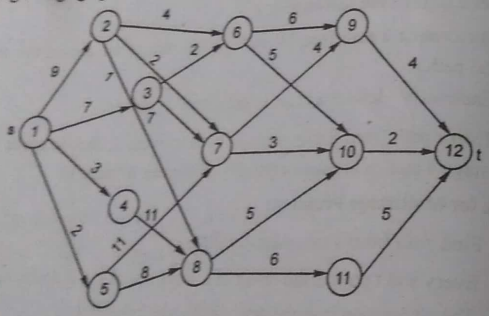


Fig. 3.7.

1. Compute cost $(k-2, j)$ for all $j \in V_{K-2}$

$k = 5$ because it is 5 stage graph

III stage contains 6, 7 and 8 i.e., 3 nodes.

$$\text{Cost}(i, j) = \min \{c(j, l), + \text{cost}(i+1, l)$$

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

$$= \boxed{7}$$

$$\text{Cost}(3, 7) = \min \{c(j+l) + \text{cost}(i+1, l)\}$$
$$= \min \{4 + \text{cost}(4, 9), 3 + \text{cost}(4, 10)\}$$
$$= \min \{4+4, 3+2\}$$
$$= \min \{8, 5\}$$
$$= \boxed{5}$$

$$\text{Cost}(3, 8) = \min \{c(j+l) + \text{cost}(i+1, l)\}$$
$$= \min \{5 + \text{cost}(4, 10), 6 + \text{cost}(4, 11)\}$$
$$= \min \{5+2, 6+5\}$$
$$= \min \{7, 11\}$$
$$= \boxed{7}$$

2. Compute cost $(k-3, j)$ for all $j \in V_{K-3}$

II stage contains 2, 3, 4 and 5nodes.

$$\text{Cost}(2, 2) = \min \{4 + \text{cost}(3, 6), 2 + \text{cost}(3, 7)\ 1 + \text{cost}(3, 8)\}$$
$$= \min \{(4+7), (2+5), (1+7)\}$$
$$= \min \{11, 7, 8\}$$
$$= \boxed{7}$$

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

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

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

3. Compute cost (1, s)

I stage contains 1 node

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

= min {9 + 7, 7 + 9, 3 + 18, 2 + 15}

= min {16, 16, 21,17}

= 16

∴ A minimum cost from 's' to t path = 16

The path from 's' to 't' in indicated by broken edge.

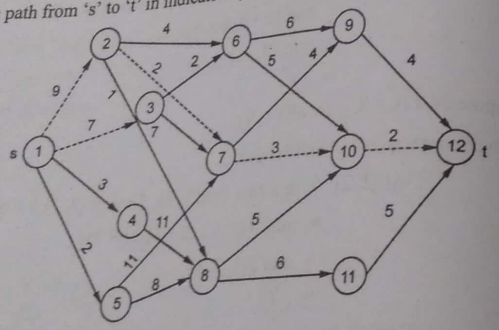


Fig. 3.8.

Example 2

Find the shortest path form source's' and sink 't' using multistage graph.

Solution:

Find the shortest path between source 's' and sink 't' using 5 stage graph.

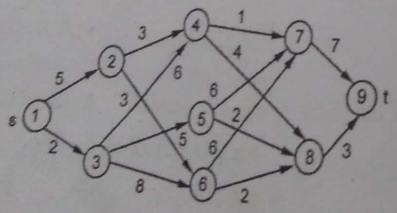


Fig. 3.9.

1. First compute cost $(k-2, j)$ for all $j \in V_{K-2}$, $k = 5$ because it is 5 stage graph.

III stage contains 3 nodes such as 4, 5, and 6.

Cost (i, j) = min {c (i, l) + cost (i + 1, l)}

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

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

= min {8, 7}

= 7

Cost (3, 5) = min {c (j, l) + cost (i + 1, l)}

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

= min {(6 + 7), (2 + 3)}

= min {13, 5}

= 5

Cost (3, 6) = min {c (j, l) + cost (i + 1, l)}

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

= min {(6 + 7), (2 + 3)}

= min {13, 5}

= 5

2. Compute cost $(k-3, j)$ for all $j \in V_{K-3}$,

where k = 5

II stage contains 2 nodes such as 2 and 3.

Cost (2, 2) = min {c (j, l) + cost (i + 1, l)}

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

= min {(3 + 7), (3 + 5)}

= min {10, 8}

= 8

Cost (2, 3) = min {c (j, l) + cost (i + 1, l)}

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

= min {(6 + 7), (5 + 5), (8 + 5)}

$= \min \{13, 10, 13\}$

$= \boxed{10}$

3. Compute cost (1, s)

I Stage contains only one node 1.

$$\text{Cost}(1, 1) = \min \{5 + \text{cost}(2, 2), 2 + \text{cost}(2, 3)\}$$
$$= \min \{(5 + 8), (2 + 10)\}$$
$$= \min \{13, 12\}$$
$$= \boxed{12}$$

$\therefore$ A minimum cost from 's' for to 't' is $\boxed{12}$

The path from s to t is indicated by broken edges.

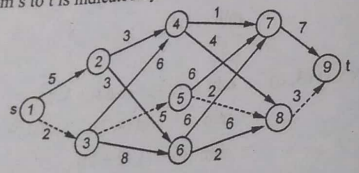


Fig. 3.10. Shortest path from 's' to t using 5 stage graph

3.4.2. ALGORITHM FOR MULTISTAGE GRAPH USING FORWARD APPROACH

```
void FGraph (Graph G, int k, int n, int p[ ] )
{ // the input is a k stage graph G = (V, E)
    With n vertices
  // Indexed in order of stages
  // E is a set of edges and c [i, j] is the cost of (i, j)
  // P [i ; k] in a minimum cost path vertex
{
  Cost [n] = 0, 0; // cost of vertex n is zero
 for (j= n – 1; j > – 1 ; j – –)
{ // compute cost [j]
```



```
    // let r be a vertex such that (j, r) is an edge of G and
    // c [j, r] + cost [r] is minimum
      Cost [j] = c[j, r] + bcost [r];
      d[j] = r;
  }
 // Find a minimum cost path
  P [1] = 1;
  P [k] = n;
  for (j = 2; j <= k – 1; j + 1)
    P [j] = d [P [j – 1]];
}}
```

Let the minimum cost path be s = 1, V_2, V_3, V_{K-1}, t,

For the figure ??

$$V_2 = d(1, 1) = 2$$
$$V_3 = d(2, d(1, 1))$$
$$V_3 = d(2, 2) = 7$$
$$= d(3, d(2, d(1, 1)))$$
$$= d(3, 7)$$
$$= \boxed{10}$$

- n vertices in V are indexed 1 through n.
- Indices are assigned in order of stages.
- 's' is assigned to index 1.
- Vertices in V_2 are assigned indices.
- Next vertices in V_3 to be assigned.
- This process is repeated up to sink vertex 't'.
- Vertex 't' has index 'n'.
- Cost and d can be computed in the order n – 1, n – 2, … 1.

3.4.3. MULTISTAGE GRAPH USING BACKWARD APPROACH

- Multistage can be solved using backward graph approach.
- Let bp (i, j) be the cost path from vertex s to a j in v_1
- bcost (i, j) be the cost of bp (i, j).
- Shortest path from source 's' to sink 't' using backward approach.

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

- bcost (2, j) = c (1, j) if (1, j) ∈ E

  bcost (2, j) = α if (1, j) ∉ E

Find shortest path from source 's' to sink 't' for the following graph backward approach.

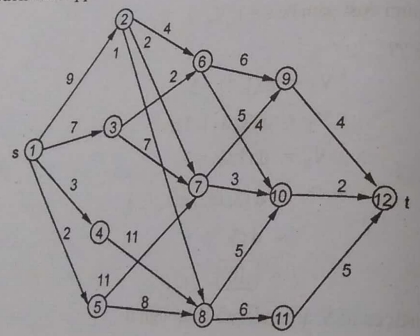


Fig. 3.11.

1. Compute bcost for i = 2

bcost (2, 2) = min {c (1, 2)}
= 9

bcost (2, 3) = c (1, 3)
= 7

bcost (2, 4) = c (1, 4)

= 3

bcost (2, 5) = c (1, 5)
= 2

2. Compute bcost for i = 3

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

bcost (3, 7) = min {bcost (2, 2) + c (2, 7), bcost (2,3) + c(3, 7), bcost (2,5) + c (5, 7)}}
= min {(9 + 2), (7 + 7), (2 + 11)}
= min {11,14,13}
= 11

bcost (3, 8) = min {bcost (2, 2) + c (2, 8), bcost (2, 4) + c (4, 8), bcost (2, 5) + c (5, 8)}
= min {(9 + 1), (3 + 11), (2 + 8)}
= min {10, 14, 10}
= 10

3. Compute bcost for i = 4

bcost (4, 9) = min {bcost (3, 6) + c (6, 9), bcost (3, 7) + c (7, 9)}
= min {(9 + 6), (11 + 5)}
= min {15, 16}
= 15

bcost (4, 10) = min {bcost (3, 6) + c (6, 10), bcost (3, 7) + c (7, 10)}
= bcost (3, 8) + c (8, 10)}
= min {(8 + 4), (7 + 2), (8 + 2)}
= min {14, 14, 15}
= 14

$$bcost(4, 11) = \min\{bcost(3, 8) + c(8, 11)\}$$
$$= \min\{(10 + 6)\}$$
$$= \min\{16\}$$
$$= \boxed{16}$$

4. Compute bcost for $i = 5$

$$bcost(5, 12) = \min\{bcost(4, 9) + c(9, 12), bcost(4, 10) + c(10, 12), bcost(4, 11) + c(11, 12)\}$$
$$= \min\{(15 + 4), (14 + 2)(16 + 5)\}$$
$$= \min\{19, 16, 21\}$$
$$= \boxed{16}$$

$\therefore$ Cost of path from source 's' to sink 't' = $\boxed{16}$

The broken edge indicate the shortest path from source 's' to sink 't'.

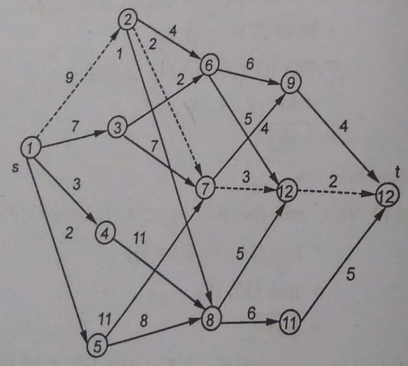


Fig. 3.12. Shortest path from source 's' to sink 't' using backward approach

Example 2

Find shortest path form source's' to sink '*t*' for the following graph using backward approach.

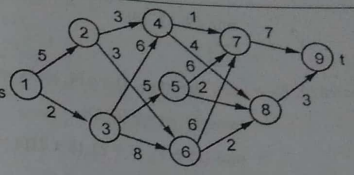


Fig. 3.13.

