GATE CSE NOTES

by

UseMyNotes

Graph Algorithms.

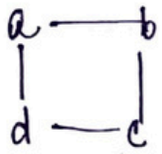
1. BFS/BFT, DFS/DFT
2. Graph Cycle
   - Detect cycle
   - Union-Find Algorithm

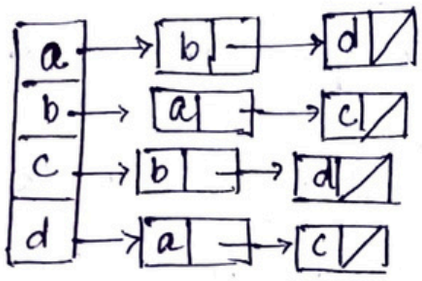
   | Finding connected components using Disjoint Set DS
3. Topological Sorting
4. MST (Prim's, Kruskal's, Boruvka's)
5. Backtracking (m-coloring problem, Hamiltonian cycle)
6. Shortest Path Algo (Dijkstra's (SSSP), Bellman-Ford, Floyd Warshall (APSP))
7. Connectivity (Connected components, Eulerian path-circuit)
8. Maximum flow problem.

Graphs

* **Representation:**

1. Adjacency matrix $O(n^2)$ or $O(V^2)$
2. Linked list (Adjacency list).





Undirected graph

$O(V+E)$

Dense

$E = O(V^2)$

↓

Adj. matrix.

Sparse

$E = O(V)$

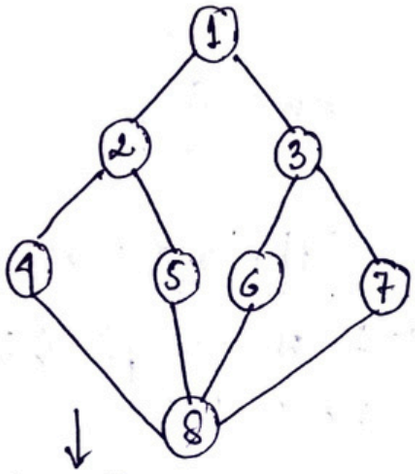
↓

Adjacency list.

কম সংখ্যক edges থাকলে adj. list

বেশি সংখ্যক edges থাকলে adj. matrix

* **Graph traversal / search**



BFS Queue | DFS Stack

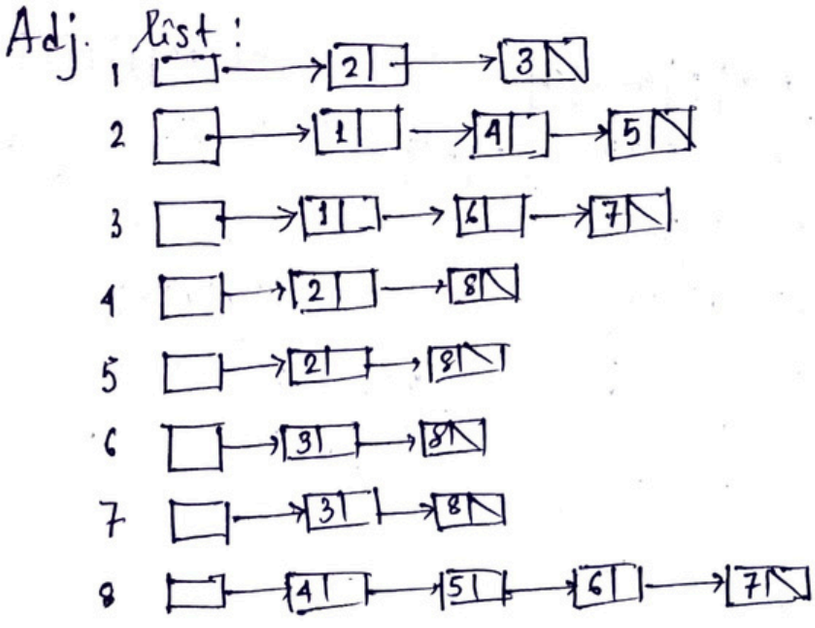
Space $O(n)$ | space $O(n)$

BFS : 1 2 3 4 5 6 7 8

DFS : 1 2 4 8 5 6 3 7

↓

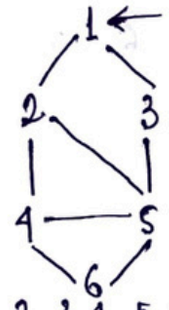
Adj. list:



