09/10/2020
FRIDAY

⇒ DYNAMIC PROGRAMMING:-

↳ Dynamic programming is a technique for solving problems with overlapping subproblems.

↳ These subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems.

Is for general Prbm solving and for more optimisation problems too.

⇒ PRINCIPLE OF OPTIMALITY:-

An optimal solution to any instance of an optimization problem is composed of optimal solutions to its sub instances.

PROBLEM
↙ ↘
Sub Prbm — Finding optimised soln for sub Prbm → Optimised soln for the main problem
Sub Prbm — Finding optimised soln ↗

Example:- ⇒ **Fibonacci Number:-**

↳ $F(n) = F(n-1) + F(n-2)$ for $n > 1$

↳ $F(0) = 0$, $F(1) = 1$, $F(2) = 1$, $F(3) = 2$ ... (Stored somewhere)

$F(2) = F(1) + F(0) = 1$

$F(3) = F(2) + F(1) = 2$, .. so on

(1) **Knapsack** Problem :- (maximization problem) (Its like a Bag.)

↳ Given 'n' items of known weights $W_1, .. W_n$ and values $V_1, .. V_n$ and a knapsack of capacity W, find the most valuable subset of the items to fit into the knapsack. (One kind of greedy approach)

↳ Assume here that all the weights and the knapsack capacity are positive integers and the item value do not have to be integers.

Also this is called 0/1 knapsack problem. Since, [either items should be taken into knapsack (1) or Fully Left outside (0)] No partial Fitting of items

Eg:- Consider a knapsack of capacity 4.

Weights	Items	value
2	Gold	100
3	Silver	30
1	Book	25
4	Waste paper	5

Say we have 4 items

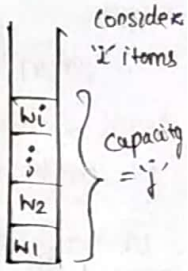
Say suppose if capacity is 7,
then if gold is taken, capacity becomes $7-2=5$ [since most valuable should be taken first]
After silver its $5-3=2$ and so, on..

F(i,j) → consider 'i' items out of first 'i' items finding out the most valuable subset which fits into knapsack of capacity 'j'.

Solution :-

↳ Consider an instance defined by the first 'i' items $1 \le i \le n$ (out of 'n' items choose 'i' items), with weights $w_1, \ldots w_i$ and values $v_1, \ldots v_i$ and a knapsack of capacity j, $1 \le j \le W$.

↳ Let $F(i,j)$ be the value of an optimal solution to this instance (i.e.) the value of the most valuable subset of the first 'i' items that fit into the knapsack of capacity 'j'.



↳ We can divide all the subsets of the first 'i' items that fit the knapsack of capacity 'j' into 2 categories.

(i) Those that do not include the 'i'th item [exclude i^{th} item] and

(ii) Those that do [may (or) may not include i^{th} item based on its value] → Based on these things we have to form a Recurrence Relation

↳ among the subsets that do not include i^{th} item, the value of an optimal subset is, by definition $F(i-1, j)$

↳ among the subsets that do include the i^{th} item (Hence, $j - w_i \ge 0$; if i^{th} item is not considered, then capacity = $j - w_i$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fit into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i-1, j-w_i)$.

Thus, [Here out of two, which has max value (i.e. most valuable is taken.]

$$F(i,j) = \begin{cases} \max\{F(i-1,j),\ v_i + F(i-1, j-w_i)\}, & \text{if } (w_i \le j) \quad j - w_i \ge 0 \\ F(i-1,j), & \text{if } j - w_i < 0 \end{cases}$$

For (ii) ↳ $j - w_i < 0$ → 'i' th element is not included so, there's no need for any space.

For (i) ↳ $(w_i > j)$

Conditions:-

$j - w_i \ge 0$

In a knapsack of capacity 'j', now 'i-1' items are filled in it. When filling the i^{th} item, $j - w_i \ge 0$ since we should have space to fill the i^{th} item

⇒ 2 APPROACHES:-

↳ TOP DOWN and

↳ Bottom up approach

Example:-

Item	Weight / Value
1	$w_1 = 2$; $12 - v_1
2	$w_2 = 1$; $10 - v_2
3	$w_3 = 3$; $20 - v_3
4	$w_4 = 2$; $15 - v_4

Capacity $N = 5$

Items, capacity.