1. Compute bcost for i = 2

$$bcost(2, 2) = c(1, 2)$$
$$= \boxed{5}$$
$$bcost(2, 3) = c(1, 3)$$
$$= \boxed{2}$$

2. Compute bcost for i = 3

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

3. Compute bcost for i = 4

$$bcost(4, 7) = \min\{bcost(3, 4) + c(4, 7), bcost(3, 5) + c(5, 7)\}$$
$$= bcost(3, 6) + c(6, 7)\}$$
$$= \min\{(8 + 1), (7 + 6), (8 + 6)\}$$

$= \min \{9, 13, 14\}$

$= \boxed{9}$

$\text{bcost}(4, 8) = \min \{\text{bcost}(3, 4) + c(4, 8), \text{bcost}(3, 5) + c(5, 8)\}$

$= \text{bcost}(3, 6) + c(6, 8)\}$

$= \min \{(8 + 4), (7 + 2), (8 + 2)\}$

$= \min \{12, 9, 10\}$

$= \boxed{9}$

4. Compute bcost for $i = 5$

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

$= \min \{(9 + 7), (9 + 3)\}$

$= \min \{16, 12\}$

$= \boxed{12}$

$\therefore$ Shortest path from source '*s*' to sink '*t*' = $\boxed{12}$

Shortest path from source '*s*' to sink '*t*' indicated by broken ??

Edges (8, 9), (5, 8), (3, 5), (1, 3) shown with broken lines.

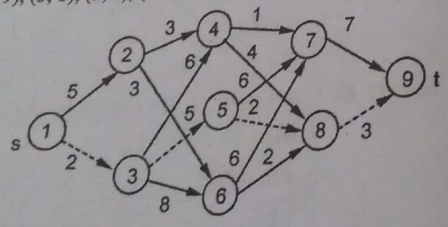


Fig. 3.14. Shortest path from source 'S' to sink 't' using backward approach

3.4.4. ALGORITHM FOR MULTISTAGE GRAPH USING BACKWARD APPROACH

```
void BGraph (Graph G, int K, int n, int p[ ])
{ bcost [1] = 0. 0;
```



```
    // cost of vertex 1 is zero.
for (j=2; j<n; j ++)
{   // compute bcost [j]
    // Let r be such that (r, j) is an edge
    // of G and bcost [r] + c [r, j] is minimum
bcost [j] = bcost [r] + c[r, j];
d[j] = r;
}
// Find a minimum cost path
  P [1] = 1;
  P [k] = n;
  For (j = k – 1; j > 2; j – –)
    P [j] = d [P [j + 1]];
}
```

3.4.5. COMPLEXITY OF MULTISTAGE GRAPH FOR BOTH FORWARD AND BACKWARD APPROACH

Time Complexity

Finding the minimum cost for each and every stage – 0 (| V | + | E |)

Shortest path from source s to sink t $\rightarrow$ 0 (k)

Space Complexity

Storage space for cost array, cost [ ] - n location

Storage space for minimum cost path P [ ] - n location

Storage space for decision array d [ ] - n location

Storage space for stage k - 1

Storage space for variable 'n' - 1

Control variable j - 1

$\therefore$ Total storage space - $3n + 3 = \boxed{3(n + 1)}$

3.4.6. APPLICATION OF MULTISTAGE PROBLEM

Resource Allocation Problem

- n units of resource are to be allocated to 'r' projects.
- If j $0 \le j \le n$ units of the resource are allocated to project i, the resulting ?? profit is N(i, j)
- The problem is to allocate the resource to the r project in such a way maximize total net profit.
- It can be formulated as an r + 1 stage.

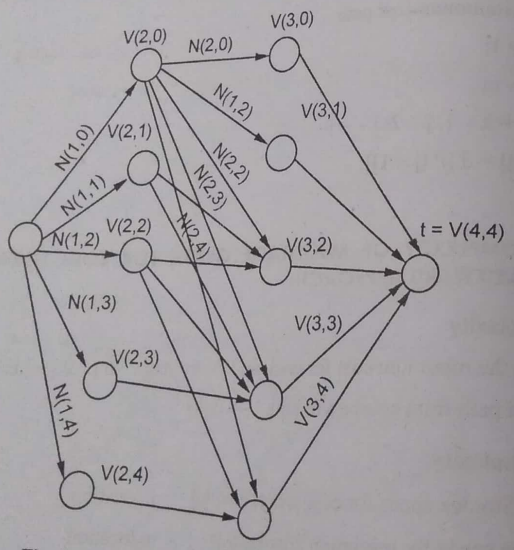


Fig. 3.15. Four stage graph for 3 project problem

- Stage i, $1 \le i \le r$ represents project i. There are n + 1 vertices associated with stage i, $2 \le i \le r$
- Stage 1 and stage r + 1 have on vertex

  Stage 1 – r (1, 0) = s

  Stage r + 1 – V (r + 1, n) = t

- The edges in G are of the from (V (i, j), V (i + 1, l))

  for all j <= l and $1 \le i < r$
- The edges in G (V(i j), V (i + 1, l)) l is assigned a weight (or) cost of N (i, l – j) and corresponds to allocating l – j units of resource to project i,

  $1 \le i \le r$

  Examples – four stage graph corresponding to a 3 project problem.

3.5. OPTIMAL BINARY SEARCH TREE

Basic Concept

Dynamic programming can be used for constructing an optimal binary search tree for a given set of keys and known probabilities of searching for them. If probabilities of searching for elements of set are known, it is natural have an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.

Example

Consider 4 keys A, B, C and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3 respectively.

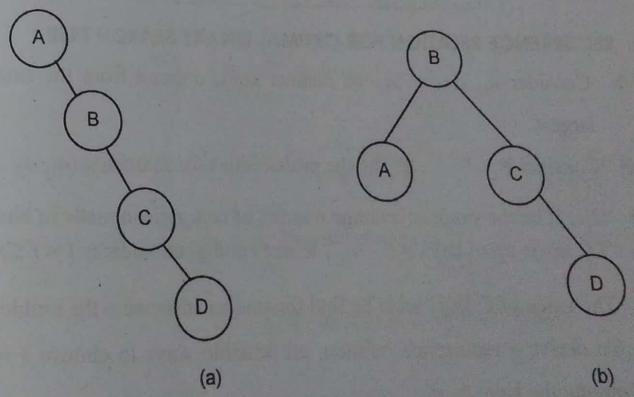


Fig. 3.16. a, b Possible binary search trees for the keys A, B, C, &D

Average no of comparisons in Fig.3.16(a)

Successful search for 'A' = 0.1 * 1 = 0.1

Successful search for 'B' = 0.2 * 2 = 0.4

Successful search for 'C' = 0.4 * 3 = 1.2

Successful search for 'D' = 0.3 * 4 = 1.2

∴ Total no of comparison = 2.9

Average no of comparison in Fig.3.16 (b)

Successful search for 'A' = 0.1 * 2 = 0.2

Successful search for 'B' = 0.2 * 1 = 0.2

Successful search for 'C' = 0.4 * 2 = 0.8

Successful search for 'D' = 0.3 * 3 = 0.9

∴ Total no of comparison = 2.1

3.5.1. FINDING POSSIBLE BINARY SEARCH TREE WITH *n* KEYS IS EQUAL TO THE n^{TH} CATALAN NUMBER.

$$C(n) = \binom{2n}{n} \frac{1}{n+1} \quad \text{for } n > 0,\ C(0) = 1$$

3.5.2. RECURRENCE RELATION FOR OPTIMAL BINARY SEARCH TREE

- Consider $a_1, a_2 \ldots a_n$ be distinct keys, ordered from the smallest to largest.
- Consider $P_1, P_2 \ldots P_n$ be the probabilities for searching $(a_1, a_2 \ldots a_n)$.
- $C[i, j]$ be the smallest average number of comparison made in binary tree T_i^j made up of keys $a_i, \ldots a_j$, where i and j are indices $1 < i \le j \le n$.
- The value of $C[i, j]$ must be find for smaller instance o the problem.
- To derive a recurrence relation, all possible ways to choose a root a_K among the keys $a_1, a_2, \ldots a_j$.

- Consider the following binary tree, the root contains key a_K ,the left sub tree T_i^{K-1} contains keys $a_i \ldots a_{K-1}$ optimally arranged and the right sub tree T_j^{K+1} contains keys $a_{K+1} \ldots a_j$ also optimally arranged.

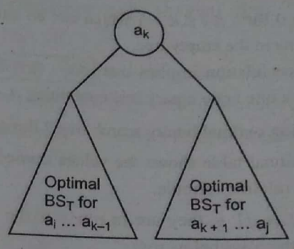


Fig. 3.17. Binary search tree with root a_k and 2 optimal binary search trees T_i^{K-1} and T_{K+1}^j

The recurrence relation is obtained when tree levels are started with 1.

$$C[i,j] = \min_{i \le k \le j} \left\{ P_K^1 + \sum_{S=i}^{K-1} \text{Ps. (level of as in } T_i^{K-1} + 1) + \sum_{S=k+1}^{j} \text{Ps. (level of as in } T_{K+1}^j + 1) \right\}$$

$$= \min_{i \le k \le j} \left\{ P_k + \sum_{S=i}^{K-1} \text{Ps level of as in } T_i^{K-1} + \sum_{S=i}^{K-1} \text{Ps.} \sum_{S=k+1}^{j} \text{level of as in } T_{k+1}^j + \sum_{S=k+1}^{j} \text{Ps} \right\}$$

$$= \min_{i \le k \le j} \left\{ \sum_{S=i}^{K-1} \text{Ps. level of as in } T_i^{K-1} \sum_{S=k+1}^{j} \text{Ps. level of as in } T_{k+1}^j + \sum_{S=i}^{j} \text{Ps.} \right\}$$

$$= \min_{i \le k \le j} \{C[i, k-1] + C[k+1, j]\} + \sum_{S=i}^{j} \text{Ps}$$

$$\therefore C[i,j] = \min_{i \le k \le j} \{ C[i, k-1] + C[k+1, j] \} + \sum_{s=i}^{j} Ps \text{ for } 1 \le i \le j \le n$$

3.5.3. PROCEDURE FOR OPTIMAL BINARY SEARCH TREE

- $C[i, i-1] = 0$ for $1 \le i \le n+1$ which can be interpreted as the number of comparisons in the empty tree.
- The recurrence relation implies that $C[i, i] = P_i$ for $1 \le i \le n$ as it should be for a one node binary tree containing a_i.
- For constructing optimal binary search tree 2 dimensional able is used.
- The 2 dimensional table shows the values needed for computing $C[i, j]$ by recurrence relation formula.
- Computation $C[i, j]$ = they are in row i & the columns to the left of column j and the rows below row i.

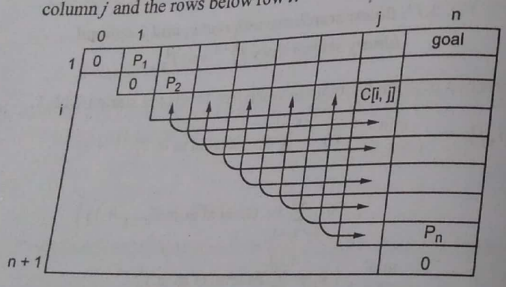


Fig. 3.18. Table of the dynamic programming algorithm for constructing an optimal binary search tree

- The arrow points to the pair of entries whose sums are computed in order to find the smaller one to be recorded as the value of $C[i, j]$.
- The table is filled along its diagonal, starting with all zeros on the main diagonal and given probabilities P_i, $1 \le i \le n$ right above it and moving toward the upper right corner.
- We can easily compute the average no of comparisons $C[1, n]$ from the table.