* BFS.

```
Algorithm BFS(v)
// visited[] array global, initialised to zeros
{
    u := v ;
    visited[v] := 1 ;
    // q is a queue of unexplored vertices
    repeat {
        for all vertices w adjacent from u do {
            if (visited[w] = 0) then {
                Add w to q ; // w unexplored
                visited[w] := 1 ;
            }
        }
        if q is empty then return ; // no unexplored vertex
        Delete next elem., u, from q ; // get 1st unexp. vertex
    } until (false) ;
}
```

eg.



Visited array V[] Queue Q

1 2 3 4 5 6
V | 0 | 0 | 0 | 0 | 0 | 0 |

Q :

① → u : 1 | w : {2, 3}

o/p : 1

② V | 1 | 0 | 0 | 0 | 0 | 0 |

u : 1 | w : {2, 3}

③ V | 1 | 1 | 0 | 0 | 0 | 0 |

u = 1 w : {2, 3} Q : | 2 |

o/p : 1, 2

④ V | 1 | 1 | 1 | 0 | 0 | 0 |

u = 1 w : {2, 3} Q : | 2 | 3 |

o/p : 1, 2, 3

⑤ V | 1 | 1 | 1 | 0 | 0 | 0 |

→ u = 2 w : {1, 5, 4}

Q : | 3 |

o/p : 1, 2, 3

⑥ V | 1 | 1 | 1 | 0 | 1 | 0 |

u = 2 w : {1, 5, 4}

Q : | 3 | 5 |

o/p : 1, 2, 3, 5

⑦ V | 1 | 1 | 1 | 1 | 1 | 0 |

u = 2 w : {1, 5, 4} →

Q | 3 | 5 | 4 |

o/p : 1, 2, 3, 5, 4

⑧ V | 1 | 1 | 1 | 1 | 1 | 0 |

u = 3 w : {1, 5}

Q | 5 | 4 |

(9) V | 1 | 1 | 1 | 1 | 1 | 0 |

u = 5 W : {3, 2, 4, 6}

Q | 4 |

O/p 1, 2, 3, 5, 4

(10) V | 1 | 1 | 1 | 1 | 1 | 1 |

u = 5 W : {3, 2, 4, 6}

Q | 4 | 6 |

O/p 1, 2, 3, 5, 4, 6

(11) V | 1 | 1 | 1 | 1 | 1 | 1 |

u = 4 W : {2, 5, 6}

Q | 6 |

O/p 1, 2, 3, 5, 4, 6

(12) V | 1 | 1 | 1 | 1 | 1 | 1 |

u = 6 W : {4, 5}

Q

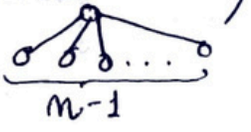
O/p | 1, 2, 3, 5, 4, 6 |

Analysis in case of adj. list rep^n :

Space complexity : $O(n)$ for visited array

$O(n-1)$ for queue (Worst case)

$\Rightarrow O(n)$ or $O(V)$



Time complexity : $O(E + V)$

E → exploration

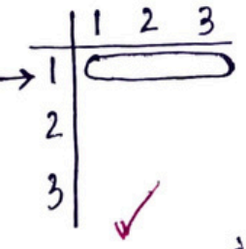
V → initialisation

As adj. list is used, all vertices adjacent to u can be determined in time $d(u)$, $d(u)$ being the degree of u. Since, each vertex can be explored at most once, total time for exploration $O(\Sigma d(u)) = O(2E) = O(E)$. To init. visited[i] it takes $O(V)$ time. Total time = $O(V + E)$.

1. Each node visited at most once.
2. For each visited node, # adjacent vertices $\approx O(d(u))$
We explore each adjacent node.
Hence, for exploration, $O(\Sigma d(u))$

BFS analysis in case of adjacency matrix:

Space complexity: $O(V)$, analysis same as before.



Time complexity: It takes $\theta(v)$ time to determine vertices adj. to u 1 time in total is $O(n^2)$ or $O(V^2)$. Initialisation - $O(V)$

So, TC is $O(V^2)$.

→ We can check graph connectivity using BFS. Starting with a vertex u, if at the end of algo. visited array becomes all 1 then connected.

