# Unit-3

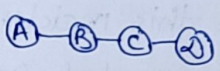
## Chapter-7 Graph Algorithm

### # Graphs

- Graph is a representation of set of objects which are connected by links.
- The objects which are interconnected are called as vertices and the links that connect them are edges.

G(V,E) = Set of Vertices and edges.

- Adjacency – Two vertices are adjacent if they are connected to each other through an edge.
- Path – It represents a sequence of edges between two vertices
  Like A-D path represented by ABCD




### # Basic Operations

- Add Vertex – to a graph
- Add edge – btw two vertices
- Display Vertex – of a graph

### # Representation of Graph;

A Graph representation is a method to store graph into the memory of computer, which requires set of vertices and edges.

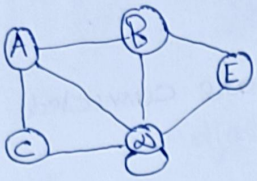
- If graph is weighted then wt. will be associated with each edge.
- Diff. ways are there to represent a graph, depending upon the density of edges, type of operations to be performed.

#### 1) Adjacency Matrix:

- It is a sequential representation of a graph.
- Represents adjacent vertices.
- In adjacency matrix representation, we have mxn matrix. if there exist an edge b/w vertex i to vertex j then element $A_{i,j} = 1$ otherwise '0'. (in case of un weighted graph)

- If graph is weighted then weight wt. instead of 1's & 0's.

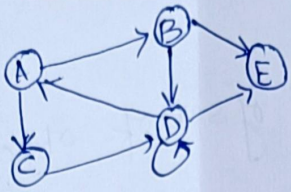
Undirected Graph




⇓

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 0 |
| B | 1 | 0 | 0 | 1 | 1 |
| C | 1 | 0 | 0 | 1 | 0 |
| D | 1 | 1 | 1 | 1 | 1 |
| E | 0 | 1 | 0 | 1 | 0 |

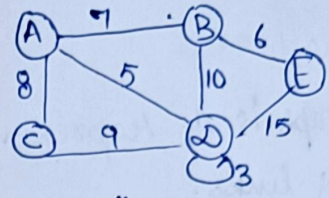
Directed graph




⇓

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 0 |
| B | 0 | 0 | 0 | 1 | 1 |
| C | 0 | 0 | 0 | 1 | 0 |
| D | 1 | 0 | 0 | 1 | 1 |
| E | 0 | 0 | 0 | 0 | 0 |

Undirected Weighted




⇓

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 7 | 8 | 5 | 0 |
| B | 7 | 0 | 0 | 10 | 6 |
| C | 8 | 0 | 0 | 9 | 0 |
| D | 5 | 10 | 9 | 3 | 15 |
| E | 0 | 6 | 0 | 15 | 0 |

2) Incidence matrix:

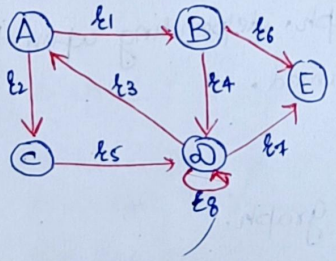
In this incidence matrix representation, a graph can be represented as; total no. of vertices by total no. of edges.

Ex; If Graph has 5 vertices and 8 edges, then it can be represented as 5×8 matrix.

where, columns represent edges & Row represents vertices.

This matrix is filled with 0 or 1 or -1;

- 0 - row edge which is not connected to column vertex
- 1 - row edge which is connected as outgoing edge to clm. vertex
- -1 - row edge which is connected as incoming edge to clm. vertex.

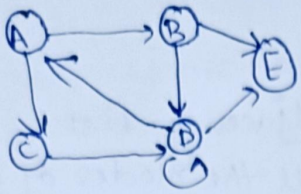


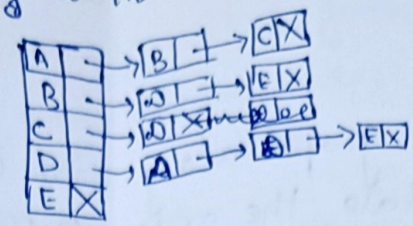

⇒

| | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ | $E_6$ | $E_7$ | $E_8$ |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 1 | -1 | 0 | 0 | 0 | 0 | 0 |
| B | -1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| C | 0 | -1 | 0 | 0 | 1 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | -1 | -1 | 0 | 1 | 1 |
| E | 0 | 0 | 0 | 0 | 0 | -1 | -1 | 0 |

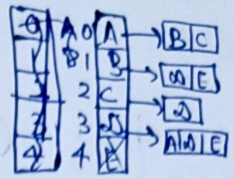
- Adjacency list:
  - Its a linked representation of graph.
  - For each vertex we maintain the list of its neighbours. ie every vertex of a graph contains list of its adjacent ~~matrix~~. vertex.