- We need to maintain another 2 dimensional table to record the value of K (minimum) its table starting with entries $R[i, i] = i$ for $1 \le i \le n$. Its entries indicate indices of the roots of the optimal sub trees.

3.5.4. ALGORITHM FOR OPTIMAL BINARY SEARCH TREE

Algorithm optimal BST (P[1…n]).

// Find an optimal binary search tree for dynamic programming.

// Input: An array P [1…n] of search probabilities for a sorted list of n keys.

// Output: Average number of comparisons in successful searches in the optimal BST & roots in optimal BST.

For $i = 1$ to n. do

  $C[i, i-1] = 0$

  $C[i, i] = P[i]$

  $R[i, i] = i$

  $C[n+1, n] = 0$ // Initial values

for $d = 1$ to $n - 1$ do // diagonal count

for $i = 1$ to $n - d$ do

  $j = i + d$

  for $k = i$ to j do

if $(C[i, k-1] + C[k+1, j]) <$ minval

  minval = $C[i, k-1] + C[k+1, j]$

  k min = k

$R[i, j] = k$

sum = $P[i]$

for S = $i + 1$ to j do

  sum = sum + $P[s]$

$C[i, j]$ = minval + sum

return $C[1, n]$, R // Average no of comparisons and root of the tree.

3.5.5. EXAMPLE FOR OPTIMAL BINARY SEARCH TREE

Find the optimal binary search tree for the following key values A, B, C & D.

Keys	A	B	C	D
Probability	0.1	0.2	0.4	0.3

❖ Fill the main diagonal of table with zero using the formula $C[i, i-1] = 0$ for $1 \le i \le n+1$.

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

$C[1, 0] = 0$

$C[2, 1] = 0$

$C[3, 2] = 0$

$C[4, 3] = 0$

$C[5, 4] = 0$

❖ Fill the table with $C[i,i]$ P_i for $1 \le i \le n$.

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

$C[1, 1] = 0.1$

$C[2, 2] = 0.2$

$C[3, 3] = 0.4$

$C[4, 4] = 0.3$

❖ The root table filled with initial value $R[i, i] = i$ for $1 \le i \le n$.

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

❖ Compute all other entries.

1. Compute C [1, 2] use the recurrence relation formula.

$$C[i,j] = \min_{i \le k \le j} \{ C[i, k-1] + C[k+1, j] \} + \sum_{S=i}^{j} Ps \text{ for } 1 \le i \le j \le n$$

When K = 1

$$C[1, 2] = C[1, 1] + C[2, 2] + \sum_{i=1}^{2} Ps.$$

$$= 0 + 0.2 + [0.1 + 0.2]$$

$$= 0 + 0.2 + 0.3 = \boxed{0.5}$$

When K = 2

$$C[1, 2] = C[1, 2] + C[3, 2] + \sum_{i=1}^{2} Ps$$

$$= 0.1 + 0 + [0.1 + 0.2]$$

$$= \boxed{0.4}$$

$$C[1, 2] = \min\{0.5, 0.4\} = \boxed{0.4}$$

❖ $\therefore C[1, 2] = 0.4$ & $K = 2$

enter $C[1, 2] = 2$ in root table.

❖ Next compute C [2, 3]

When K = 2

$$C[2,3] = C[2,1] + C[3,3] + \sum_{S=2}^{3} Ps$$

$$= 0 + 0.4 + [0.2 + 0.4]$$

$$= \boxed{1.0}$$

When K = 3

$$C[2,3] = C[2,2] + C[4,3] + \sum_{S=2}^{3} Ps$$

$$= 0.2 + 0 + [0.2 + 0.4]$$

$$= \boxed{0.8}$$

When K = 3,

The values C [2, 3] is minimum value

∴ C [2, 3] = 0.8 in main table &

C [2, 3] = 3, in root table

❖ Compute C [3, 4]

When K = 3

$$C[3,4] = C[3,2] + C[4,4] + \sum_{S=3}^{4} Ps$$

$$= 0 + 0.3 + [0.4 + 0.3]$$

$$= \boxed{1.0}$$

When K = 4

$$C[3,4] = C[3,3] + C[5,4] + \sum_{S=3}^{4} Ps$$

$$= 0.4 + 0 + [0.4 + 0.3$$

$$= \boxed{1.1}$$

∴ When K = 3, the values C [3, 4] is minimum value

so C [3, 4] = 1.0 in main table &

C [3, 4] = 3 in root table

The contents of root & main table

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

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

❖ Compute C [1, 3]

When K = 1

$$C[1,3] = C[1,0] + C[2,3] + \sum_{S=1}^{3} Ps$$

$$= 0 + 0.8 + [0.1 + 0.2 + 0.4]$$

$$= 0.8 + 0.7 = \boxed{1.5}$$

When K = 2

$$C[1,3\} = C[1,1] + C[3,3] + \sum_{S=1}^{3} Ps$$

$$= 0.1 + 0.4 + [0.1 + 0.2 + 0.4]$$

$$= \boxed{1.2}$$

When K = 3

$$C[1,3] = C[1,2] + C[4,3] + \sum_{S=1}^{3} Ps$$

$$= 0.4 + 0 + [0.1 + 0.2 + 0.4]$$

$$= \boxed{1.1}$$

When K = 3, C [1, 3] is minimum

∴ C [1, 3] = 1.1 in main table &

$C[1, 3] = 3$ in root table.

❖ Compute $C[2, 4]$

When K = 2

$$C[2, 4] = C[2, 1] + C[3, 4] + \sum_{S=2}^{4} Ps$$

$$= 0 + 1.0 + [0.2 + 0.4 + 0.3]$$

$$= \boxed{1.9}$$

When K = 3

$$C[2, 4] = C[2, 2] + C[4, 4] + \sum_{S=2}^{4} Ps$$

$$= 0.2 + 0.3 + [0.2 + 0.4 + 0.3]$$

$$= \boxed{1.4}$$

When K = 4

$$C[2, 4] = C[2, 3] + C[5, 4] + \sum_{S=2}^{4} Ps$$

$$= 0.8 + 0 + [0.2 + 0.4 + 0.3]$$

$$= \boxed{1.7}$$

When K = 3, the value of $C[1, 3]$ is minimum

$\therefore C[2, 4] = 1.4$ in main table &

$C[2, 4] = 3$ in root table

Now the contents of root and main table.

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

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

❖ Compute C [1, 4]

When K = 1

$$C[1, 4] = C[1, 0] + C[2, 4] + \sum_{S=1}^{4} Ps$$

$$= 0 + 1.4 + [0.1 + 0.2 + 0.4 + 0.3] = \boxed{2.4}$$

When K = 2

$$C[1, 4] = C[1, 1] + C[3, 4] + \sum_{S=1}^{4} Ps$$

$$= 0.1 + 1.0 + [0.1 + 0.2 + 0.4 + 0.3] = \boxed{2.1}$$

When K = 3

$$C[1, 4] = C[1, 2] + C[4, 4] + \sum_{S=1}^{4} Ps$$

$$= 0.4 + 0.3 + [0.1 + 0.2 + 0.4 + 0.3] = \boxed{1.7}$$

When K = 4

$$C[1, 4] = C[1, 3] + C[5, 4] + \sum_{S=1}^{4} Ps$$

$$= 1.1 + 0 + [0.1 + 0.2 + 0.4 + 0.3] = \boxed{2.1}$$

When K = 3, the value of C [1, 4] is minimum

So enter C [1, 4] = 1.7 in main table &

C [1, 4] = 3 in root table.

❖ The contents of root & main table

	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

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

- The average number of key comparison in the optimal tree = 1.7
- Since R [1, 4] = 3. The root of the optimal tree contains the 3rd key ie C.
- Its left subtree made of A and B and right subtree contains just key ie D.
- The root of [1, 2] = 2 ie key B, and its left child is A.

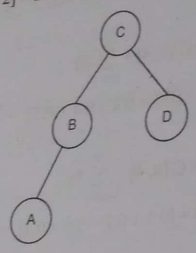


Fig. 3.19. Optimal binary search tree

3.5.6. COMPLEXITY OF OPTIMAL BINARY SEARCH TREE

The complexity of an algorithm depends on space and time.

Space Complexity

Space for cost array = n^2 location

Space for root array = n location

Space for search probabilities = n location

Space for control variables = 3 location

$(i, j\ k)$

$\therefore$ Total space = $\boxed{n^2 + 2n + 3}$ locations

Time Complexity

Entries in the root table are always non decreasing along each row and column. This limits values of R $[i, j]$ to the range R $[i, j-1]$ ...R$[i+1, j]$ and makes it possible to reduce the running time of the algorithm = 0 (n^2).

3.5.7. MERITS

- It is one of the most important data structure in computer science
- It is used to implement a dictionary, a set of elements with the operation of searching, insertion and deletion.

3.6. KNAPSACK PROBLEM

Concept

Given n items of known weighs $w_1, w_2 \ldots w_n$ and values $V_1, V_2, \ldots V_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

3.6.1. RECURRENCE RELATION FOR KNAPSACK PROBLEM

- Recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solution to its smaller sub instances.
- Consider an instance defined by the first i

  items $1 \le i \le n$

  weights $w_1, w_2 \ldots w_i$

  values $V_1, V_2 \ldots V_i$

  knapsack capacity j $1 \le j \le w$.
- Let V $[i, j]$ be the value of an optimal solution.
- The value of the most valuable subset of the first i item that fit into the knapsack of capacity j.
- We can divide all the subsets of the first i item that fit into the knapsack of capacity j into 2 categories.
  1. Do not include the i^{th} item.
  2. Include the i^{th} item.

- The subsets that include the i^{th} item, the value of an optimal solution $\boxed{V[i-1, j]}$
- The subset that include the i^{th} item $[j - w_i \geq 0]$, an optimal subset is made up of this item and an optimal subset of the first $i-1$ item that fit into the knapsack of capacity $j - w_i$.

  The value of optimal subset is

  $V_i + V[i-1, j-w_i]$
- The value of an optimal solution among all feasible subsets of the first i items is the maximum of these values.
- Suppose if the item does not fit into the knapsack, the value of an optimal subset selected form the first 'i' items is the same as the value of an optimal subset selected from the first $i-1$ item.
- The recurrence relation is

$$V[i,j] = \begin{cases} \max\{V[i-1,j], V_i + V[i-1, j-w_i]\} & \text{If } (j-w_i) \geq 0 \\ V[i-1,j] & \text{if } (j-w_i) < 0 \end{cases}$$

3.6.2. PROCEDURE FOR KNAPSACK PROBLEM

- Define the initial condition as follows

  $V[0, j] = 0$ for $j \geq 0$

  $V[i, 0] = 0$ for $i \geq 0$
- Find $V[n, w]$, the maximum value of a subset of the n given items fit into the knapsack of capacity W.
- For $i, j > 0$ to compute the entry in the i^{th} row and j^{th} column.
- The table can be filled either row by row (or) column by column.

Fig. 3.20. Table for solving the knapsack problem by dynamic programming

	0	$j - w_i$	j	w
0	0	0	0	0
.	0			
W_i V_i $i-1$	0	$V[i-1, j-W_i]$	$V[i-1, j]$	
. 0	0		$V[i, j]$	
.	.			
.	.			
.	.			
n 0				Goal

3.6.3. EXAMPLE FOR KNAPSACK PROBLEM

Let us consider the instance given by the following data.