Goal: To find $F(4,5)$

① BUA → Start from end. subproblems and move towards goal

② TDA → Start from goal and move toward the subproblems (individually).

Optimal ⑤ Subset – most valuable subset

NKT, for any recurrence relation we need and have initial condition.

↳ If items are there but there is no space (capacity is zero) then, $F(i,0) = 0$

↳ If space is there but there are no items (items = 0) then in that case too it's zero (ie) $F(0,j) = 0$.

Bottom up ⇒ -approach

capacity $j \rightarrow$

$i \downarrow$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37 → Goal $F(4,5)$

$F(1,2)$ → 1 item (1st item) and capacity 2.

$j - w_i = 2 - w_i = 0$

Whenever $j - w_i \geq 0$, we can include i^{th} item.

So, $F(1,2) = \max[F(0,2), F(0,0) + 12]$

$= \max[0, 0+12] = \max[0,12]$

$= 12$.

$F(1,3)$ ($i=1, j=3$) → 1 item and capacity 3.

$j - w_i = 3 - 2 \geq 0$

$\Rightarrow F(1,3) = \max [F(0,3), F(0,1)+12]$

$= \max [0, 12] = 12.$

$F(1,4)$ ($i=0, j=4$) → 1 item and capacity 4

$j - w_i = 4 - 2 = 2 \geq 0$

$\Rightarrow F(1,4) = \max [F(0,4), F(0,2)+12]$

$= \max [0, 12] = 12.$

$F(1,5)$ → 1 item and capacity 5

$j - w_i = 5 - 2 = 3 \geq 0$

$\Rightarrow F(1,5) = \max [F(0,5), F(0,3)+12$

$F(2,1)$ ($i=1, j=1$) → (1st 2 items) 2 items and capacity 1

$j - w_i = 1 - 1 = 0$

$\Rightarrow F(2,1) = \max [F(1,1), F(1,0)+10]$

$= \max [0, 10] = 10$

$F(2,2)$ ($i=1, j=2$) → 2 items and capacity 2

$j - w_i = 2 - 1 \geq 0$

$= 1$

$\Rightarrow F(2,2) = \max [F(1,2), F(1,1)+10]$ ($i=1, j=2$)

$= \max [12, 10] = 12$

$F(2,3)$ → 2 items and capacity 3

$j - w_i = 3 - 1 = 2 \geq 0$

$\Rightarrow F(2,3) = \max [F(1,3), F(1,2)+10]$

$= \max [12, 22] = 22$

$F(2,4)$ ($i=1, j=4$) → 2 items and capacity 4

$= \max [12, 22]$

$j - w_i = 4 - 1 = 3 \geq 0$

$\Rightarrow F(2,4) = \max [F(1,4), F(1,3)+10] = 22$

$j-w_i = 4$
≥ 0

$F(2,5) = \max[F(1,5), F(1,4)+10]$
$= \max[12, 22] = 22$

$j-w_i = -2$
< 0

$F(3,1) = F(2,1) = 10$

$j-w_i = -1$
< 0

$F(3,2) = F(2,2) = 12$

$j-w_i = 0$
≥ 0

$F(3,3) = \max[F(2,3), F(2,0)+20]$
$= \max[22, 20] = 22$

$j-w_i = 1$
≥ 0

$F(3,4) = \max[F(2,4), F(2,1)+20]$
$= \max[22, 30] = 30$

$j-w_i = 2$
≥ 0

$F(3,5) = \max[F(2,5), F(2,2)+20]$
$= \max[22, 32] = 32$

$j-w_i = -1$
< 0

$F(4,1) = F(3,1) = 10$

$j-w_i = 0$
≥ 0

$F(4,2) = \max[F(3,2), F(3,0)+15]$
$= \max[12, 15] = 15$

$j-w_i = 1$
≥ 0

$F(4,3) = \max[F(3,3), F(3,1)+15]$
$= \max[22, 10+15] = 25$

$j-w_i = 2$
≥ 0

$F(4,4) = \max[F(3,4), F(3,2)+15]$
$= \max[30, 27] = 30$

$j-w_i = 3$
≥ 0

$F(4,5) = \max[F(3,5), F(3,3)+15]$
$= \max[32, 37] = 37$ //

Now we only know that the value of the most optimal subset is 37. But what are the subsets used to fill the knapsack? What values are combined to that this value is computed?