• we can represent the graph in two ways either linked list or array.





or



## Adjacency matrix

- Adv Representation is easier to implement.
- Dis Takes lot of space & time to visit all neighbour of a vertex.

## Adjacency list

- Adv Saves time
- Easy insertion & deletion as linked list is used.
- Adjacent nodes are clearly visible

# Graph Traversal Algorithm ; By which, we can traverse all the vertices of a graph.

• Two standard algo. of traversing graph are DFS and BFS.

### 1) DFS

- Algo start with a initial node of a graph 'G' and then
- Goes downwards until we find the desired node or the node having no child.
- Then backtrack from dead end towards the most recent unvisited node.
- Data structure used in DFS is stack.
- The edges that leads to an unvisited node are Discovery Edges.
- While the edges that leads to already visited nodes are Block edges.

Algo :- Step 1 : Set Status = 1 i.e ready state for each node in 'G'

Step 2 : Push starting node in stack and set Status = 2 i.e waiting state.

Step 3 : Pop the top element from stack then process it (print it) and set its Status = 3 ie Processed state

Step 4 : Push all the adjacent node of that processed elem into the stack. and update their Status = 2 (waiting)

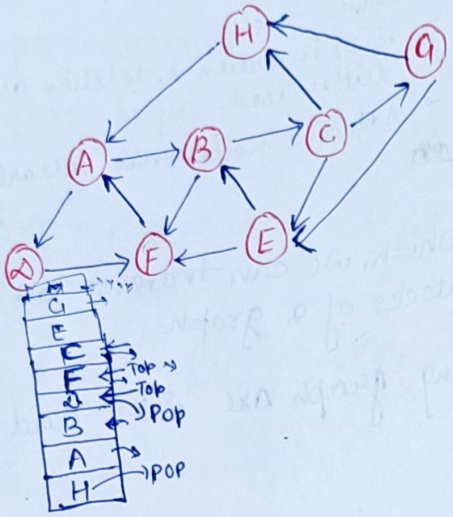
Step5: Repeat step 3 and 4 until Stack is empty.

Step6: Exit

Example;, Given graph along with its adjacency list. calculate the order to print all the nodes of graph starting from H, using DFS

Adjacency list

A – B, D
B – C, F
C – E, G, H
D – F
E – B, F
F – A
G – E, H
H – A




Print: ↗Top in stack.
H-A-D-F-B-C-G-E

BFS HABDCFEG

① Stack: H
Pop H and Push all adjacent in Stack

② Stack: A
Print: H
Pop A & add adj of A

③ Stack: B, D
Print: H, A
pop top ele. in Stack i.e D & add adj.

④ Stack: B, F
Print: H, A, D

⑤ Stack: B,
Print: H, A, D, F

⑥ Stack: C
Print: H, A, D, F, B

⑦ Stack: E, G,
Print: H, A, D, F, B, C

⑧ Stack: E
Print: H, A, D, F, B, C, G

⑨ Print: H A D F B C, G, E

★ The Stack is empty now and all the nodes of graph have been traversed.

## ② BFS

- BFS is a graph traversal algo. start from root node
- then explore all the neighbouring nodes and add all the nodes and ensure that each node is visited exactly once.

**Algo:** Step1: Set Status = 1 i.e ready for each node in G.

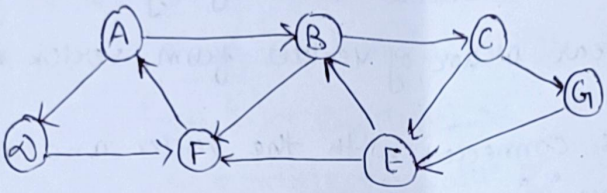
Step2: Enqueue the starting node and set status = 2 (waiting)

Step3: Dequeue a node 'N'. Process it and set its status = 3 (processed)

Step4: Enqueue all the neighbours of that node 'N' and set their status as 2 i.e waiting.

Step 5: Repeat step 3 & 4 until Queue is empty.

Step 6: Exit.




Adj List

A - B, D
B - C, F
C - E, G
D - F
E - B, F
F - A
G - E

↗ Consider a graph 'G' ↑.
Calculate minimum path 'p' from (A) to (F).
Given that each edge has length of 1.