Item	Weight	Value
1	2	$ 12
2	1	$ 10
3	3	$ 20
4	2	$ 15

Capacity W = 5

Find the optimal subset.

- Define the initial condition

  $V[0, j] = 0$

  $V[0, 0] = 0$

  $V[0, 1] = 0$

  $V[0, 2] = 0$

$V[0, 3] = 0$

$V[0, 4] = 0$

$V[0, 5] = 0$

and $V[i, 0] = 0$

$V[0, 0] = 0$

$V[1, 0] = 0$

$V[2, 0] = 0$

$V[3, 0] = 0$

$V[4, 0] = 0$

❖ The initial table contains the following data.

		Capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$W_1 = 2$ $V_1 = 12$	1	0					
$W_2 = 1$ $V_2 = 10$	2	0					
$W_3 = 3$ $V_3 = 20$	3	0					
$W_4 = 2$ $V_4 = 15$	4	0					

❖ Compute value of $V[i, j]$ row by row.

I row

$$V[1, 1] = V[i-1, j] \text{ if } j - w_i \le 0$$
$$= V[0, 1]$$
$$= \boxed{0}$$
$$V[1, 2] = \max \{V[i-1], V_i + V[i-1, j-w_i >= 0$$
$$= \max \{V[0, 2], 12 + V[0,2-2]\}$$

$$= \max \{V[0, 3], 12 + V[0, 0]\}$$
$$= \max \{0, 12 + 0\}$$
$$= \boxed{12}$$
$$V[1, 3] = \max \{V[i-1, j], V_i + V[i-1, j-w_i]\}$$
$$= \max \{V[0, 3], 12 + V[0, 1]\}$$
$$= \max \{0, 12 + 0\}$$
$$= \boxed{12}$$
$$V[1, 4] = \max \{V[0, 4], 12 + V[0, 2]\}$$
$$= \max \{0, 12 + 0\}$$
$$= \boxed{12}$$
$$V[1, 4] = \max \{V[0, 5], 12 + V[0, 3]\}$$
$$= \max \{0, 12 + 0\}$$
$$= \boxed{12}$$

II row

Value and weight of the item.

$W_2 = 1$, $V_2 = 10$

$$V[2, 1] = \max \{V[i-1, j], V_i + V[i-1, j-w_i]\}$$
$$= \max \{V[1, 1], 10 + V[1, 1-1]\}$$
$$= \max \{0, 10 + 0\}$$
$$= \boxed{10}$$
$$V[2, 2] = \max \{V[1, 2], 10 + V[1, 1]\}$$
$$= \max \{12, 10 + 0\}$$
$$= \boxed{12}$$
$$V[2, 3] = \max \{V[1, 3], 10 + V[1, 2]\}$$
$$= \max \{12, 10 + 12\}$$
$$= \boxed{22}$$
$$V[2, 4] = \max \{V[1, 4], 10 + V[1, 3]\}$$

$= \max \{12, 10 + 12\}$

$= \boxed{22}$

$V[2, 5] = \max \{V[1, 5], 10 + V[1, 4]\}$

$= \max \{12, 10 + 12\}$

$= \boxed{22}$

III row

The weight and value for 3rd item.

$W_3 = 3, V_3 = 20$

$V[3, 1] = V[i-1, j]$, if $j - w_i < 0$

$= V[2, 1]$

$= \boxed{10}$

$V[3, 2] = V[i-1, j]$, if $j - w_i < 0$

$= V[2, 2]$

$= \boxed{12}$

$V[3, 3] = \max \{V[i-1, j], V_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 3], 20 + V[2, 0]\}$

$= \max \{22, 20 + 0\}$

$= \boxed{22}$

$V[3, 4] = \max \{V[2, 4], 20 + V[2, 1]\}$

$= \max \{22, 20 + 10\}$

$= \boxed{30}$

$V[3, 5] = \max \{V[2, 5], 20 + V[2, 2]\}$

$= \max \{22, 20 + 12\}$

$= \boxed{32}$

IV row

The value and weight of 4th item.

$W_4 = 2, V_4 = 15$

$V[4, 1] = V[i-1, j]$, if $j - w_i < 0$

$= V[3, 1]$

$= \boxed{10}$

$V[4, 2] = \max \{V[i-1, j], V_i + V[i-1, j-w_i]\}$

$= \max \{V[3, 2], 15 + V[3, 0]\}$

$= \max \{12, 15 + 0\}$

$= \boxed{15}$

$V[4, 3] = \max \{V[3, 3], 15 + V[3, 1]\}$

$= \max \{22, 15 + 10\}$

$= \boxed{25}$

$V[4, 4] = \max \{V[3, 4], 15 + V[3, 2]\}$

$= \max \{30, 15 + 12\}$

$= \boxed{30}$

$V[4, 5] = \max \{V[3, 5], 15 + V[3, 3]\}$

$= \max \{32, 15 + 22\}$

$= \boxed{37}$

❖ Now the table contains the following data.

		Capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$W_1 = 2$ $V_1 = 12$	1	0	0	12	12	12	12
$W_2 = 1$ $V_2 = 10$	2	0	10	12	22	22	22
$W_3 = 3$ $V_3 = 20$	3	0	10	12	22	30	32
$W_4 = 2$ $V_4 = 15$	4	0	10	15	25	30	37

Thus the maximum value is $V[4, 5] = \boxed{37}$

Find the Optimal Subset

$V[4, 5] \neq V[3, 5]$

- So item 4 included in the set.
- Remaining capacity of the knapsack = 5 – 2 = 3

$V[3, 3] \neq V[2, 3]$

3

So item 4 is not included.

- Check for the item 2.

$V[2, 3] \neq V[1, 3]$

So item 2 is included in the subset

- Remaining capacity is 3 – 1 = 2
- Check for item 1.

$V[1, 2] \neq V[0, 2]$

So the item 1 is included.

∴ The optimal subset is {1, 2, 4}

3.6.4. COMPLEXITY OF KNAPSACK PROBLEM

Space Complexity

Space for value array = n location

Space for weight array = n location

Capacity of Knapsack = 1 location

W

Total Space = $n^2 + 1$ location

Time Complexity

Running time for knapsack problem = $0(n \cdot w)$

To find the optimal subset = $0(n + w)$

3.7. MEMORY FUNCTIONS

Concept

It works on bottom up. It fills a table with solution to all smaller sub problem but each of them is solved only once.

The goal is to get a method that solves only sub problem and does it only once.

3.7.1. PROCEDURE FOR MEMORY FUNCTIONS

- It solves a given problem in top down manner
- It maintains a table in bottom up dynamic programming algorithm
- Initially all the table's entries are initialized with a special 'null' symbol to indicate that they have not yet been calculated.
- For calculating the new entries, the method checks the corresponding entry in the table first, if this entry is not "null" it is simply retrieved from the table, otherwise it is computed by the recursive call whose result is then recorded in the table.

3.7.2. ALGORITHM FOR MEMORY FUNCTION USING KNAPSACK PROBLEM

Algorithm MF knapsack (i, j)

// implements the memory function method for the knapsack problem.

// Input = i – indicates the no of 1st items.

j – indicates the knapsack's capacity

// Output: The value of an optimal feasible sub set of the first i items.

// It uses a global variables input array weights $[1 \ldots n]$, values $[1 \ldots n]$ and tables $V[0 \ldots n, 0 \ldots w]$ whose entries are initialized with – 1, except for row 0 and column 0 initialized with 0's

if $V[i, j] < 0$

if $j <$ weights $[i]$

Value = MF knapsack $(i-1, j)$

else

value = max (MF knapsack (*i*–1, *j*)
values [*i*] + MF knapsack (*i*–1, *j*-weights [*j*]))

V [*i*, *j*] = value

return V [*i*, *j*]

3.7.3. EXAMPLE FOR MEMORY FUNCTION USING KNAPSACK PROBLEM

Apply the bottom up dynamic programming algorithm to the following instance of the knapsack problem.

Item	Weight	Value
1	2	$ 12
2	1	$ 10
3	3	$ 20
4	2	$ 15

Capacity W = 5

- Initially row 0 and column 0 initialized with 0's

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$W_1 = 2$ $V_1 = 12$	1	0					
$W_2 = 1$ $V_2 = 10$	2	0					
$W_3 = 3$ $V_3 = 20$	3	0					
$W_4 = 2$ $V_4 = 15$	4	0					

- Compute V [1, 1] = V [0, 1] = 0
- Compute V [1, 2] = max (V [0, 2), 12 + V [0, 0]
  = max (0, 12 + 0)

= 12

- Similar way all the entries are calculated
- The table contains

	i	0	1	0	3	4	5
	0	0	0	0	0	0	0
$W_1 = 2$ $V_1 = 12$	1	0	0	12	-	12	12
$W_2 = 1$ $V_2 = 10$	2	0	-	12	22	-	22
$W_3 = 3$ $V_3 = 20$	3	0	-	-	22	-	32
$W_4 = 2$ $V_4 = 15$	4	0	-	-	-	-	37

3.7.4. MERITS AND DEMERITS

Merits

1. It solves common sub problem only once.
2. It requires less time and space

Demerits

1. Solution to some of these smaller sub smaller sub problems is often not necessary for setting solution to the given problem.

3.8. GREEDY METHOD

Concept

The greedy approach suggests that constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so for, until a complete solution to the problem is obtained.

3.8.1. GENERAL METHOD

Problems have *n* inputs and require us to obtain a subset that satisfies some constraints.

Some important terms in Greedy Techniques

Feasible

Any subset that satisfies problem constraint is called feasible.

Objective function

To find a feasible solution that either maximizes or minimizes a given objective function.

Optimal Solution

It has to be the best choice among all feasible solution available on that step.

Procedure

- *It suggests that algorithm works in stages.*
- *At each stage, a decision is made regarding whether a particular input is an optimal solution.*
- *This is done by considering the inputs are an order.*
- *If the inclusion of the next input into the partially constructed solution will result in feasible solution, then only the input will be added.*

Greedy Algorithm

```
Algorithm Greedy (a, n).
// a [1 : n] contains n inputs.
{ Solution = 0; // Initialise the solution.
for (i = 1; i < n; i++)
{ x = select (a)
  if Feasible (solution, x) then
   solution = union (solution, x);
}
return solution;
}
```


3.9. CONTAINER LOADING PROBLEM

Concept

A large ship is to be loaded with cargo. The cargo is containerized and all containers are the same size. Different containers may have different weights w_i be the weight of i^{th} container $\boxed{1 \le i \le n}$. The cargo capacity is C. Based on the greedy concept load the ship with the maximum number of containers.

3.9.1. PROCEDURE FOR CONTAINER LOADING

- The ship may be loaded in stages.
- At each stage we need to select a container to load.
- Select the container with least weight
- Check each time before the loading of container $\boxed{C[i] \text{ weight} \le = \text{capacity}}$
- The process is repeated until it reaches cargo capacity 'C'.

Feasible Solution

- Every set of x_i's (container) that satisfies the constraint $\sum_{i=1}^{n} w_i x_i < C$ then x_i is assigned to value 1. Otherwise it is assigned to ϕ.

$$\boxed{\sum_{i=1}^{n} w_i x_i < = C, x_i \in \{0, 1\}, 1 \le i \le n}$$

where w_i - weight of the container 'i'.

x_i - value of the container is assigned to '0' (or) 1. If the container is loaded, it is assigned to 1, otherwise it is ϕ.

c - cargo capacity

n - No. of containers in cargo

Optimal Solution

- Load the ship with the maximum number of containers.

