t all.

2

UNIT-III Dynamic programming

Dynamic programming:-

It is an algorithm design method, used when Solution to a problem can be viewed (in) as result of a sequence of decisions.

=) Knapsack, (Greedy) Shortest path.

Principle of optimality:-

optimal Sequence of decisions has the property that whatever initial State & decision are, the remaining decisions must constitute an optimal decision Sequence with regard to the State returning from the first decision.

=) Algorithm Allpaths(cost, A, n)

```
{
  // cost[1:n, 1:n] is adjacency matrix of a graph
  // with n vertices, A[i,j] is shortest path
  // from vertex, i to vertex j, cost[i,i]=0.0, for 1≤i≤n
   for i = 1 to n do
      for j=1 to n do
          A[i,j]:=cost[i,j];
   for K=1 to n do
      for i=1 to n do
      for j=1 to n do
```

$A(i,j) := \min(A[i,j], A[i,k] + A[k,j]);$

}

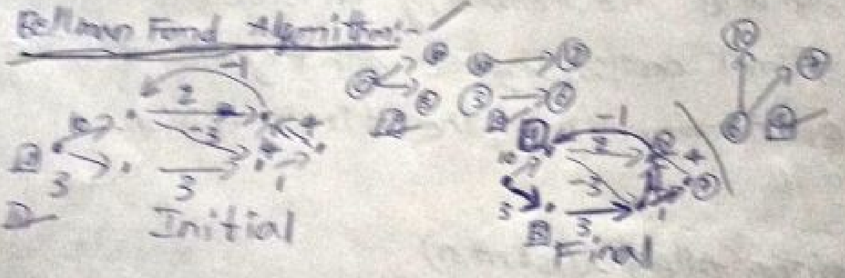
Recurrence Relation for APS:-

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

From \ To	1	2	3
1	0	1	∞
2	-2	0	∞
3	∞	∞	0

complexity, $\theta(n^3)$

Bellman Ford Algorithm:-



Initial Final

Algorithm BellmanFord (v, cost, dist, n)

```
{
    for i = 1 to n do
        dist[i] := cost(v, i);
    for k = 2 to n-1 do
        for each u such that u ≠ v and u has
                    at least one incoming edge do
            for each (i, u) in the graph do
            if dist[u] > dist[i] + cost[i,u] then
```

dist[u] := dis

}

Complexity

=> 0/1 Knapsa

	0/1	No
50	50	
60	0	
70	70	

profit | 9

Ex:-

=> n = 3,

[illegible]
}

Complexity $O(n^2)$

=> 0/1 Knapsack:

Item want to be taken completely.

Normal Knapsack – Item can be taken partially.
we want to gain maximum profit.

	0/1	Normal
50	50	50
60	0	60
70	70	10
profit	9	5[illegible]

50 – 3
60 – 2
70 – 6

k = 180

To take Dynamic programming for:-

a) To make optimal solution

b) It make sequential solutions.

Ex:-
=> $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(P_1, P_2, P_3) = (1, 2, 5)$, $m = 6$

$S^0 = \{(0,0)\}$; $S^0_1 = \{(1,2)\}$

$S^1 = \{(0,0), (1,2)\}$; $S^1_1 = \{\cancel{(1,2)}, (2,3), (3,5)\}$

$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$; $S^2_1 = \{(5,4), (6,6), \cancel{(7,7)}, (8,9)\}$

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

=> (3,5) is not useful

So, it is not in P_3.
Ignored in the P_3 list.
The optimal Solution is $(x_1, x_2, x_3) = (1, 0, 1)$

=> matrix

matrices

=> Algorithm DKP (p, w, n, m)

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

PX > PY
PX > PY

Time complexity:- $O(2^n)$

⇒ matrix chain multiplication:-

→ we have $A_1, A_2, \ldots A_n$ of n matrices to be multiplied & we compute $A_1 A_2 \ldots A_n$.

→ can be solved by standard Algorithm for multiplying pairs of matrices as a subroutine, once parenthesized it to solve ambiguities in how matrices multiplied together.

→ matrix multiplication is Associative (all will give the same result).

$A(B*C) = (A*B)C$.

→ Product (matrices) is fully parenthesized, if it 1 (or) 2 fully parenthesized matrix products, surrounded by parenthesis.

→ if chain of matrices, A_1, A_2, A_3, A_4 we can parenthesize the product A_1, A_2, A_3, A_4 in five distinct ways.

$(A_1(A_2(A_3*A_4)))$

$(A_1((A_2*A_3)A_4))$

$((A_1(A_2*A_3))A_4)$

$(((A_1*A_2)A_3)A_4)$

$((A_1*A_2)(A_3*A_4))$

→ How we paranthesize a chain of matrices can have a dramatic impact on the cost of Evaluating the product.