→ Search techniques are used for connected graphs, as it may not explore all nodes if the graph is not connected. (Search - visit nodes that are reachable)

But, when the search necessarily involves the examination of every vertex in the object being searched, it is called a traversal.

Breadth First Traversal :

```
Algorithm BFT (G, n) {
    for i := 1 to n do
        visited [i] := 0;
    for i := 1 to n do
        if (! visited [i]) then BFS (i);
}
```

BFS is called for each component of graph once.

	Adj. list	Adj. Matrix
SC	$O(V)$	$O(V)$
TC	$O(V+E)$	$O(V^2)$

- Pseudocode BFS.

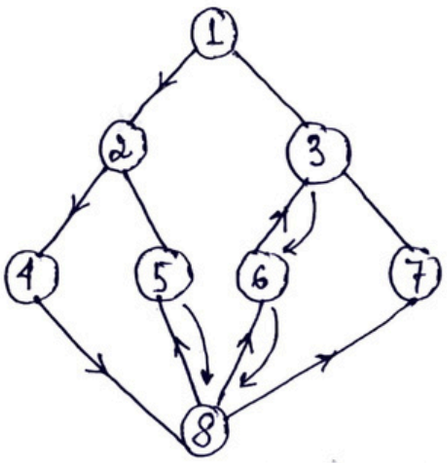
Set all nodes to "not visited"

```
Void BFS () {
    int v;
    for (v = 0; v < n; v++)
        visited [v] = 0
    printf ("Enter start vertex: ");
    scanf ("%d", &v);
    int i;
    push_queue (v);
    while (! is_Empty_Queue ()) {
        v = pop_queue ();
        if (visited [v])
            continue;
        printf ("%d ", v);
        visited [v] = 1;
        for (i = 0; i < n; i++) {
            if (adj [v][i] == 1 && visited [i] == 0)
                push_queue (i);
        }
        printf ("\n");
}
```

* DFS.

```
Algorithm DFS (v)
// array visited[]  set to 0
{
    visited[v] := 1;
    for each vertex w adjacent to v do
    {
        if (visited[w] = 0) then DFS(w);
    }
}
```

- **Example:**



visited

0	0	0	0	0	0	0	0

Stack

v = 1	v = 2	v = 4	v = 8	v = 5	v = 6	v = 3	v = 7
w = {2,3}	w = {1,4,5}	w = {2,8}	w = {1,5,6,7}	w = {2,8}	w = {3,8}	w = {1,6,7}	w = {3,8}

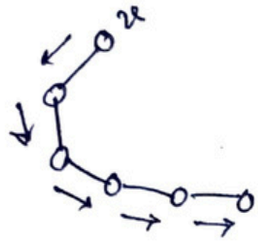
O/p: 1, 2, 4, 8, 5, 6, 3, 7.

- **Code:**

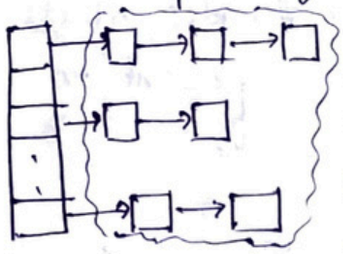
```
void DFS ( struct Graph* graph, int vertex) {
    struct node * ptr = graph → adjLists [vertex] ;
    graph → visited [vertex] = 1 ;
    printf (" %d ", vertex );
    while ( ptr) {
        int adj_vertex = ptr → vertex ;
        if ( graph → visited [ adj_vertex] == 0)
            DFS ( graph, adj_vertex );
        ptr = ptr → next ;
    }
}
```

- **Analysis of DFS.**

SC: O(V) for visited array. Also, in the worst case (a chain graph) the stack can grow upto O(V). So, SC is O(V).



TC: In case of adj. lists.



= 2E

We call DFS for each node in the list. for a node all adjacent vertices are visited once.

So, for that O(2E) = O(E)

Also, the initialization : O(V)

$\Rightarrow$ O(V+E)

In case of adj. matrix : $O(V^2)$.

- **DFT :**

Algorithm DFT {

```
for i := 1 to n do
    visited [i] := 0 ;
for i := 1 to n do
    if (! visited [i]) then DFS (i) ;
}
```

SC : O(V)

TC : O(V+E) O(V²)
list matr.