⇒ find min path from A to E using BFS we required 2 Queues.
Queue 1 :- that holds all the nodes that are to be processed
Queue 2 : holds nodes that are processed and deleted from Queue 1.

(1) Add A to $Q_1$ and NULL to $Q_2$
$Q_1 = \{A\}$
$Q_2 = \{NULL\}$

(2) Delete node A from $Q_1$ and insert all neighbou of 'A' to $Q_1$
$Q_1 = \{B, D\}$ (Front: B, Rear: D) & insert A to $Q_2$
$Q_2 = \{A\}$

(3) Delete B from $Q_1$ & insert to $Q_2$ and also insert all n.g of B to $Q_1$
$Q_1 = \{D, C, F\}$
$Q_2 = \{A, B\}$

(4) $Q_1 = \{C, F\}$
$Q_2 = \{A, B, D\}$

(5) $Q_1 = \{F, E, G\}$
$Q_2 = \{A, B, D, C\}$

(6) $Q_1 = \{E, G\}$
$Q_2 = \{A, B, D, C, F\}$

(7) $Q_1 = \{G\}$ Ans — exhausted
$Q_2 = \{A, B, D, C, F, G\}$

(8) $Q_1 = NULL$
$Q_2 = \{A, B, D, C, F, E, G\}$

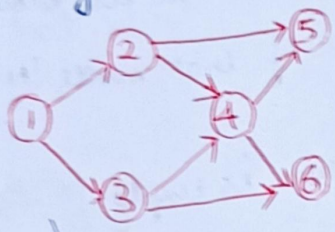
## Topological Sort:

Also called topological ordering.

- Topological sorting only possible if the graph is a Directed Acyclic graph.
- Every DAG may have multiple Topological Orders.
- It is a linear order of the vertices in such a way that if there exist an edge going from vertex u to v, then "u" comes before "v" in order.

Vertex to be visited first or task to be performed before another ⇒ Valid sequence of task

Find "in" no. of topological order for given graph.

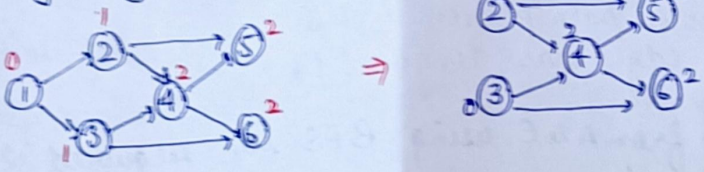



Case 1 | 1 2 3 4 5 6
Case 2 | 1 2 3 4 6 5
Case 3 | 1 3 2 4 5 6
Case 4 | 1 3 2 4 6 5

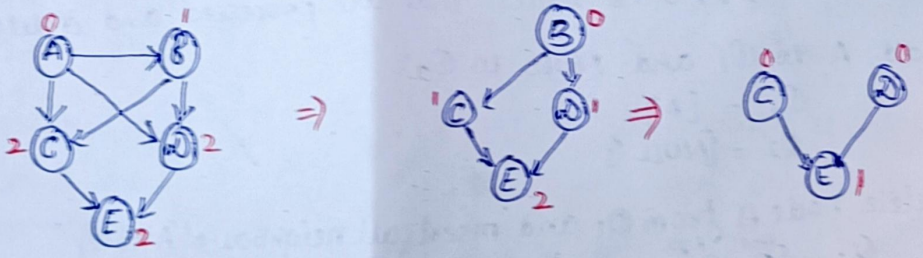
Step 1: Find in degree for all vertices ie no. of edges coming to a vertex.

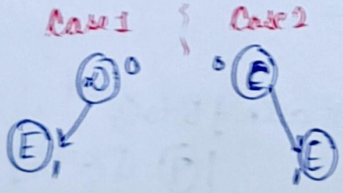
Step 2: Start writing linear order of vertices from vertex having in order = 0

Step 3: Remove the edges connected with the vertex and update the in degree of remaining graph.




Example 2:




→ A B C D E OR
→ A B D C E E

# Single ~~pair~~ source Shortest path; solved by dijkstra & Bellman ford algo.

① Dijkstra Algo:→ it salves SSSP problem on a directed graph i.e $G_1 = (V, E)$, where all edges are non-negative. (i.e $w(u,v) \geq 0$) ~~$\forall (u,v) \in E$~~

(Greedy algo)

: Extract-Min() function is used here, which extracts the node with smallest key.