Always compare a value to a value above it

Say, consider [37 / 37] [checking whether 4th item is included / not] Suppose if both are same, go with 1st 3 items and with first 4 items, in both the cases, we got a value of 37.

Say the knapsack capacity = 5

So, 4th is neglected and 3rd is added to knapsack.

In this eg, 1st 4 items are taken. [37 / 21] max. If different, 4th item is included to fill the knapsack.

So, now capacity = $5 - w_4 = 5 - 2 = 3$.

Now, to check whether 3rd item is included or not.

→ F(3,3) = 22 (we have)

[22 / 22] same value. So, 3rd item is neglected and 2nd item is considered.

F(2,3) = 22

[12 / (22)] max → 2nd item is considered and included to ~~fill~~ the knapsack.

Now capacity = $3 - w_2 = 3 - 1 = 2$

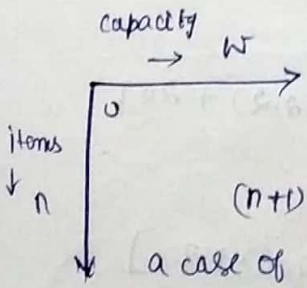
To check whether 1st item is included or not

F(1,2) = 12 (refer next page – Top → down approach for exp.)

[0 / (12)] max → 1st item is included to fill the knapsack

Now capacity = $2 - w_1 = 2 - 2 = 0$ // Hence Full

Complexity ⇒



If we have n items then we have (n+1) rows and (w+1) columns [Since we consider a case of 0 items and a case of capacity 0 Respectively]

In total $(n+1)(w+1) = O(nw)$

[∵ we have to fill each and every row and column]

Problems here is:-

↳ We are not satisfying all the values. Just a few values only are verified. So waste of time

Hence more for Top down approach

Top down approach ⇒

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

Memory function:- While performing a function recursively, it stores some value in the memory it's called a memory function.

virtual initialization :-

Initializing all other values other than the row and column 0 :: since WKT its all value is zero) to -1.

[F(i,j) is
i → 1st i items
j - capacity]

Consider

$F(4,5) = \max[F(3,5), F(3,3)+15]$

$j - w_i = 5-2 = 3 \geq 0$

$F(3,5) = \max[F(2,5), F(2,2)+20]$

$j - w_i = 5-3 = 2 \geq 0$

$F(3,3) = \max[F(2,3), F(2,0)+20]$

$j - w_i = 3-3 = 0 \geq 0$

$F(2,5) = \max[F(1,5), F(1,4)+10]$

$j-w_i = 5-1 = 4 \geq 0$

$F(2,2) = \max[F(1,2), F(1,1)+10]$

$j-w_i = 2-1 = 1 \geq 0$

$F(2,3) = \max[F(1,3), F(1,2)+10]$

$j-w_i = 3-1 = 2 \geq 0$

$F(2,0) = 0$

$F(1,5) = \max[F(0,5), F(0,3)+12]$

$= \max[0,12] = 12$

$j-w_i = 5-2 = 3 \geq 0$

$F(1,4) = \max[F(0,4), F(0,2)+12]$

$= \max[0,12] = 12$

$j-w_i = 4-2 = 2 \geq 0$

$F(1,2) = \max[F(0,2), F(0,0)+12] = 12$

$j-w_i = 2-2 = 0 \geq 0$

$F(1,3) = \max[F(0,3), F(0,1)+12] = 12$

$j-w_i = 3-2 = 1 \geq 0$ and $F(1,1) = F(0,1) = 0$.

$j-w_i = 1-2 \leq 0$

Thus $F(2,3) = \max(12, 22) = 22$

$F(2,2) = \max[12, 10] = 12$

$F(2,5) = \max[12, 22] = 22$

$F(3,3) = \max[22, 20] = 22$

$F(3,5) = \max[22, 32] = 32$

and $F(4,5) = \max[32, 37] = 37$. Hence Found.