Scanned By Camera Scanner

→ No. of Columns in 1st matrix
⇒ No. of Rows in 2nd matrix

→ Size of output matrix ⇒ No. of Rows in 1st matrix * No. of Columns in 2nd matrix

⇒ $\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$

$3 \times 2 \quad 2 \times 3 \quad 3 \times 3$

multiplications required $= 3 \times 2 \times 3$

⇒ $A_1 = \text{size}(10 \times 100)$
$A_2 = \text{size}(100 \times 5)$
$A_3 = \text{size}(5 \times 50)$

$((A_1 * A_2) A_3)$

$(10 \times 100)(100 \times 5) \quad 10 \times 5$

A_1 & A_2 (10×5)

$((10 * 100) 5) = 5000$ Scalar multiplications for A_1 and A_2

$((10 * 5 \times 50) = 2500$ Scalar multiplications for A_3

$\underline{7500}$

$(A_1 (A_2 * A_3))$

$(100 (5 * 50)) = 25{,}000$ Scalar multiplication

$(100 \times 5)(5 \times 50) \quad 100, 50$ A_2 & $A_3 (100 \times 50)$

$(10 (100 * 50)) = 50{,}000$ Scalar multiplication to multiply A_1

$\underline{75{,}000}$

⇒ Algorithm for matrix-multiplication
if A.co
else

⇒ Travelli

we wand
every

The problem

⇒ Algorithm for matrix multiplication:-

```
matrix-multiply(A, B)
    if A.columns ≠ B.rows
        error "incompatible dimensions"
    else let C be a new A.rows × B.columns matrix
        for i = 1 to A.rows
            for j = 1 to B.columns
                c_ij = 0
                for k = 1 to A.columns
                    c_ij = c_ij + a_ik · b_kj
    return C
```

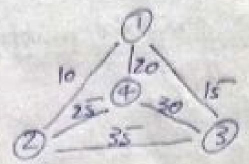
⇒ Travelling salesman problem:-

Cities, with distance b/w them we want to find shortest path/route that visits every city exactly once & returns to the starting point.

Hamiltonian cycle - To find if there exist a tour that visits every city exactly once.

The problem is to find a min weight Hamiltonian cycle.

Ex:-



1 → 2 → 4 → 3 → 1

10+25+30+15 = cost = 80

=) Dynamic programming for TSP:-

vertices = 1, 2, 3, ... n

1 as starting point & ending point.

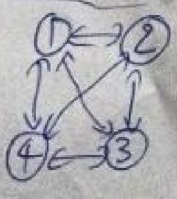
→ For other vertex i (not 1), we find min. cost path from '1', i as ending point & all vertices appearing exactly once.

→ Cost as cost(i), cost (corresponding cycle) is cost(i) + dist(i,1) where dist(i,1) is the distance from i to 1.

→ Finally, return min. of all [cost(i) + dist(i,1)] values

→ To calculate cost(i) using Dynamic programming, we need to have some recursive relation in terms of sub-problems. Let, C(S,i) be min. cost path visiting each vertex in set S exactly once, starting at 1 & ending at i.

→ $O(n^2 * 2^n)$



$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

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

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

$g(4,\phi) = C_{41} = 5$

using (5,21) we obtain,

$g(2,\{3\}) = C_{23} + g(3,\phi) = 15$

$g(3,\{2\}) = 18$

$g(4,\{2\}) = 13$

$g(2,\{4\}) = 18$

$g(3,\{4\}) = 20$

$g(4,\{3\}) = 15$

$g(i,s)$ with $[s] = 2,\ i \neq 1,\ i \notin s$ & $1 \notin s$

$g(2,\{3,4\}) = \min[C_{23} + g(3,\{4\}),\ C_{21} + g(4,\{3\})] = 25$

$g(3,\{2,4\}) = \min[C_{32} + g(2,\{4\}),\ C_{24} + g(4,\{2\})] = 25$

$g(4,\{2,3\}) = \min[C_{42} + g(2,\{3\}),\ C_{23} + g(3,\{2\})] = 23$

Algorithm for Travelling sales person problem through dynamic programming.

Data:- S: Starting point, N: Subset of input cities,
dist(): distance among the cities.