≈ In sigle source shortest path, one source vertex is given and from that vertex we have to find all the shortest paths to other verteces.

Steps :-

1- Initially dist. of source to itself is 0 and dist. of all other vertices are ∞.

2- Consider 'u' as source (∵ smallest dist among all vertex.) and some adjacent vertex of 'u' as 'v' (i.e destination) and find their dist. of 'v' from 'u' using formula;

This is Relaxation ⇒ if (dist.(u) + Cost(u,v) < d(v))

then update the value of dist(v) as; d(v) = d(u) + c(u,v)

3- Next, check all the adjdcent vertices of 'u' and follow step 2 to ~~solve~~ find their dist. from 'u'

4- In next step make the source 'u' vertex as visited vertex.

#note once a vertex is visited we can't update its distance.

5- Next check the smallest distance among all remaining vertices and repeat step 2-4 untill we get all vertices as visited vertices.

# Drawback; dijkstra may or may not give correct results if wt's are -ve. ~~in a DAG~~ but if graph is cyclic and wt. -ve we can't ~~But diffin~~ apply dijkstra. algo. in that case.

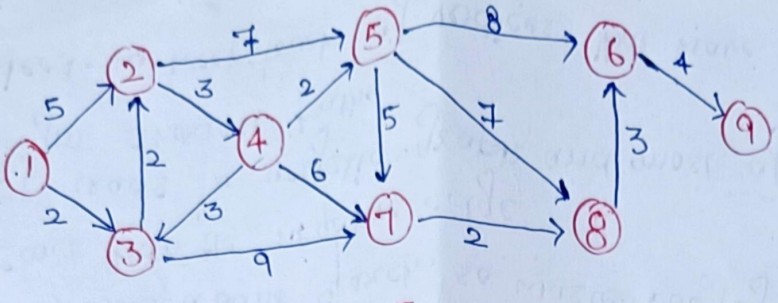
Disadv; 1) Does a blind search, so waste lot of time while processing

2) Can't handle negative edge

3) It leads to acyclic graph and most often can't obtain right shortest path.

4) Need to keep track of vertices that have been visited.

Ex:-




Source node = 'A'

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | [0] ← | ∞ | ∞ ← change | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 3 | | 5 | [2] ← | ∞ | ∞ | ∞ | ∞ ← change | ∞ | ∞ |
| 2 | | [4] | | ∞ | ∞ | ∞ | 11 ← | ∞ | ∞ |
| 4 | | | | [7] | 11 | ∞ | 11 ← | ∞ | ∞ |
| 5 | | | | | [9] | ∞ | 11 ← | ∞ | ∞ |
| 7 | | | | | | 17 | [11] ← | 16 ← change | ∞ |
| 8 | | | | | | 17 ← change | | [13] ← | ∞ |
| 6 | | | | | | [16] ← | | | ∞ ← change |
| 9 | | | | | | | | | [20] ← |

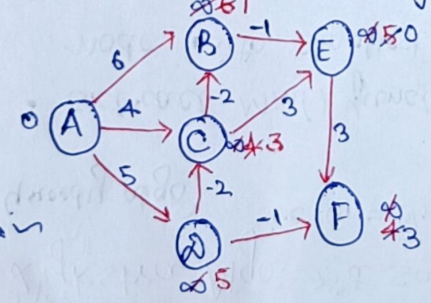
1-3-7-8-6-9

## #) Bellman Ford Algo;

- Solves SSSP problem of a directed graph in which the weight may be negative, without forming a cycle.
- Also used to solve/find shortest path, if -ve wt cycle doesn't exist. (drawback if exist)
- Slower than dijkstra algo but capable of handling -ve wt. edges.

★ n-1 time relaxation of all edges are done (n is no. of edges)
↳ $d(u) + c(u,v) < d(v)$ then, $d[v] = d[u] + c[u,v]$

Lets take a source edge (or given 'A')




5 vertices are there so 6-1 = 5 time relaxation done. i.e 5 iteration.

But when doing manually we know that after third iteration values are not updating so stop!

- Write all the possible edges in any order.
  (A,B), (A,C), (A,D), (B,E), (C,E), (D,C), (D,F), (E,F), (C,B)

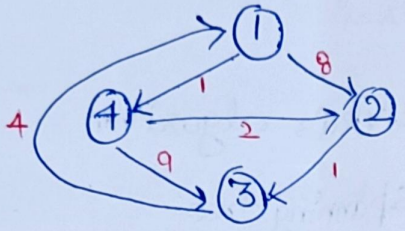
# #All pair Shortest path

① Apply Floyd-Warshall algo. for constructing shortest path for all pairs.