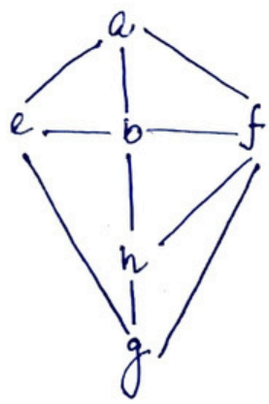
Q'ns Sequences possible if run DFS

i) a b e g h f

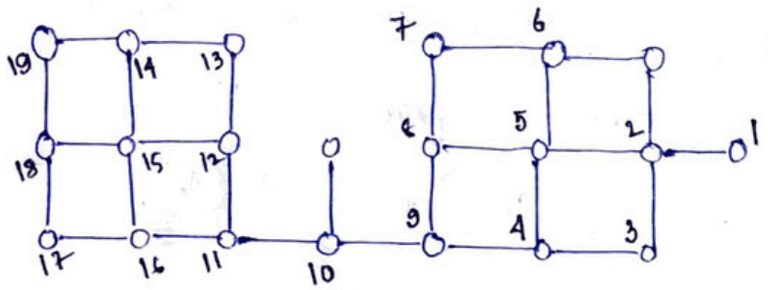
ii) a b f e h g

iii) a b f h g e

iv) a f g h b e



Q'14 Deepest stack possible (longest path) for this (np-hard)



(use intuition to find longest path starting from any node)

Ans: 19 max # nodes that can be present in the stack

Q'06 IT → DFT

Consider a DFS of an undirected graph with 3 vertices P, Q, R. Let discovery time d(u) represent the time instant when the vertex u is first visited, & finish time f(u) represent the time instance when the vertex u is last visited. Given that:

$d(P) = 5$ $d(R) = 14$ $f(Q) = 10$

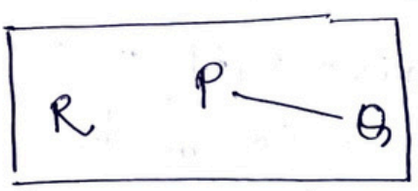
$d(Q) = 6$ $f(P) = 12$ $f(R) = 18$

What kind of graph this is.?

→ P first → Q first → Q last → P last → R first → R last

(DFT) not DFS

There are 2 connected components & P & Q are connected.



* If a graph ~~(has no loop)~~ and all edges are unweighted or of the same weight, then we can use BFS to find shortest path.

We use modified version of BFS in which we keep storing the predecessor while performing the algo.

* Using BFS, we can record levels
* Using BFS/DFS, we can find connected components.
* BFS tree : Edges explored by BFS form a tree.
  Acyclic graph = connected, with n-1 edges
  (Non-BFS tree edges form cycle)
* DFS tree : Non-tree edges form cycle.
* Directed cycles : Tree edge, Forward edge, Back edge, Cross edge

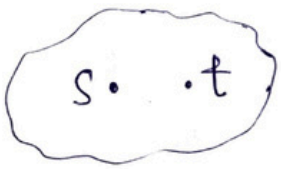
A directed graph has cycle only if its DFS reveals a back edge.

BFS - DFS Appl^n^s

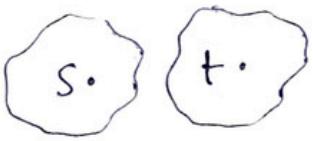
* BFS applications.

1. Finding set of connected components.

- Maximal connected subgraph.



case I



case II

Applications.

i) Network connectivity

ii) Clustering

- For any 2 nodes s & t in G, their connected components are either identical or disjoint.

- BFS of node i discovers the connected component containing i.

```
connComp = 0
Discovered[i] = false  ∀ i : 1 to n

for i = 1 to n
    if Discovered[i] = false
        connComp ++;
        BFS(i);
```

O(V + E)

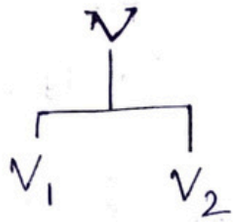
initialisation ↑ edge processing ↑

- One edge can belong to only one connected component.

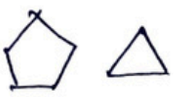
2. The path found by the BFS to any node is the shortest path to that node from a single source (smallest # edges in unweighted graph).