Result:- cost: TSP result

```
visited[N] = 0;
cost = 0;

procedure TSP(N,S)
{   visited[S] = 1;
    if [N] = 2 and K ≠ S then
    {
        cost[N,K] = dist(S,K);
```

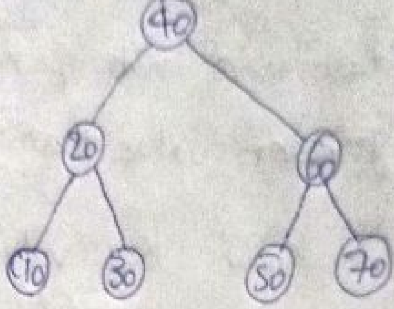
optimal Binary Search Tree:-

Binary search tree

- In BST, nodes in left subtree have lesser value than root node & nodes in right subtree have greater value than root node.
- we know key values (each node) in the tree & we know frequencies of each node in terms of searching means how much time is required to search a node.
- The frequency & key-value determine the overall cost of a searching a node.
- The cost of searching is a very important factor in various applications.
- cost of (searching a node) should be less. The time required to search a node is Binary Search tree is more than the balanced binary search tree as a balanced binary search tree contains a lesser no. of levels than the Binary search Tree.
- There is one way that can reduce the cost of a binary search tree is known as an optimal binary search tree.

Ex:-

If keys:- 10, 20, 30, 40, 50, 60, 70



The maximum time required to search a node is equal to the minimum height of the tree.

equal to logn.

To find 50

3 comparisons

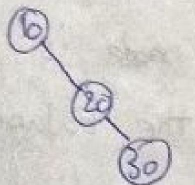
40 and 50

60 and 50

50 and 50

-> 2n-1 no. of trees can be formed for n keys.

Ex:- 10, 2, 30 are the keys.



Average number no. of comparisons

For (10) = 1

For (20) = 2

For (30) = 3

$\Rightarrow \frac{1+2+3}{3} = 2$

Average Avg. number of comparisons

For(10) | For(30) | For(20)

$\frac{1+2+3}{3} = 2$

Avg. Comparisons

For (10), for (30) = 2

For (20) = 1

$\frac{2+1+2}{3} = \frac{5}{3}$

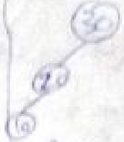
Avg. Comparisons

For (10) = 2

For (20) = 3

For (30) = 1

$\frac{2+3+1}{3} = 2$



Avg. Comparisons

For (10) = 3

For (20) = 2

For (30) = 1

$\frac{3+2+1}{3} = 2$

less time.

Ex:- we read about Height-balanced Search tree.

→ To find optimal Binary Search tree, OBST, we will determine the frequenly of Searching a Key.

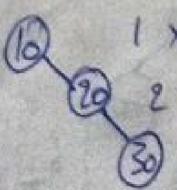
→ Let's assume that frequencies associated with the Keys 10, 20, 30 are 3, 2, 5.

→ The discussed trees have different frequencies.

→ The tree with the lowest frequenly would be considered the optimal binary search tree.

→ The tree with the frequenly 17 is the lowest, So it would be considered as the optimal binary Search tree.

For T_1



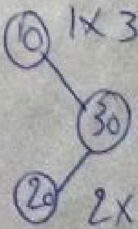
1×3

2×2

3×5

22

T_2



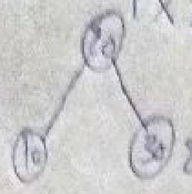
1×3

5×2

2×3

19

T_3



1×2

2×3 $\quad$ 2×5

$6 + 2 + 10 = 18$

T_4

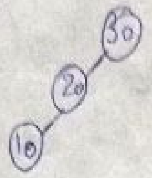
1×5

2×3

3×2

17

T_5



1×5

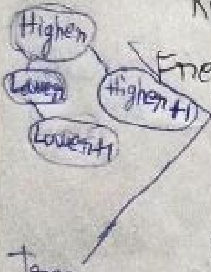
2×2

3×3

18

=)

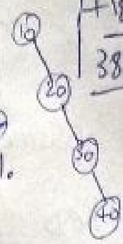
	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3



Keys
Frequency } $6 \times 1 + 2 \times 4 + 3 \times 2 + 2 \times 3$

$6 + 8 + 6 + 6 = 26$

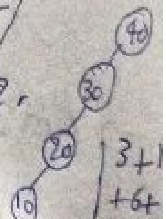
1.



$4 + 4 + 18 + 12$

38

2.



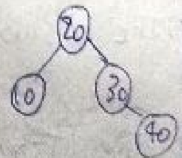
$3 + 1$
$+ 6 +$
37

7.

$2 + 12 + 12 + 16$

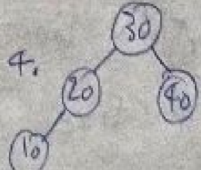
42

3.



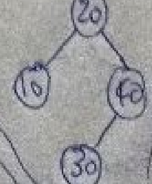
$2 + 8$
$+$
31

4.



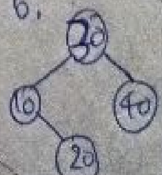
$6 + 4 +$
$+ 6$
34

5.



$2 + ~~8~~ + 6$
$+$
3

6.