✓ As a result of this algo, it generates matrix which will represent min. dist. from any node to all other nodes in graph.

- Floyd Warshall Algo. is an example of Dynamic Programming

Adv: Simple and easy to implement.




Step1: Remove all self loop and parallel edges.

Step2: Write initial dist. matrix which represent dist b/w every pair of vertices

- For diagonal element dist. = 0
- For vertices having directed edge, dist = weight of edge.
- For vertices having no directed edge, dist = ∞.

initial matrix

$$d_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

Step 3: Use floyd Warshal algo to generate a final matrix for all pairs path one by one.

$$d^k(i,j) = \min\left[d^{k-1}(i,j),\ d^{k-1}(i,k) + d^{k-1}(k,j)\right]$$

1st matrix

$$d^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$d^1[2,3] = \min[d^0(2,3),\ d^0(2,1)+d^0(1,3)]$

$d^1[2,4] = \min[d^0(2,4),\ d^0(2,1)+d^0(1,4)]$

$d^1[3,2] = \min[d^0(3,2),\ d^0(3,1)+d^0(1,2)]$

$d^1[3,4] = \min[d^0(3,4),\ d^0(3,1)+d^0(1,4)]$

go by 1 to all vertex

$d^1[4,2] = \min[d^0(4,2),\ d^0(4,1)+d^0(1,2)]$

$d^1[4,3] = \min[d^0(4,3),\ d^0(4,1)+d^0(1,3)]$

2nd matrix

$$d^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$d^2[1,3]$ $d^2[3,4] = d^1[3,4]$ or $d^1[3,2]+d^1[2,4]$

$d^2[1,4]$ $d^2[4,1]$

$d^2[3,1]$ $d^2[4,3]$

go by path (2)

3rd matrix

$d^3$ =

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 8 | 9 | 1 |
| 2 | 5 | 0 | 1 | 6 |
| 3 | 4 | 12 | 0 | 5 |
| 4 | 7 | 2 | 3 | 0 |

$d^3[1,2]$
$d^3[1,4]$
$d^3[2,1]$
$d^3[2,4]$

$d^3[4,1] = \min(d^2[4,1], d^2[4,3]+d^2[3,1])$
$d^3[4,2]$

Go by path ③ for all

4th matrix;

$d^4$ =

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 3 | 4 | 1 |
| 2 | 5 | 0 | 1 | 6 |
| 3 | 4 | 7 | 0 | 5 |
| 4 | 7 | 2 | 3 | 0 |

$d^4[1,2]$ $d^4[1,3]$ $d^4(2,1]$ $d^4[2,3]$
$d^4[3,1]$ $d^4[3,2] = d^3[3,2]$ or $d^3[3,4]+d^3[4,2]$

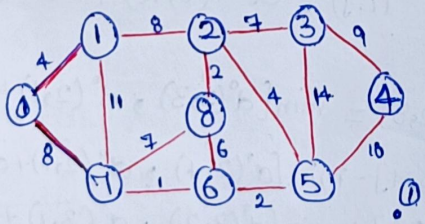
Go by path ④

final answer
all pairs shortest path

# Sollin's Algo; / Boruvka's algorithm

- Used to find Minimum Spanning tree.
- This algo is a greed algorithm and hybrid of prim's & kruskal.

Step1: Input is connected, weighted and undirected graph.
Step2: Initialize all vertex as individual component.
Step3: Initialize MST as empty.
Step4: While there are more than one components, do following for each;
① find closest wt. edge (adj edge) that connects this component to any another component.
② Add this closest edge to MST if not already added.
Step5; Return MST.

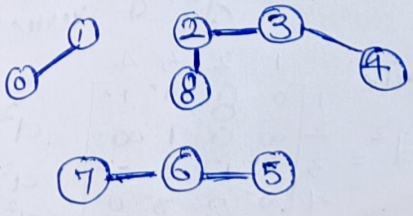



0 : 0-1
1 : 0-1
2 : 2-8
3 : 2-3
4 : 3-4
5 : 5-6
6 : 6-7
7 : 6-7
8 : 2-8

- Initially MST is empty and every vertex is single component.
ie component are 0, 1, 2, 3, 4...

① Find the min. cost edge for all components.