(most optimal subset)

OUR GOAL : $F(4,5)$ → With 4 items what is the optimal value to fill the capacity of 5.

~~Consider F(4,5) → 1st 4 items with capacity 5~~ (i, j)

~~considering the maximum value, which is 37~~

To find whether 4th item is included or not

⟹ So we have F(4,5) = 37.

↳ comparing with the value above (32)

↳ 37 is greater than 32

↳ Hence 4th item (with weight 2) was included to fill the knapsack

↳ Now capacity of knapsack is 5−2 = 3//

To find :- Whether 3rd item is included or not.

F(3,3) = 22

↳ comparing with the above value (22)

↳ values are same, no need for 3rd item to be included

↳ So, 2nd item with weight 1 was included to fill the knapsack.

↳ So, capacity of knapsack = 3−1 = 2.

To find :- Whether 1st item is included or not.

F(1,2) = 12.

↳ comparing with the value above (0)

↳ 12 > 0 hence, 1st item was included to fill the knapsack (weight = 2)

↳ Capacity of knapsack = 2−2 = 0. Hence knapsack is FULL.

So the 1st, 2nd and 4th item are used to fill the knapsack and the most optimal subset is F(4,5) with value 37.

Algorithm ⇒

MFKnapsack (i, j) // F[i, j]

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

// I/P: A non-negative integer 'i' indicating a
// number of the first 'i' items being considered
// and a non-negative integer 'j' indicating
// the capacity of knapsack.

// O/P :- The value of optimal feasible subset
of the first 'i' items.

// Note :- Uses as global variables input arrays
// Weights [1..n], values [1..n], and table F[0..n, 0..W]
// whose entries are initialized with -1's except
// for row 0 and column 0 whose values are
// initialized with 0 ⇐ Virtual Initialization

~~At last, goal's value~~

if F[i, j] < 0 j < Wi

if j < Weights [i] // (ith item excluded)

value ← MFKnapsack (i-1, j).

else // (ith item may or, may not be included)

j > Wi

value ← max (MFKnapsack (i-1, j),

values [i] + MFKnapsack (i-1,

j - Weights [i]))

F[i, j] ← value.

return F[i, j].

Initially i = n and j = W

Analysis ⇒

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

Here maximum, only ½ of the table values are computed. Remaing is left Blanck.

So, $\frac{nW}{2} = O(nW)$ → same as Bottom up approach.

[Although there are only ½ of values, complexity Remains the same]

II

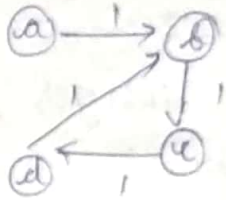
⇒ TRANSITIVE CLOSURE :- (Binary matrix)

↳ FOR ONLY directed graph.

Reach-ability matrix

Ability (ie) to travel (have a path) from one vertex to another one

Consider a directed graph



a to c → Path length = 2 (weights on edges)

a to d → path length = 3.

[If there's an edge bet. 2 vertices)
↳ Able to travel - 1

[Not Able to travel - 0
(Else)

Here (Not bothered about the Length of path)
⇒

For 'n' vertices, in a graph, its transitive close is a n x n matrix (Binary).

Eg:

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

PROBLEM :-

Given an adjacency matrix Represented as R_0 (actually A)

FROM R_0, To find :- $R_1, R_2 .. R_n$ where n - no. of vertices

↳ R_0 - a path matrix with intermediate vertices whose number is ≤ 0 (not exceeds 0) or with 0 intermediate vertex.

↳ R_1 - a path matrix with intermediate vertices whose number is ≤ 1 (or) with zero intermediate vertex and so on, ...

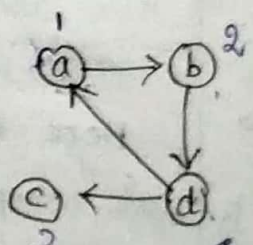
⇒ Qn :-

Given A = (adjacency matrix)

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

and

GRAPH



In R_0 (ie) Matrix A given, only 'a' vertex can be an intermediate vertex.