3. Testing Bipartiteness.

$G = (V, E)$



$\forall e \in E$ s.t. $u \in X$, $v \in Y$, $e = (u, v)$

- Odd length cycles are not bipartite. 

1, 2, 3, ..., 2K, 2K+1
R B R B R

⇒ Graph containing odd len cycle is not bipartite.

- Absence of odd cycles proves bipartiteness (can be proved).

- Procedure (assume G is connected, if not check for every component)

Pick any node s - color it red.

neighbor(s) - blue

O(V+E) neighbor(neighbor(s)) - red

⋮

until all nodes are colored.

Does every edge has ends of opp. colors?

Yes ↙ Bipartite

No ↘ Not bipartite

Bipartite Graph Test using DFS:

```
bool dfs (int v, int c) {
    visited[v] = 1;
    color[v] = c;
    for (int child : or[v]) {
        if (visited[child] == 0) {
            if (dfs (child, c ^ 1) == false)
                return false;
        }
        else {
            if (color[v] == color[child])
                return false;
        }
    }
    return true;
}
```

(c ^ 1 : XOR)

---

4. Finding the shortest path in a graph with weights 0 or 1 (0-1 BFS).

(In each step of BFS, we check the optimal distance condition. Here, we use doubly ended queue to store the node. If edge weight is 0, then push it at the front, otherwise at the rear.)

5. Finding shortest cycle in a directed unweighted graph (as soon as we go from current vertex to the source vertex, we have found a cycle containing source vertex. Find all such cycles, choose shortest.)

- BFS tree.

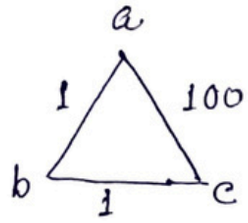
Tree formed by BFS.

- Let x & y be nodes in $Layer_i$ & $Layer_j$ of the BFS tree, & let (x,y) be an edge of G. Then, i & j differ by at most 1 (& nodes in adjacent layers).

- BFS tree gives the shortest path from source to any node (for unweighted graphs).

⇒ Layer L_j is the set of all nodes at distance exactly j from s.

⇒ For weighted graphs, doesn't give shortest path.



6. Finding diameter of graph.

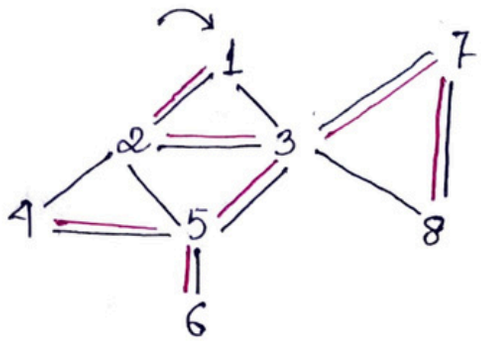
(Max. distance b/w 2 vertices in graph)

- Perform BFS on all nodes.
- Output the max # levels in any BFS tree.

* DFS applications.

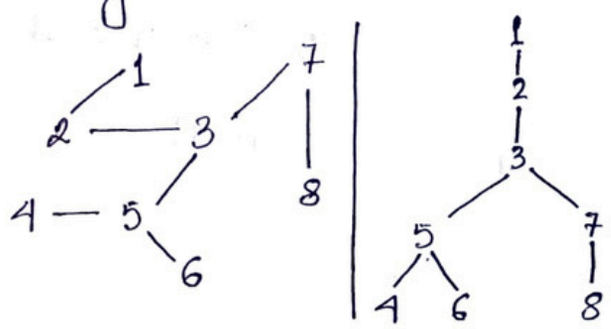
- Maintain 2 timestamps for each node while processing DFS : 'discovery' & 'processing done'. Helps to know how we visited the nodes, what was the order.

	discovery	proc. done
1	0	15
2	1	14
3	2	13
4	4	5
5	3	8
6	6	7
7	9	12
8	10	11



- DFS-tree not unique.

- n-1 edges in DFS-tree



- discovery time (ancestor) < discovery (descendant) < finish time (desc.) < finish time (anc.)

Used in top-sort. (DAG)

if $(u, v) \in E$,

$u < v \quad \forall e \in E$.