⇒ ② now MST become



After 1 step the new component / sets are;

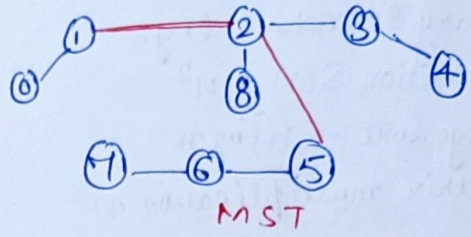
(0,1), (2,3,4,8) & (5,6,7)

Repeat step 4 again.

↳ (0,1): 1-2 or 0-7

(2,3,4,8): 2-5

(5,6,7): 2-5



MST

Now there exist only one component (0,1,2,3,4,5,6,7,8)

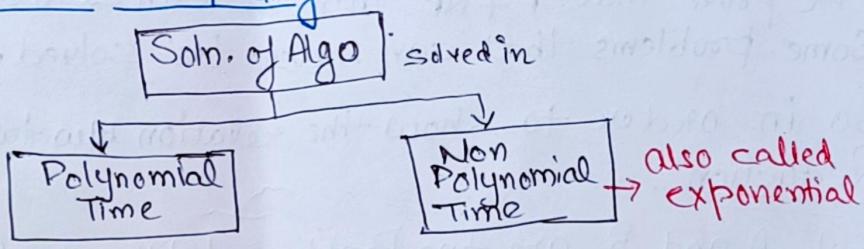
apply step (5) ↪

(6) Exit

---

# UNIT-III

## ch-8 Computational Complexity



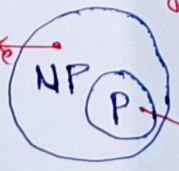

**P class**; Problems which can be solved on polynomial time is known as P-class problem

Ex: All searching and sorting algo. are in P-class.
→ linear, binary, Insertion, Merge ---

**NP-class**; A prob. which cannot be solved on polynomial time but is verified in polynomial time, is known as non-deterministic Polynomial or NP-class.
(once solved)

Ex: Sudoku, n Queen, travelling Salesman, Prime factor.

P and NP difference




NP class
- Hard to solve
- Easy to verify once the problem is solved
- Exponential time

P class
- Easy to solve
- Easy to verify
- Polynomial-time.

$P \subseteq NP$

P is subset of NP

## Polynomial time problems

Linear Search - $n$
Binary search - $\log n$
Insertion Sort - $n^2$
Merge Sort - $n \log n$
matrix multiplication $n^3$

## Non-Polynomial time Problems

0-1 knapsack - $2^n$
Travelling SP - $2^n$
Sum of subset - $2^n$
Graph coloring - $2^n$
Hamiltonian cycle - $2^n$

* Want to solve them in poly. time *

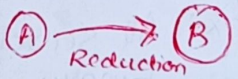
**Problem; Is P = NP?**

if we prove that P=NP then online security is vulnerable to attack, everything become efficient such as :- Transportation, scheduling, DNA understanding etc.

if we prove that P≠NP then we can also say that there are some problems that can never be solved.

- So in order to show the relation btw two problems we used **Reduction**

Let A and B are two Problems of different classes.
if we (succeed to) reduces problem A to problem B in polynomial time then; we can say A is directly proportional to B. $A \propto B$
So when we reduce A to B




then, if the problem 'A' is solved by deterministic algo or in poly. time then we can say prob. 'B' can also be solved in poly. time.

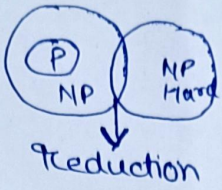
**Properties of reduction**

1) If A is reducable to B and B is in 'P' class then A is also in 'P'
2) If A is not in P then B is not in P
   $A \neq P \Rightarrow B \neq P$

# NP- hard problem

A problem is N-P hard if every problem in NP is reduced in polynomial time.

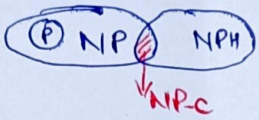
Every NP-H is a NP until it solved in polynomial time.




ex; Optimization problem.
To find best solution from all feasible solutions.

# NP- Complete

A problem is NP complete if it is in NP and NP-Hard or can say that the intersection of NP and NP Hard classes are NPC




ex; Decision problem

True or false / either choose or neglect like ~~0-1~~ knapsack problem.

- Determining wheather a graph has hamiltonian cycle.

★ If problem A is decision problem and B is optimization problem then it may be possible that $A \propto B$

★ All NP- Complete Problem are NP Hard
But All NP-Hard problems are not NP-Complete.