❖ Every feasible solution that maximizes the $\sum_{i=1}^{n} x_i$ function is an optimal solution.

3.9.2. EXAMPLES FOR CONTAINER LOADING

The cargo contain 8 containers, $n = 88$, weight of the containers $\{w_1, w_2, w_3, \ldots w_8\} = \{100, 200, 50, 90, 150, 50, 20, 80\}$ and capacity of the cargo is 400. Find optimal solution for loading the containers.

1. Arrange the containers in ascending order of their weights
   $\{7,3,6,8,4,1,5,2\} = \{20, 50, 50, 80, 90, 100, 150, 200\}$
   Initially none of the containers will be loaded. The solution set becomes $\{0,0,0,0,0,0,0,0\}$
2. In stage 1, the container 7 is selected, whose weight is 20.
   Check the constraint $\sum w_i <= c$, i.e., $20 <= 400$ the container 7 is loaded
   X, value is assigned to 1.
   ∴ value of the solution set = $\{0,0,0,0,0,0,1,0\}$
3. In stage 2, the container 3 is selected whose weight is 50.
   Check the constraint $\sum w_i < c$, $50 + 20 < 400$ $70 <= 400$ x_3 value is assigned to 1 ∴ value is assigned to 1.
4. In stage 3, the container 6 is selected, whose weight is 50. $\sum w_i <= C$ i.e., $120 < 400$ x_6 value is assigneed to 1.
   ∴ value of the solution set = $\{0, 0, 1,0, 0,1, 1, 0\}$
5. In stage 4, the container 8 is seelcted, whose weight is 80,check the constraint
   $\sum w_i <= c$ i.e., $200 < 400$ x_8 value is assigned to 1.
   ∴ Value of the solution set = $\{0,0,1,0,0,1,1,1\}$
6. In stage 5, the container 4 is selcted, whose weight is 90
   Check the constraint $\sum w_i < C$ = i.e $\geq 290 < 490$ x_4 value is assigned to 1.
   ∴ value of the solution set = $\{0,0,1,1,0,1,1,1\}$

7. In stage 6, the container 1 is selected, whose weight is 100
   Check the constraint $\sum w_i <= C$ i.e $390 < 400$
   x_1 value is assigned to 1
   ∴ Value of the solution set = $\{1,0,1,1,0,1,1,1\}$
8. In stage 7, the container 5 is selected, whose weight is 150.
   Check the constraint $\sum w_i <= 540 < 400$ the container 5 is not loaded.
   ∴ Optimal solution $\sum x_i = 6$

3.9.3. ALGORITHM FOR CONTAINER LOADING

```
void container-loading (container *c, int capacity, int numberofcontainers, int * x)
  // c –capacity of containers
  {
  // Sort the container in ascending order of their weights
  Sort (c, numberofcontainers);
  n = number of containers;
  // Initialize the variable x
   for (i = 1; i <= n; i + +)
    x [i] = 0;
 I = 1;
 //Select the container in order of their weight
   While (I <= n && c[i] <= capacity)
{

// c[i] weight of the container i
   x[i] = 1; // the container 1 is loaded.
   capacity = capacity – c[i]; // remaining capacity of the cargo
   i++;
   }
}
```

3.9.4. COMPLEXITY OF CONTAINER LOADING

The complexity of algorithm measures by space and time complexity.

Space Complexity

Space for weight of n containers stored in an array $c[\,]$ = n locations

Space for indication of container loading stored in an array $x\,[\,]$ = n locations

Capacity of cargo c = 1 location

Number of containers 'n' = 1 location

Control variable 'i' = 1 location

$\therefore$ Total space required = $\boxed{2n+3}$

Time Complexity

1. Sorting the container increasing order of their weight using merge sort = $0(\log n)$
2. Algorithm for loading opf container = $0(n)$

$\therefore$ Time complexity = $\boxed{0(n \log n)}$

3.10. PRIM'S ALGORITHM

Concept

Based on greedy technique, prim's algorithm is used to construct a minimum spanning tree of a weighted connected graph.

Definition of Spanning tree

A spanning tree of a connected graph is its connected acyclic subgraph that contains all the vertices of the graph.

Definition of minimum spanning tree

A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights of all its edges.

Example

Compute the spanning & minimum spanning tree for the following graph.

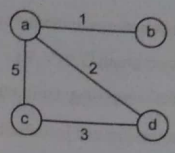
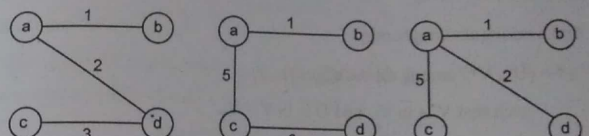


Fig. 3.21. Graph

Spanning tree drawn for the above graph.



$W(T_1) = 6$ $W(T_2) = 9$ $W(T_3) = 8$

Fig. 3.22. a, b, c Spanning tree

❖ Among the spanning trees Fig.3.22 a minimum spanning tree.

3.10.1. PROCEDURE FOR PRIM'S ALGORITHM

1. The initial sub tree contains single vertex selected arbitrarily from the set V of graph's vertices.
2. The current tree is expanded in greedy manner by attaching the nearest vertex to it.
3. The nearest vertex will have the smallest weight.
4. The algorithm stops, when all vertices have been included.
5. If n is the number of vertices in a graph then the total number of iterations is $n-1$.

3.10.2. PRIM'S ALGORITHM

Algorithm Prim (G)

{// Prim's algorithm for constructing a minimum spanning tree.

// Input: A weighted connected graph G = (V, E).

// Output: E_T, the set of edges composing a minimum spanning tree.

$V_T = \{V_0\}$ // The set of tree vertices can be initialized with any vertex.

$E_T = \varphi$

For $i = 1$ to $|V| - 1$ do

{ find a minimum weight edge

$e^* = (U^*, V^*)$ among all the edges (u, v)

such that V is in V_T and U is in $V - V_T$

$V_T = V_T \cup \{V^*\}$

$E_T = E_T \cup \{e^*\}$

}

return E_T

}

3.10.3. COMPLEXITY OF PRIM'S ALGORITHM

Complexity of an algorithm measured in terms of space and time.

Space complexity

Space for set of vertices = n locations

Space for set of edges = n locations

Space for control variables = 1 locations

Total space = $(2n + 1)$ locations

Time complexity

The efficiency of Prim's algorithm depends an data structures chosen for the graph.

1. Graph – represented as weighted matrix

If the graph is represented by its weight matrix and the priority queue is implemented as an unordered array.

Running time = $\boxed{0(|V^2|)}$

2. Adjacency list

If the graph is represented by its adjacency list and the priority queue is implemented as a min – heap

Running time = $\boxed{0((|E| \log |V|))}$

3.10.4. EXAMPLE FOR PRIM'S ALGORITHM

Example 1

Find minimum spanning tree for the following graph using prim's algorithm.

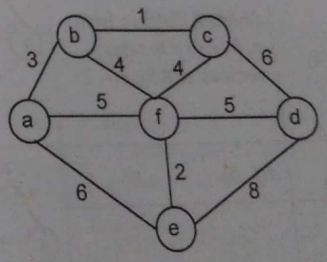
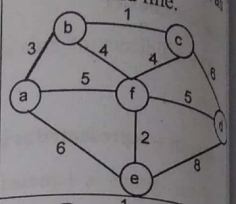
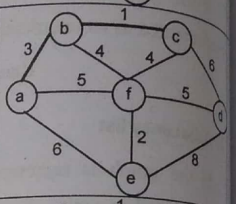
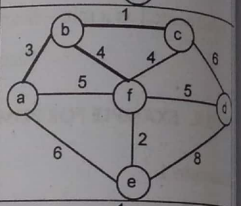
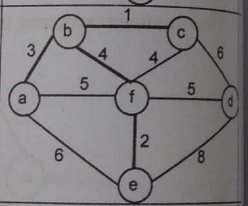
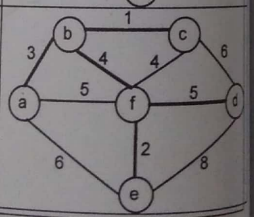


Fig. 3.23.

S.No.	Tree Vertices	Remaining Vertices	Illustration
1.	Initially a is included 'a' is starting vertex a $(-, -)$	Distance from a to other vertices. $b(a, 3)$ $c(-, \alpha)$, $d(-, \alpha)$ $e(a, 6)$, $f(a, 5)$	Included vertex shown as solid line. 
2.	Vertex 'b' is included $b(a, 3)$	Shortest distance from a or b to other vertices $C(b, 1)$, $d(-, \alpha)$, $e(a, 6)$, $f(b, 4)$	
3.	Vertex 'C' is included $C(b, 1)$	Shortest distance from a, b or c to other vertices $d(c, 6)$, $e(a, 6)$, $f(b, 4)$	
4.	Vertex f is included $f(b, 4)$	Shortest distance from a, b, c or f to other vertices. $d(8,5)$, $e(f, 2)$	
5.	Vertex e is included $e(f, 2)$	$d(f, 5)$	
6.	Vertex d is included $d(f, 2)$		

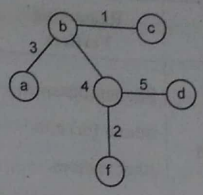


Fig. 3.24.

$\therefore$ Minimum spanning tree $= 3 + 1 + 4 + 2 + 5$

(Sum of edges or all vertices) $=$ 15

Example 2

Find the minimum spanning tree for the following graph.

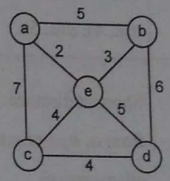
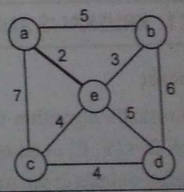
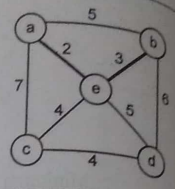
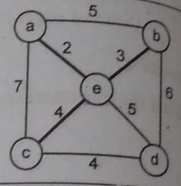
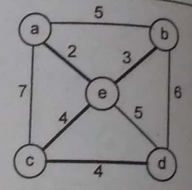


Fig. 3.25.

S.No.	Tree Vertices	Remaining Vertices	Illustration
1.	Initially a is included, a is the starting vertex $a(-, -)$	Distance from a to other vertices $b(a, 5)$, $C(a, 7)$, $e(a, 2)$, $d(-, \alpha)$	Included vertex shown as solid line. 

S.No.	Tree Vertices	Remaining Vertices	Illustration
2.	Vertex e is included $e(a, 2)$	Shortest distance from a (or) e to other vertices. $b(e, 3)$, $c(e, 4)$, $d(3, 5)$	
3.	Vertex b is included	Shortest distance from a, b or e to other vertices $c(e, 4)$, $d(e, 5)$	
4.	Vertex c is included	Shortest distance from a, b, c or e to other vertices $d(c, 4)$	
5.	Vertex d is included	$\therefore$ Minimum spanning tree	$= 2 + 4 + 3 + 4$ $= \boxed{13}$

3.11. KRUSKAL'S ALGORITHM

Concept

Kruskal's algorithm find a minimum spanning tree for a weighted connected graph $G = (V, E)$ as an acyclic sub graph with $|V| - 1$ edges for which the sum of the edge weights is the smallest.