$6 + 8 + 6 + 6$

26

=) Algorithm OBST(p,q,n)

```
// Given n distinct identifiers a1 < a2 < ... an &
// probabilities p[i], 1 ≤ i ≤ n, & q[i], 0 ≤ i ≤ n,
// this algorithm computes the cost[i,j] of optimal
// binary search trees tij for identifiers
// ai+1, ... aj, for identifiers it also computes
// r[i,j], the root of tij.
// w[i,j] is the weight of tij.
{
  for i := 0 to n-1 do
  {  // initialize,
     w[i,i] := q[i]; r[i,i] = 0; c[i,i] := 0.0;
     // optimal trees with one node
     w[i, i+1] := q[i] + q[i+1] + p[i+1];
     r[i, i+1] := i+1;
     c[i, i+1] := q[i] + q[i+1] + p[i+1];
  }
  w[n,n] := q[n]; r[n,n] = 0; c[n,n] = 0.0;
  for m := 2 to n do // Find optimal trees with m nodes.
    for i := 0 to n-m do
    {
       j := i+m;
       // solve 5.12 using Knuth's result;
       w[i,j] := w[i, j-1] + p[j] + q[j];

       k := Find(c, r, i, j);
       // A value of l in the range r[i,j-1] ≤ l
       // ≤ r[i+1,j] that minimizes c[i, l-1] + c[l,j];
```

```
        c[i,j] := w[i,j] + c[i,k-1] + c[k,j];
        r[i,j] := k;
    }
    write( c[0,n], w[0,n], r[0,n]);
}
Algorithm Find(c,r,i,j)
{
    min := ∞;
    for m := r[i,j-1] to r[i+1,j] do
        if (c[i,m-1] + c[m,j]) < min then
        {
            min := c[i,m-1] + c[m,j]; ~~l:=m;~~
            l := m;
        }
    return l;
}
```

Complexity is $O(n^3)$.

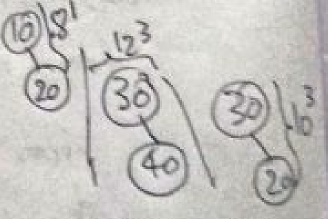
10 20 30 40
4 2 6 3

	0	1 (one nodes)	2 (2 nodes)	3 (3 nodes)	4 (4 nodes)
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3 (3 nodes)
2			0	6	12^3 (2 nodes)
3				0	3 (1 nodes)
4					0

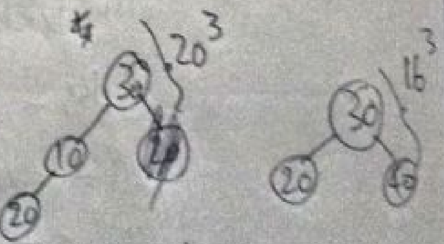
10^3 by 3-root cost is 12

10	4
20	2
30	6
40	3

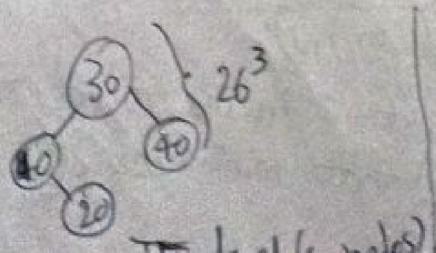
I level (1 node)



II level (2 nodes)



III level (3 nodes)



IV level (4 nodes)

→ Flow shop scheduling:-

→ The processing of a Job requires the performance of several distinct tasks.

→ Computer programs run in a multiprogramming environment are input & then executed.

→ Following the execution, the Job is queued for output & the output eventually printed.

→ In a general flow shop we may have n Job each requiring m tasks $T_{1i}, T_{2i}, \ldots, T_{mi}$, $1 \le i \le n$, to be performed.

→ A non-preemptive schedule is a schedule in which the processing of a task on any processor is not terminated until the task is complete.

→ A schedule for which this need not be true is called preemptive.

$$F(S) = \max_{1 \le i \le n} \{f_i(S)\}$$

The mean flow time MFT(S) is defined to be

$$MFT(S) = \frac{1}{n} \sum_{1 \le i \le n} f_i(S)$$

→ Task T_{ji} is to be performed on processor P_j, where $1 < j < m$.

→ The time required to complete Task T_{ji} is t_{ji}.

→ A schedule for the n Jobs is an

assignment of tasks to time intervals on the processors.

→ Task T_{ji} must be assigned to processor P_j. No processor may have more than 1 task assigned to it in any time interval. Additionally, for any job i the processing of task T_{ji}, $(j>1)$, can't be started until task $T_{j-1,i}$ has been completed.

Ex:-

2 Jobs scheduled on 3 processors.
The task times are given in matrix J.
2 possible schedules

$$J = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$