Warshall's Algorithm ⇒

I

The transitive closure of a directed graph with 'n' vertices can be defined as an n×n boolean matrix $T = \{t_{ij}\}$ in which the element in the i^{th} row and the j^{th} column is 1 if there exists a non-trivial path (i.e directed path of a +ve length) from i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is 0.

II ⇒ Soln:-

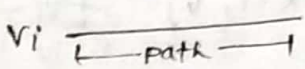
$$R_1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

$$R_2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$$R_3 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

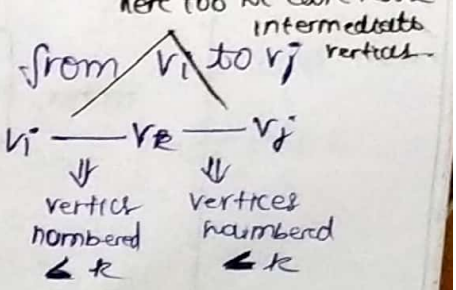
$$R_4 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Since n = 4, //

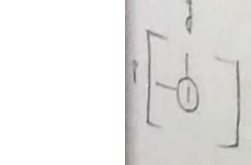


Let R^k be some path matrix with intermediate vertices numbered $\leq k$

Just to understand ⇒ [(ii) When we have a path from v_i to v_j with v_k as intermediate vertex and if $r^{k-1}_{ik} = 1$ and $r^{k-1}_{kj} = 1$ then $r^k_{ij} = 1$ //.



That is if there is path from v_i to v_k and from v_k to v_j in R^{k-1} then there will be a path from v_i to v_j in R^k.



(3) If the path has no intermediate vertices, (v_k) and if $r_{ij}^{k-1} = 1$ then $r_{ij}^{k} = 1$.

↳ Warshall's algorithm constructs the transitive closure through a series of n×n Boolean matrices:

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

↳ Let $r_{ij}^{(k)}$, the element in the i^{th} row and j^{th} column of matrix $R(k)$ be equal to 1.

[Path → is a sequence of vertices]

~~↳ Warshall's algorithm constructs the transitive closure through a series of n×n boolean matrices~~

① ↳ v_i, a list of intermediate vertices each numbered not higher than k, v_j $[v_i \text{——} v_j]$

② ↳ v_i, vertices numbered $\leq k-1$, v_k, vertices numbered $\leq k-1$, v_j $[v_i \text{—} (v_k) \text{—} v_j]$

$r_{ik}^{(k-1)} = 1$

$r_{kj}^{(k-1)} = 1.$

Hence, $\boxed{r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } (r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}).}$

Algorithm ⇒

```
Warshall (A[1..n, 1..n])
// Implements Warshalls Algorithm for
// computing the transitive closure
```

(1) In short, r_{ij}^{k} is 1 iff $r_{ij}^{k-1} = 1$ (There is a direct Edge between the starting (v_i) and Ending (v_j) vertex)

(2) $r_{ij}^{k} = 1$ iff $r_{ik}^{k-1} = r_{kj}^{k-1} = 1$ (If any intermediate vertex k is between v_i and v_j).

(*) ↑ HINT

// I/P : The Adjacency matrix A of a digraph with n vertices.

// O/P : The transitive closure of digraph

$R^{(0)} \leftarrow A$ // Adjacency matrix

for $k \leftarrow 1$ // no. of repetitions = no. of vertices to n do

[for $i \leftarrow 1$ to n do
  for $j \leftarrow 1$ to n do] // To find for each and every cell in the matrix

$R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ OR ($R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j]$)

return $R^{(n)}$.

16/10/2020 FRIDAY

Complexity ⇒

$$\sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} c = O(n^3)$$

FLOYDS ALGORITHM :- → (All pairs shortest path algorithm)

In Warshall's, if there a path between v_i and v_j, the value = 1.

But in Floyds algorithm, if there a path, the distance between v_i and v_j is found and saved in matrix

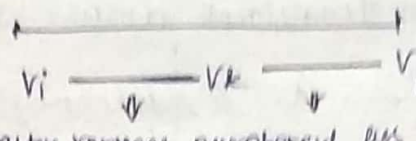
↳ Floyds algorithm computes the distance matrix of a weighted graph with 'n' vertices through a series of n×n matrices :

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

NOTE

[Here the distance bet. a and a, b and b and so on is always zero]

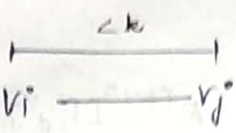
(ii) If to find distance between v_i and v_j with an intermediate vertex v_k, Distance $d_{ij}^{k} = d_{ik}^{k-1} + d_{kj}^{k-1}$.