3.11.1. PROCEDURE FOR KRUSKAL'S ALGORITHM

The algorithm constructs a minimum spanning tree as an expanding sequence of sub graph, which are always acyclic but are not necessarily connected on the intermediate stage of the algorithm.

1. The algorithm begins by sorting the graph's edges in non-decreasing order of their weights.
2. Starting with the empty sub graph, it scans the sorted list adding the next edge on the list to the current sub graph if such as inclusion does not create a cycle and simply skipping the edge otherwise.

3.11.2. ALGORITHM FOR KRUSKAL'S ALGORITHM

Algorithm Kruskal (G)

// Kruskal's algorithm for constructing a minimum spanning tree.

// Input: A weighted connected graph $G = (V, E)$

// Output: E_T, the set of edges composing a minimum spanning tree of G.

Sort E is non decreasing order of the edge

weights $W(ei_1) \leq \ldots\ldots \leq W(ei_{|E|})$

{

$E_T = 0$; Initialize the set of tree edges and its size.

e counter $= 0$;

$K = 0$; Initialize the number of processed edge.

While (ecounter $< |V| - 1$)

{ $K = K + 1$

if $E_T \cup \{ei_K\}$ is acyclic

$\{ E_T = E_T \cup \{e_k\};$

ecounter = ecounter + 1;

} }

return (T);

}

3.11.3. COMPLEXITY FOR KRUSKAL'S ALGORITHM

The complexity of an algorithm is measured in terms of Space and time.

Space complexity

Space for set of vertices = n locations

Space for set of edges = n locations

Space for control variables = 1 location

Total space = $(2n + 1)$ locations

Time Complexity

Tunning time of Kruskal's algorithm will be dominated by the time needed for sorting the edges weights of a given graph } $0 (| E | \log | E |)$

3.11.4. EXAMPLE FOR KRUSKAL'S ALGORITHM

Example 1

Find minimum spanning tree for the following graph using Kruskal's algorithm.

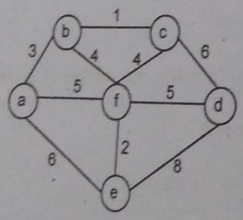
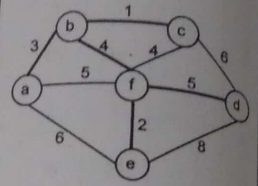
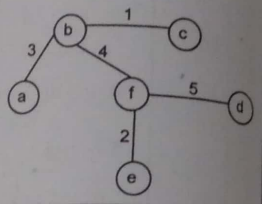


Fig. 3.26.

S. No.	Tree edges	Sorted list of edges					Illustration
		bc	ef	ab	bf	cf	
		1	2	3	4	4	
1.		af	df	ae	cd	de	
		5	5	6	6	8	
		bc	ef	ab	bf	cf	
	bc	1	2	3	4	4	
2.	1	af	df	ae	cd	de	
		5	5	6	6	8	
		bc	ef	ab	bf	cf	
	ef	1	2	3	4	4	
3.	2	af	df	ae	cd	de	
		5	5	6	6	8	
		bc	ef	ab	bf	cf	
	ab	1	2	3	4	4	
4.	3	af	df	ae	cd	de	
		5	5	6	6	8	

S. No.	Tree edges	Sorted list of edges	Illustration
		bc ef ab bf cf	a-f, c-f creates cycle so d-f selected.
		1 2 3 4 4	
5.	Bf 4	af **df** ae cd de	
		5 **5** 6 6 8	
6.	Df 5	∴ The obtained minimum spanning tree is ⟶	
		Sum of weights of tree = **15**	





Example 2

Find minimum spanning tree for the following graph using Kruskal's algorithm.

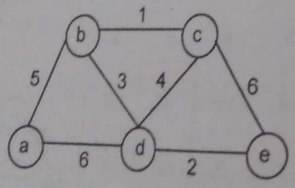
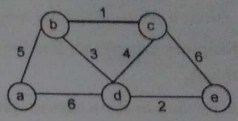
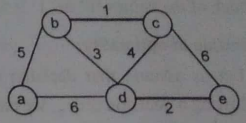
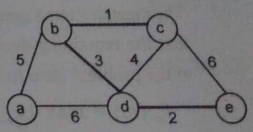


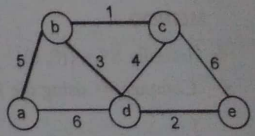
Fig. 3.27.

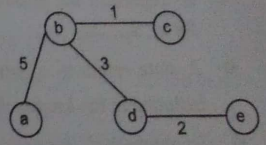
Tree edges	Sorted list of edges	Illustration
	bc de bd ed ab	
bc	1 2 3 4 5	
1	ce ad	
	6 6	
	bc **de** bd ed ab	
de	1 **2** 3 4 5	
2	ce ad	
	6 6	
	bc de **bd** ed ab	
bd	1 2 **3** 4 5	
3	ce ad	
	6 6	
	bc de bd ed **ab**	
ab	1 2 3 4 **5**	
5	ce ad	
	6 6	
	∴ The obtained minimum spanning tree = **11**	











3.12. 0/1 KNAPSACK PROBLEM

Concept:

Given n items of Known weights $w_1, w_2, \ldots w_n$ and profits $p_1, p_2 \ldots p_n$ and a Knapsack capacity m, find the most valuable subset of the items that fit in to the knapsack.

A solution to the Knapsack problem can be obtained by making a sequence of decision on the variables $x_1, x_2 \ldots x_n$. A decision variables x_i involves determining which of the values '0' or '1' is to be assigned to it.

Principle of optimality:

Let us assume that decision 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 in one of 2 possible stages,

1. The capacity remaining in the knapsack is m, no profit has gained.
2. The capacity remaining is $m - w_n$ and a profit of p_n has gained. It is clear that the remaining decisions $x_{n-1} \ldots x_n$ must be optimal with respect to the problem state resulting from the decision on x_n.

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

3.12.1. PROCEDURE TO FIND 0/1 KNAPSACK PROBLEM USING GENERAL METHOD

- Initially $S^0 = \{(0, 0)\}$
- Compute S^i using the formula $S_1^i = \{(P, w)/(P - p_{i+1}, w - w_{i+1} \in S^i\}$
- S^i represent the possible states resulting from the 2^i decision sequence for $x_1, x_2 \ldots x_n$.
- A state refers to a pair (P_j, W_j), W_j being the total weight of objects included in the knapsack and P_j being the corresponding profit.
- To obtain S^{i+1}, we note that the possibilities for x_{i+1} are $x_{i+1} = 0$, $x_{i+1} = 1$
- When $x_{i+1} = 0$, the resulting states are the same as s_i.

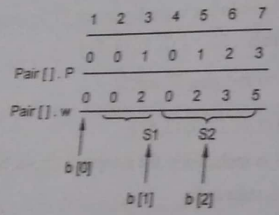
- When $x_{i+1} = 1$, the resulting states, are obtained by adding (P_{i+1}, W_{i+1}) to each state in s_i.
- S^{i+1} can be computed by merging the states S^i and S_1^i together.
- If S^{i+1} contains 2 pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \le P_k$ and $W_j \ge W_k$ then the pairs (P_j, W_j) can be discarded. This pair is called dominated tuples.
- The dominated tuples are deleted.

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.

- Initially $S^0 = \{(0, 0)\}$
- The object 1 is included in the Knapsack $S_1^0 = \{(1, 2)\}$
- S^1 is obtained by merging of S^0 and S_1^0 $S1 = \{(0,0), (1,2)\}$
- The object 2 is included in the Knapsack S_1^1 is obtained by adding (2, 3) to each state in S^1. $S_1^1 = \{(2, 3), (3, 5)\}$
- Compute S2 by merging S_1^0 and S_1^1
- $S^2 = \{(0, 0) (1,1), (2, 3) (3, 5)\}$
- The object 3 is included in the Knapsack (5, 4). S_1^2 is obtained by adding (5, 4) to each state in S_0^2
- $S_1^2 = \{(5, 4), (6, 6) (7, 7) (8, 9)\}$
- S_0^3 computed by merging of S_0^2 and S_1^2
- $S3 = \{(0, 0), (1, 2), (2,3), (3,5), (5,4), (6,6), (7,7), (8,9)\}$
- (3, 5) and (5, 4), $P_j \le P_k$ ie $3 < 5$ & $W_j > W_k$ ie $5 > 4$. So the pair (3, 5) dominated tuples eliminated.
- $\therefore S^3 = \{(0, 0) (1, 2), (2, 3) (5, 4), (6, 6), (7, 7), (8, 9)\}$

- When all pairs (P_j, W_j) with $W_j > m$ are eliminated from S_t (ie) the pairs $\{(7, 7)\ (8, 9)\} > m$ ie 6. So these pair eliminated from S^3.
  $S^3 = \{(0, 0)\ (1, 2), (2, 3), (5, 4)\ (6, 6)\}$
- With $m = 6$, the value of f_3 (6) is given by last tuple (6, 6). The tuple $(6, 6) \in S^2$ set $x_3 = 1$.
- The pair (6, 6) come from the pair $(6 - P_3, 6 - W_3)$ ie $(6 - 5, 6 - 4) =$ (1, 2) Hence $(1, 2) \in S^2$ so we can set $x_2 = 0$.
- Since $(1, 2) \notin S^0$, we obtain $x_1 = 1$
- ∴ The optimal solution is $(x_1, x_2, x_3) = (1, 0, 1)$
- We can use an array pair [ ] to represent all the pairs (P, *w*). The P values are stored in pair [ ]. P and the *w* values in pair [ ]. *w*.
- Sets $S^0, S^1, \ldots S^{n-1}$ can be stored adjacent to each other.
- Representation of S^0, S^1, S^2



3.12.2. GENERAL ALGORITHM FOR 0/1 KNAPSACK PROBLEM

```
void DKP (p, w, n, m)
{   //p  - Profit of objects
    //w  - weight of object
    //n  - number of objects
    //m  - capacity of Knapsack
```



```
    S^0 = {(0, 0)}
for (i = 1; i < n; i++)
{   S^(i+1) = {(P w) | (P − p_i, w − w_i) ∈ S^(i+1) and w ≤ m};
    Si      = merge purge (S^(i-1), S^(i-1));
}
(PX, WX)  = last pair in S^(n-1);
(PY, WY)  = (P^1 + P_n, W^1 + W_n)
//where w' is the largest of w in any pair in S^(n-1) such that w + W_n ≤ m;
//satrace back of x_n, x_(n-1)...x_1
if (PX > PY) then X_n = 0
else
   x_n = 1;
Traceback for (x_(n-1) .....x_1);
}
```

3.12.3. PROCEDURE FOR 0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

- Initially set $S^0 n = \{0, 0\}$
- Each S^i, $i > 0$ is obtained by merging S^{i-1} and S_1^{i-1}
- Each pair (P, *w*) in S^i has an integer
  P and $\boxed{P \le \sum_{1 \le j \le i} P_j}$
- W_j's are integers each *w* is an integer and $w \le m$
- In any S^i the pairs have distinct *w* values and also distinct P values.
- Let L be an estimate on the value of an optimal solution such that $\boxed{f_n\ (m) \ge L}$

- Assign PLEFT $(i) = \sum_{i<j\le n} P_j$
- If S_i contains a tuple (P, w) such that $P + P$ LEFT $(i) < L$, then (P, w) can be purged from S^i.
- A simple way to estimate L such that $L \le f_n(m)$ is to consider the last pair (P, w) in S^i,

$$\boxed{P < f_n(m)}$$

Example

Consider the following instances of the Knapsack problem $n = 6$, $(P_1, P_2, P_3, P_4, P_5, P_6) = (W_1, W_2, W_3, W_4, W_5, W_6) = (100, 50, 20, 10, 7, 3)$ and $m = 165$.