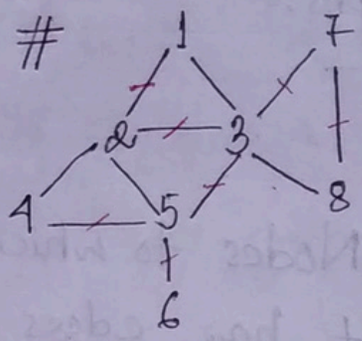
i) Perform DFS.

ii) Arrange nodes in decreasing order of finish times.

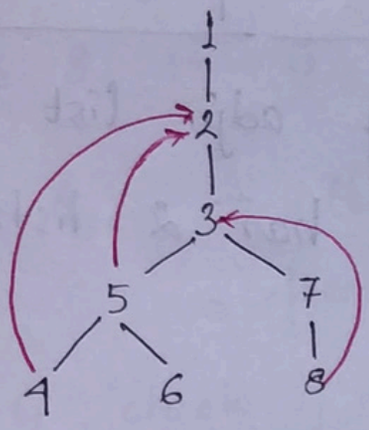
For above graph, 1, 2, 3, 7, 8, 5, 6, 4.

DFS TREE

#

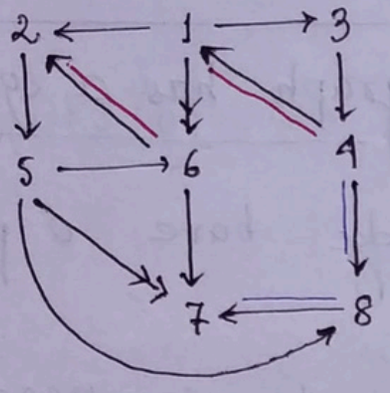


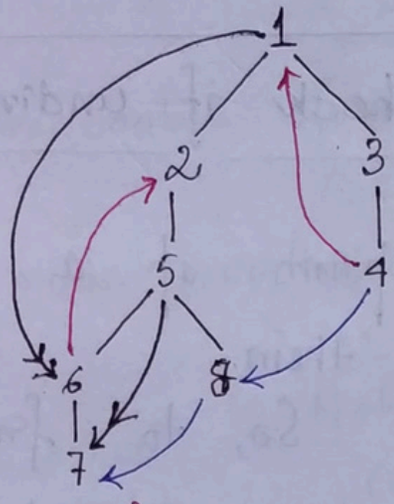
Undirected – only tree & back edges



Back edges

#





Directed graph →

→ Tree edges

→ Back edges (form cycle)

→ Cross edges →» Forward edges

↓

always from right to left (?)

Tree edge (u,v) discover(u) < disc(v) ... < finish(v) < fin(u)

Fwd edge (u,v) disc(u) < disc(v) < fin(v) < fin(u)

Back edge (u,v) disc(u) > disc(v) and fin(u) < fin(v)

Cross edge (u,v) disc(v) < fin(v) < ... < disc(u) < fin(u)

1. Application 1 : Connected components

```
for i = 1 to n
    if discovered[i] = false
        DFS(i)
```

* Representation of directed graphs.

Variant of adj. list

Each node has 2 lists : a) Nodes to which it has edges

b) Nodes from which it has edges

---

2. Check if undirected graph has a cycle.

End points of a back edge have a path b/w them.

So, to find a cycle, we need to detect back edge.

Back edge ⇒ cycle

No Back edge ⇒ No cycle

3. Check if directed graph has a cycle

By checking if there's back edge.

4. Given a directed graph, and a node s find set of nodes having path to s.

→ $G_{rev} = (V, E_{rev})$

E_{rev} with dir^{n}s reversed

A node has a path from s in G_{rev} iff it has a path to s in G.

⇒ DFS(s) in G_{rev} → all nodes reachable from s in G_{rev}

5. Given a directed G, check if it is strongly connected (2 nodes mutually reachable)

DFS(u) → nodes reachable from u

↓

Is every node reachable from u?

↙ Yes

u has path to every other vertex

↓

repeat for all vertices

↘ No

Not strongly connected

If DFS performed on every node yields the vertex set V,

⇒ each node is reachable from every other node

⇒ any 2 nodes are mutually reachable.

So, strongly connected.

TC $O(V(V+E))$

$\rightarrow O(VE)$

O(V+E) algorithm

DFS(u) : all nodes reachable from u?

↙ Yes ↘ No ⇒ no!

$G_{rev} = (V, E_{rev})$

↓

DFS(u)

all nodes reachable from u? $\xrightarrow{No}$ No!