[may have vertices numbered less than k]

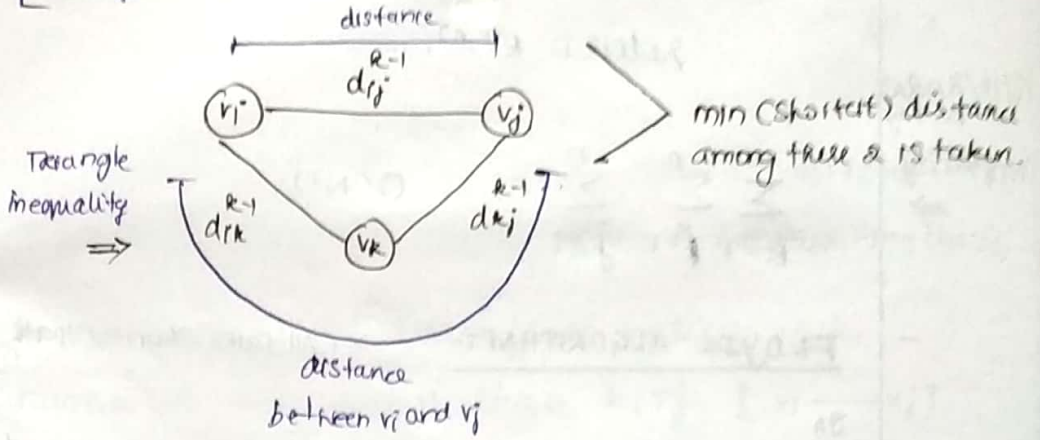
D^k → Distance matrix having intermediate vertices numbered $\leq k$.

(i) No intermediate vertex, but still there is a path

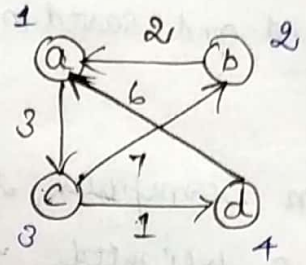


Distance $d_{ij}^{k} = d_{ij}^{k-1}$

[To find shortest path choose the min among these two]



⇒ **Example:-**



Initial matrix

$$D_0 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

NOTE

[In Warshall's, when there's a path, value = 1

Here, for the vertex having distance as zero (distance from a vertex to itself) then value = 0. When there's no path, value = ∞ (since dont know the actual distance)].

$$D_1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array} \qquad D_2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

D_3 =

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

D_4 =

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

Note: [In case, If we have 2 paths between v_i and v_j (say) for example here c to a we have c-b-a and c-d-a. Here the path having min distance is only considered]

Algorithm ⇒

FLoyd [W[1..n, 1..n])

// Implements Floyd's algorithm for all-pair shortest paths problem

// I/P: The weight matrix W of a graph with no negative-length cycle.

// O/P: The distance matrix of the shortest paths lengths (a distance matrix)

D ← W // not necessary if W can be overwritten

for k ← 1 to n do // no. of times to repeat the procedure

  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]]

(D[i, j] ← direct path; D[i, k] + D[k, j] ← path with intermediate vertex (v_k))

return D.

use during implementation [if i == j, ⇩ Then D_{ij} = 0]

20/10/2022

BACK-TRACKING:- (Only for non-optimization problems)

Given a problem and some constraints, Find a soln that satisfies these constraints.

↳ **Idea:-** Construct solutions one component at a time and evaluate such partially constructed candidates as follows.