Attempting to fill the Knapsack using object in the order 1, 2, 3, 4, 5 and 6. We see that objects 1, 2, 4, and 6 fit in to the Knapsack and yield a profit of 163 and a capacity utilization of 163.

1. $L = 163$ as a value with the property $L \le f_n(m)$ (ie) $163 < 165$
2. $P_i = W_i$, every pair $(P, w) \in S^i\ 0 \le i \le 6$ has $P = w$
3. Each pair can be repeated by singleton P or w.
4. PLEFT $(i) = \sum_{i<j\le n} P_j$

PLEFT (6) = 0

PLEFT (5) = $\sum(0+3) = 3$

PLEFT (4) = $\sum(0+3+7) = 10$

PLEFT (3) = $\sum(0+3+7+10) = 20$

PLEFT (2) = $\sum(0+3+7+10+20) = 40$

PLEFT (1) = $\sum(0+3+7+10+20+50) = 90$

PLEFT (0) = $\sum(0+3+7+10+20+50+100) = 190$

5. Initially

$S^0 = \{0\}, S_1^0 = \{100\}$

$S_0^1 = \{0, 100\}, S_1^1 = \{50, 150\}$

$S_0^2 = \{0, 50, 100, 150\}, S_1^2 = \{20, 70, 120, 170\}$

$m < 170$, 170 eliminated from the set S_1^2

Now S_1^2 contains $S_1^2 = \{20, 70, 120\}$

$S_0^3 = \{0, 20, 50, 70, 100, 120, 150\}$

$S_1^3 = \{10, 30, 60, 80, 110, 130, 160\}$

S_0^4 = merging of S_0^3 and S_1^3

$S_0^4 = \{0, 10, 20, 30, 50, 60, 70, 80, 100, 110, 120, 130, 150, 160\}$

S_1^4 = Obtained by adding 7 to S_0^4

$S_1^4 = \{7, 17, 27, 37, 57, 67, 77, 87, 107, 117, 127, 137, 157, 167\}$

S_0^5 = Obtained by merging of $\boxed{S_0^4 \text{ \& } S_1^4}$

$S_0^5 = \{0, 7, 10, 17, 27, 30, 37, 50, 57, 60, 67, 70, 77, 80, 87, 100, 107, 110, 117, 127, 130, 150, 157, 160\}$

S_1^5 Obtained by adding 3 to S_0^5 & 6 (165) can be determined by the tuple ???

$S_1^5 = \{3, 10, 13, 20, 30, 33, 40, 53, 60, 63, 70, 73, 80, 83, 90, 103, 110, 123, 130, 133, 140, 153, 160, 163\}$

& 6 (165) can be determined by the tuple $\{163\}$ in S_1^5

Example 2

Let us consider the instance given by the following data:

Item	weight	profit
1	2	12
2	1	10
3	3	20
4	2	15

Capacity w = 5, find the optimal solution for Knapsack problem

1. Define the initial condition

$P[0, j] = 0$ for $j >= 0$ & $P[i, 0] = 0$ for $i >= 0$

So, $P(0, 0) = 0$ $P(1, 0) = 0$

$P(0, 1) = 0$ $P(2, 0) = 0$

$P(0, 2) = 0$ $P(3, 0) = 0$

$P(0, 3) = 0$ $P(4, 0) = 0$

$P(0, 4) = 0$

2. The item i's weight & profit $W_1 = 2, P_1 = 12$

Compute $P(i, j)$ using the formula

$P(i, j) = \max \{P(i-1, j), Pi + P(i-1, j - Wi)\}$ if $j - Wi >= 0$

$P(i, j) = P(i-1, j)$ if $j - Wi < 0$

Compute $P(1, 2) = \text{Max}\{P(i\text{-}1, j), P_i + P(i-1, j-w_i)\}$

Compute $P(1, 2) = \text{Max}\{P(0, 2), 12 + P(0, 0)\}$

$= \text{Max}\{0, 12 + 0\}$

$= 12$

Compute $P(1, 3) = \max\{P(0, 3), 12 + 0\}$

$= \max\{0, 12\}$

$= 12$

Compute $P(1, 4) = \max\{P(0, 4), 12 + P(0, 3)\}$

$= \max(0, 12 + 0)$

$= 12$

Compute $P(1, 5) = \max\{P(0, 5), 12 + P(0, 3)\}$

$= \max(0, 12 + 0)$

$= 12$

3. The item 2's weight and profit $W_2 = 1, P_2 = 10$

Compute $P(2, 1) = \max\{P(1, 1), 10 + P(1, 0)\}$

$= \max(0, 10 + 0)$

$= 10$

Compute $P(2, 2) = \max\{P(1, 2), 10 + 0(1, 1)\}$

$= \max\{12, 10 + 0\}$

$= \max\{12, 10\}$

$= 12$

Compute $P(2, 3) = \max\{P(1, 3), 10 + P(1, 2)\}$

$= \max\{12, 10 + 12\}$

$= 22$

Compute $P(2, 4) = \max\{P(1, 4), 10 + P(1, 3)\}$

$= \max\{12, 10 + 12\}$

$= 22$

4. The item 3's weight and profit $W_3 = 3, P_3 = 20$

Compute $P(3, 1) = \max\{P(2, 1)\}$

$= 10$

Compute $P(3, 2) = \max\{P(2, 2)\}$

$= 12$

Compute $P(3, 3) = \max\{P(2, 3), 20 + P(2, 0)\}$

$= \max\{22, 20 + 0\}$

$= 22$

Compute $P(3, 4) = \max\{P(2, 4), 20 + P(2, 1)\}$

$= \max\{22, 20 + 10\}$

$= 30$

Compute $P(3, 5) = \max\{P(2, 5), 20 + P(2, 2)\}$

$= \max \{22, 20 + 12\}$

$= \boxed{32}$

5. The item 4's weight and profit $W_4 = 2$, $P_4 = 15$

Compute $P(4, 1) = \max \{P(3, 1)\}$

$= \boxed{10}$

Compute $P(4, 2) = \max \{P(3, 2), 15 + P(3, 0)\}$

$= \max \{12, 15 + 0\}$

$= \boxed{10}$

Compute $P(4, 3) = \max \{P(3, 3), 15 + P(3, 1)\}$

$= \max \{22, 15 + 10\}$

$= \boxed{10}$

Compute $P(4, 4) = \max \{P(3, 4), 15 + P(3, 2)\}$

$= \max \{30, 15 + 12\}$

$= \boxed{10}$

Compute $(4,5) = \max \{P(3, 5), 15 + P(3, 3)\}$

$= \max \{32, 15 + 22\}$

$= \boxed{37}$

	i \ j	0	1	2	3	4	5
	0						
	1	0	0	0	0	0	0
$w_1 = 2,\ P_1 = 12$	2	0	0	12	12	12	12
$w_2 = 1,\ P_2 = 10$	3	0	10	12	22	22	22
$w_3 = 3,\ P_3 = 20$	4	0	10	12	22	30	32
$w_4 = 2,\ P_4 = 15$	5	0	10	15	25	30	37

- Thus the maximum value is $P[4, 5] = \boxed{37}$
- We can find the composition of an optimal subset by tracing back the computation of the entry in the table.
- $P[4, 5] \neq v[3, 5]$ so item 4 included in the solution set
- Knapsack capacity = 5, weight of an item 4 is 2, $5 - 2 = 3$ remaining units of the knapsack capacity.
- $P[3, 3] = P[2, 3]$ so item 3 is not included
- $P[2, 3] \neq P[1, 3]$ so item 2 is included in the solution set
- The remaining capacity of knapsack is 3, weight of item 2 is 1 i.e., $3 - 1 = 2$.
- $P[1, 2] \neq P[0, 2]$ item 1 is included in the solution set

  $\therefore$ Optimal solution $\boxed{\{1, 2, 4\}}$

3.12.4. ALGORITHM FOR 0/1 KNAPSACK PROBLEM

```
void D knap (P, w, x, n, m)
{
b [0] = 1;
Pair [1]. P = Pair [1] 2 = 0.0 // S^0 set
t = 1, n = 1;
b [1] next = 2 // next free spot in pair [ ]
for (i = 1; i < n; i++)
{
// generate S^i
k = t;
u = Largest (pair, w, t, h, i, m)
for (int j = t; j < u; J++)
```

```
{
// 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))
{
Pair [next] . p = Pair [k] . p;
Pair [next] . w = pair [k] . w;
next = next + 1;
k = k + 1;
}
if ((k < = h) & (pair [k] . w = ww))
{
if (pp < pair [k] . p)
pp = pair [k] . p;
k = k + 1;
}
if (pp > pair [next – 1] . p)
{
Pair [next] . p = pp;
Pair [next] . w = ww;
next = next + 1;
}
```



```
While ((k < = h) & (pair [k] . p < = pair [next – 1] . p))
k = k + 1;
}
// merging remaining terms from S^i – 1
While (k < = h)
{
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);
}
```

3.12.5. COMPLEXITY OF 0/1 KNAPSACK PROBLEM

Time complexity

Time needed to compute all the S^i, S is $0(2^n)$

Time needed for trace back = $0(n^2)$

$\therefore$ Total time needed for 0/1 knapsack problem = $0(2^n)$

3.13. OPTIMAL MERGE PATTERN

Concept

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.

Procedure

Two-way merge patterns can be represented by binary merge trees.

Let us consider a set of n sorted files $\{f_1, f_2, f_3, \ldots, f_n\}$.

Initially, each element of this is considered as a single node binary tree.

To find this optimal solution, the optimal merge pattern algorithm is used.

Finally, the weight of the root node represents the optimal cost.

Algorithm: OPTTREE (n)

```
for i := 1 to n – 1 do
  declare a new node
  node->leftchild := tree (list)
  node->rightchild :=tree (list)
  node->weight:=(node->leftchild)->weight)+(node->rightchild)->weight)
  insert (list, node);
return tree (list);
```

Example: Merge the following files, f_1, f_2, f_3, f_4 and f_5 with sizes 20, 30, 10, 5 and 30 so on using optimal merge pattern.

1. Sorting the files according to their size in an ascending order, we get the following sequence – f_4, f_3, f_1, f_2, f_5

2. Merge the files using optimal merge pattern algorithm

m_1 = **merge** f_4 **and** $f_3 \Rightarrow 5 + 10 = 15$

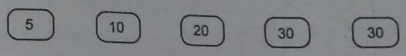
m_2 = **merge** M_1 **and** $f_1 \Rightarrow 15 + 20 = 35$

m_3 = **merge** M_2 **and** $f_2 \Rightarrow 35 + 30 = 65$

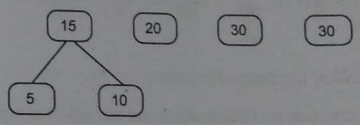
m_4 = **merge** M_3 **and** $f_5 \Rightarrow 65 + 30 = 95$

Therefore, the total number of operations is $15 + 35 + 65 + 95 = 210$

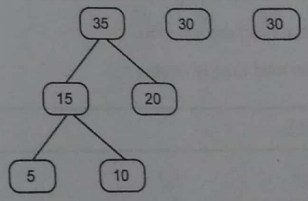
3. Initial Set



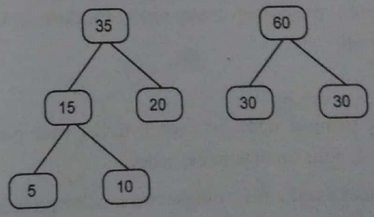
After merging first two sequence



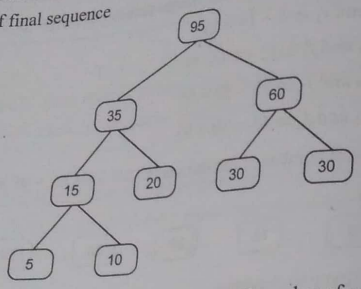
After merging next two sequence



Based on the ascending order, merging of last two sequence



Merging of final sequence



The optimal solution is $(15 + 35 + 60 + 95) = 205$ number of comparisons.

Time Complexity

- The algorithm iterates (n-1) times.
- At every iteration two delete-mins and one insert is performed.
- The 3 operations take O(log n) in each iteration.
- O(n) for initial heap construction.
- Therefore the total time is O(nlog n).

3.14. HUFFMAN TREE

Concept

Binary tree with each non-terminal node having 2 children. It gives optimal (min average code-length) prefix-free binary code to each $I_i \in \Sigma_o$ for a given probabilities $p(I_i) > 0$.

Procedure

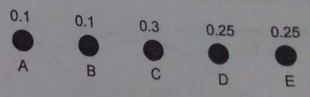
1. Create a terminal node for each $I_i \in \Sigma o$, with probability $p(I_i)$ and consider L = the set of terminal nodes.
2. Select nodes x and y in L with the two smallest probabilities.

3. Replace x and y in L by a node with probability $p(x) + p(y)$.
4. Also, create a node in the tree which is the parent of x and y.
5. Repeat (2) - (4) until $|L| = 1$.

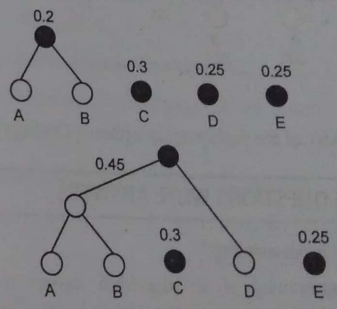
Example 1 : Draw the Huffman tree for the following list of elements.

$\Sigma 0 = \{A, B, \cdots, E\}$ and $p(A) = 0.1 = p(B)$, $p(C) = 0.3$, $p(D) = p(E) = 0.25$. The nodes in L are shown shaded.

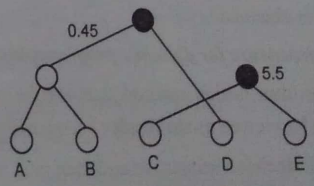
1. The set of nodes in L



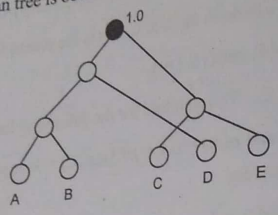
2. Select the nodes with least probabilities such as A and B are considered.



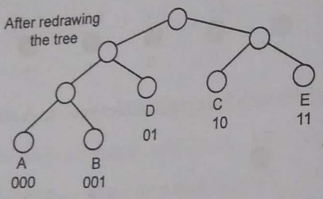
4. Again the nodes C and E considered.



5. Final huffman tree is obtained by merging of this two trees



6. The resultant huffman tree is



Time Complexity

The time complexity of the Huffman algorithm is O(nlogn).

3.15. TWO MARKS QUESTIONS WITH ANSWERS

1. ***What is dynamic programming?***

Dynamic programming is an algorithm design technique for solving problems with overlapping sub problems. The smaller sub programs are solved only once and recording the results in a table from which solution to the original problem is obtained.

2. ***What is general procedure for dynamic programming?***

- It is an algorithm design method that can be used when solution to a problem can be viewed as the result of sequence of decisions.
- Enumerate all decision sequences and then pick out the best.
- Optimal sequence of decision is obtained.

3. ***What is the formula for binomial coefficient?***

Binomial Coefficient Formula

$(a+b)^n = C(n,0)\,a^n + C(n-1)\,a^{n-1}\cdot b + C(n,i)\,a^{n-i}\,b^i + \ldots C(n,n)\,b^n$

4. ***Define Warshall's Algorithm.***

Warshall's Algorithm

It is an application of dynamic programming technique. It compute the transitive closure of a directed graph.

Transitive Closure

The transitive closure of a directed graph with n vertices can be defined as the n by n Boolean matrix.

$T = \{ t_{ij} \}$ in which the element in the i^{th} row$(1 \le i \le n)$ and j^{th} column $(1 \le j \le n)$ is 1, if there exists a non trivial directed path from the i^{th} vertex to the j^{th} vertex, otherwise t_{ij} is 0.

5. ***What is the complexity for Warshall's algorithm?***

Complexity of an algorithm depends on time and space.

Space Complexity

Space for adjacency matrix A $(n \times n) = n^2$ location

Control variable i, j, k = 3 location

$\therefore$ Total space = $\boxed{n^2+3}$ location

Time Complexity

The algorithm requires 3 loops = $n \times n \times n$

Order of time complexity = $0(n^3)$

6. ***Define Floyd's algorithm.***

Given a weighted graph, Floyd's algorithm is used to find the distance from each vertex to all other vertices.

The graph may be either directed (or) undirected graph.

7. *What is the complexity of Floyd's algorithm?*

Complexity of an algorithm depends on space and time.

Space Complexity

Storage space for weighted array = n^2 location

Storage space for control variable = 4 location

(i, j, k, n)

$\therefore$ Total space = $\boxed{n^2+4}$ location

Time complexity

For finding all pair shortest path algorithm uses 3 loops

$\therefore$ Time complexity = $\boxed{0\,(n^3)}$

8. *What is meant by optimal binary search tree?*

Dynamic programming can be used for constructing an optimal binary search tree for a given set of keys and known probabilities of searching for them. If probabilities of searching to elements of set are known, it is natural have an optimal binary search for which the average number of comparisons in a search is the smallest possible.

9. *What is the formula for finding possible binary search for the given n keys?*

The total number of binary search tree with 'n' keys is equal to the n^{th} Catalan number

$$\boxed{C(n) = \binom{2n}{n}\frac{1}{n+1} \text{ for } n > 0,\ C(0) = 1}$$

10. *What is the complexity of optimal binary search tree?*

The complexity of an algorithm depends on space and time.

Space Complexity

Space for cost array = n^2 locations

Space for root array = n location

Space for search probabilities = n location

Space for control variables = 3 locations

$\therefore$ Total space = $\boxed{n^2+2n+3}$ locations

Time complexity

Entries in the root tables one always non decreasing order along each row and column.

The limits values of $R[i, j]$ to the range $R[i, j-1] \ldots R[i+1, j]$ and makes it possible to reduce the running time of the algorithm = $\boxed{0\,n^2}$

11. *What are the advantages of optimal search tree?*

Merits

- It is one of the most important data structure in computer science.
- It is used to implement a dictionary, a set of elements with the operation of searching, insertion and deletion.

12. *Define $\frac{0}{1}$ knapsack problem.*

Given 'n' items of known weights $w_1, w_2, \ldots w_n$ and values $V_1, V_2 \ldots V_n$ and a knapsack of capacity W, fid the most valuable subset of the items that fit into the knapsack.

13. *What is the recurrence relation of $\frac{0}{1}$ Knapsack problem?*

The recurrence reaction

$$V[i,j] = \begin{cases} \max\{V[i-1, j], V_i + V[i-1, j-w_i]\} \\ \quad \text{if } j - w_i \geq 0 \\ V[i-1, j] \text{ if } j - w_i < 0 \end{cases}$$

14. *What is the time complexity of $\frac{0}{1}$ knapsack problem?*

Complexity of an algorithm depends on space and time.

Space Complexity

Space for value array = n location

Space for weight array = n location

Capacity of knapsack W = 1 location

$\therefore$ Total space = $\boxed{n^2+1}$ location

Time Complexity

Running time for knapsack problem $= O(n \cdot w)$

To find the optimal subset $= O(n + w)$

15. *What is meant by virtual initialization?*

Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Whenever a new value needs to be calculated, the method checks the corresponding entry in the table if the entry is not null, it is simply retrieved from the table, otherwise it is computed by the recursive call.

16. *Explain principle of optimality or define optimality.*

The principle of optimality says that an optimal solution to any instance of an optimization problem is composed of optimal solution to its sub instances.

17. *What is the formula used in Floyd's algorithm.*

All Pair Shortest Path

$$A^k(i,j) = \min\left\{A^{(k-1)}_{(i,j)}, A^{(k-1)}_{(i,k)} + A^{(k-1)}_{(k,j)}, \right\} k \geq 1$$

$$A^0(i,j) = W(i,j)$$

Where W = –Weight matrix

k - Intermediate vertex

18. *What are the features of dynamic programming?*

- It is a technique for solving problems with overlapping sub problems.
- It solves the smaller sub problem only once.
- It can be refined to avoid using extra space.

19. *Define optimal solution?*

Given a problem with n inputs, we obtain a subset that satisfies some constraints. Any subset which either maximizes a given objective these constraints is called a feasible solution.

A feasible solution which either maximizes or minimizes a given objective function is called optimal solution.

20. *What is meant by memory function technique?*

The memory function technique seeks to combine strengths of the top down & bottom up approaches to solving problems with over lapping sub problem. It solves sub problem only once and recording their solution in table.

3.16. REVIEW QUESTIONS

TWO MARKS QUESTIONS

1. What is dynamic programming?
2. What is the formula for binomial coefficient?
3. Define Huffman tree.
4. Define Floyd's algorithm.
5. What is the complexity of Floyd's algorithm?
6. What is the complexity of optimal binary search tree?
7. Explain principle of optimality?
8. Define $\frac{0}{1}$ knapsack problem.
9. What are the features of dynamic programming?
10. Define optimal solution?

16 MARKS QUESTIONS

1. Explain in detail about Optimal merge pattern algorithm with suitable examples.

(16) (May/June 200

2. Explain an algorithm to find optimal search tree with example.

(16) (May/June 20

3. Write Floyd's algorithm. Illustrate the algorithm by solving all-pairs sho path problem for the following digraph. (16) (Nov/Dec 2

$$\begin{bmatrix} 0 & 2 & \alpha & 1 & 8 \\ 6 & 0 & 3 & 2 & \alpha \\ \alpha & \alpha & 0 & 4 & \alpha \\ \alpha & \alpha & 2 & 0 & 3 \\ 3 & \alpha & \alpha & \alpha & 0 \end{bmatrix}$$

4. (*i*) Explain the concept of memory function. (6)

   (*ii*) Give a dynamic formulation for solving optimal binary search tree problem. **(10) (Nov/Dec 2009)**

5. Apply the bottom up dynamic programming algorithm to the following instance of the knapsack problem.

Item	Weight	Value
1	3	$ 25
2	2	$ 20
3	1	$ 15
4	4	$ 40
5	5	$ 50

Capacity W = 6