How to Find soln? ⟹ *

Construct Partial soln which satisfies these constraints [one component at a time]

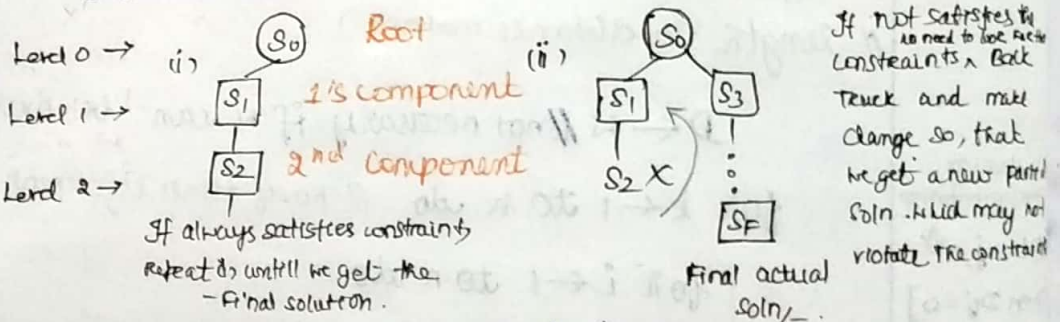
↳ (i) If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the 1st remaining legitimate option for the next component. // Repeat above process if not satisfied go to (ii) below.

Evaluate these partially constructed solutions. ⟹

↳ (iii) If there is no legitimate option, for the next component, no alternatives for any remaining component need to be considered.

(ii) Backtracking - going back to previous stages, make some changes to the partial soln and now check whether we can proceed further. If so, then continue else, repeat the same process.

↳ In this case, the algorithm backtracks (going back to previous stages) to replace the last component of the partially constructed soln. with its next option.



NOTE

[Root (S_0) is the Initial cond.]

↳ **State-space tree:-** (tree of choices)

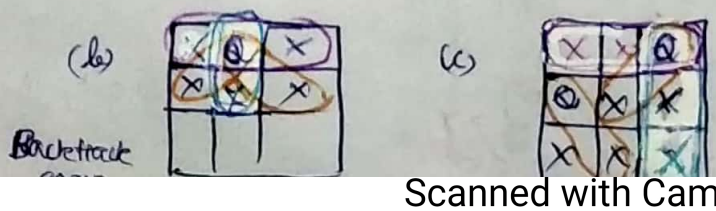
↳ Its root represents an initial state before the search for a soln. begins.

↳ The nodes of the first level in the tree represent the choices made for the first component of a solution

↳ The nodes of the 2nd level represent the choices for the second component and so on;

[$S_1, S_2 \ldots S_F$ are collectively called "state space tree"]

Starting from root constructing component (P. Soln)



[Leaves that still have scope for extending the soln and may lead to find the final soln]

else non-promising leaves

↳ Promising :-

⇒ A node in a state space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution.

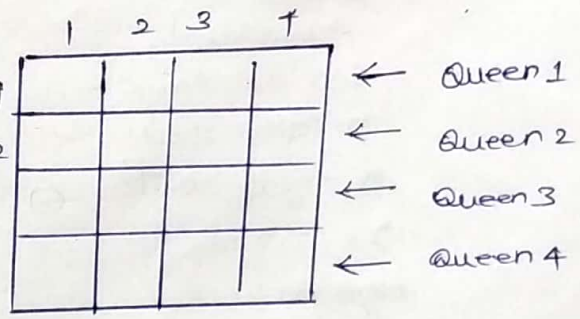
Leaves represent either non-promising dead ends (or) complete solutions found by the algorithm.

⇒ N-QUEEN PROBLEM :-

[Given NxN chessboard and 'N' queens]

→ The problem is to place 'n' queens on an n x n chessboard so that no 2 queens attack each other by being in same row or, same column (or) in same diagonal.

Eg :- [4-Queens - Problem]



∴ For 4 Queen - problem [SOLUTION ↓ - EXISTS]

(Refer next page)

↳ Do we have solution for 1 Queen problem? ✓

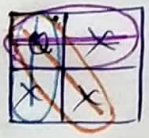
yes [since there is only one box]

[Backtracking not only means going back to the last previous partial soln, it may be any of the previous one]

↳ For 2 Queen problem ? NO

HINT

[Same row
Same column
Diagonal]

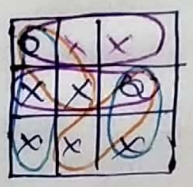


← 1st Queen

can't insert 2nd Queen

↳ For 3 Queen problem ? NO.

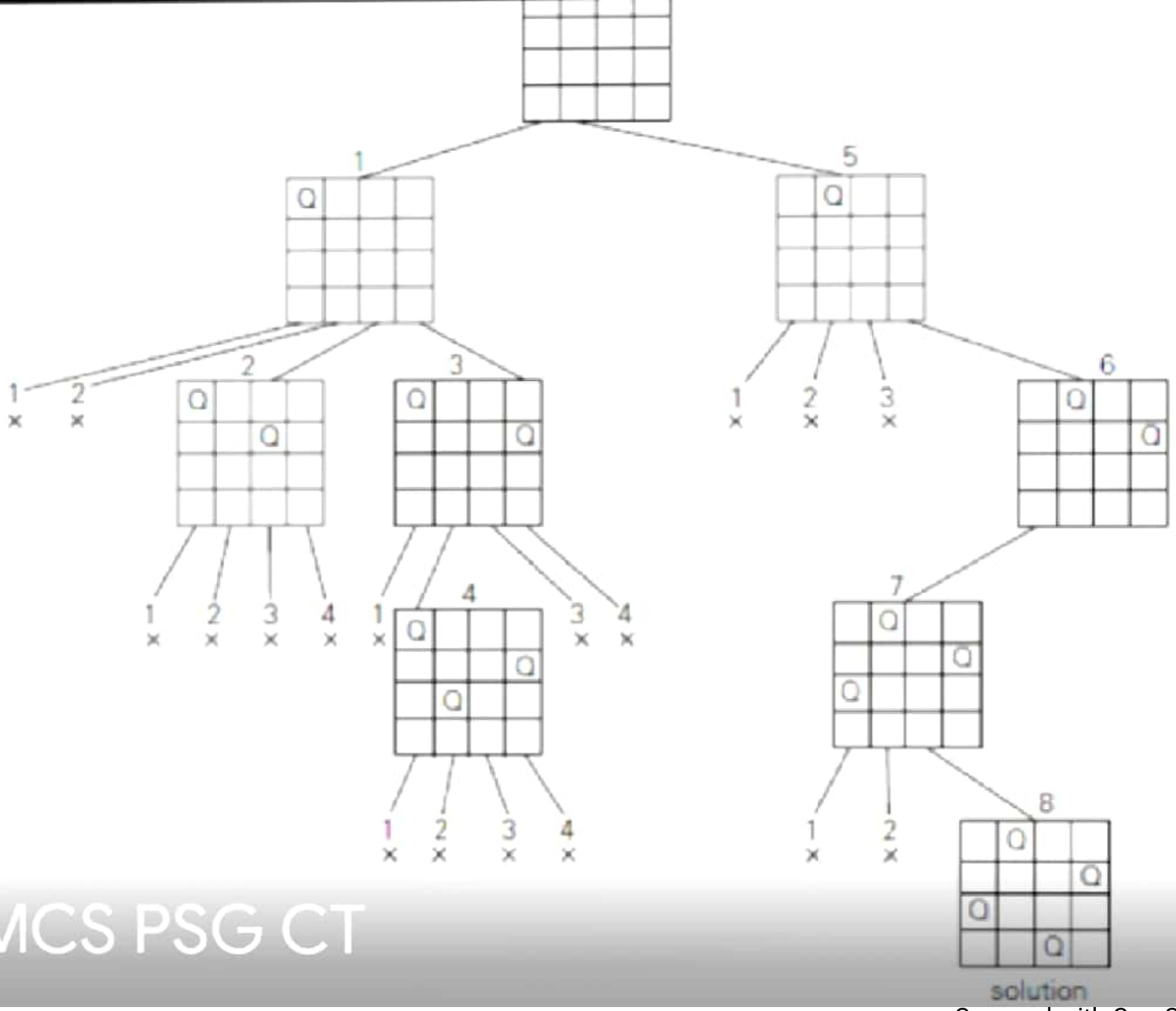
(a)



← 1st Queen

← 2nd Queen

[We can't change the position of 2nd Queen so Backtrack to 1st Queen and change its position and try on].



x-ysvb-djz

Hamiltonian Circuit Problem

- Start from a vertex, visit all other vertices exactly once and return back to the same vertex.
- Starting vertex will be visited twice.