↓ Yes

all nodes have a path to u in G.

$u \rightsquigarrow s$ $s \rightsquigarrow u$. $\forall s \in V$

for $v_1, v_2 \in V$, u & v_1, u & v_2 are mutually reachable.

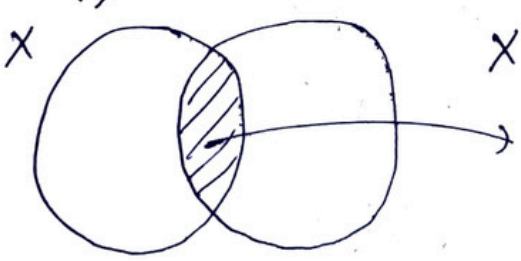
⇒ v_1 & v_2 are mutually reachable.

So, strongly connected!

6. Strongly connected components in directed graphs:

Strong component containing node s

DFS(s) in G DFS(s) in G_{rev}



nodes mutually reachable with s,

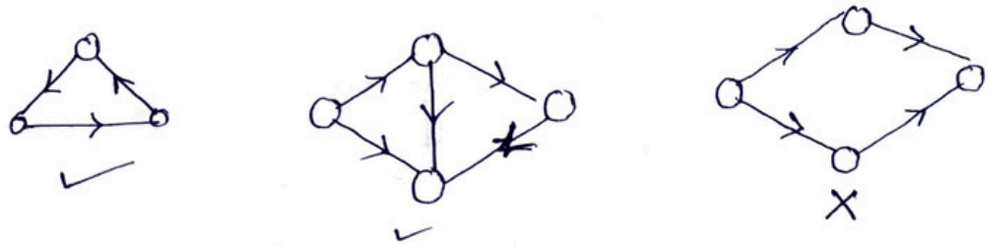
$X \cap X'$

⇓

strong component of s

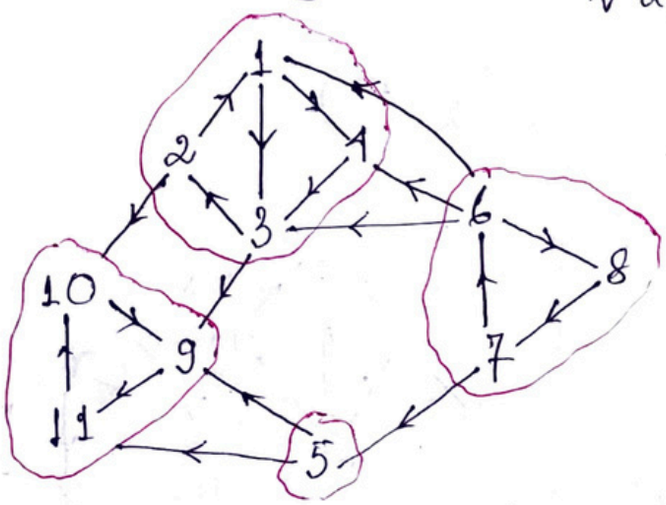
- For any 2 nodes s & t in a directed graph, their strong components are either identical or disjoint.

- Strongly connected example.



- Strong components example

$\forall u, v \; \exists \; u \rightsquigarrow v, \; v \rightsquigarrow u$ | maximal components.



(10, 9, 11)
(1, 4, 3, 2)
(5)
(7, 6, 8)

7. Finding strong components. (2)

Start DFS from a node in a component that has no out-edges.

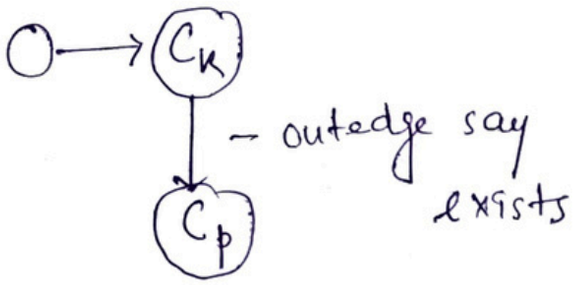
→ Strong components of G & G_{rev} would be the same.

→

in G_{rev} $C_i \rightarrow C_j$ ⇓ $f_{in}(c_i) > f_{in}(c_j)$	C – component but in G, $C_j \rightarrow C_i$

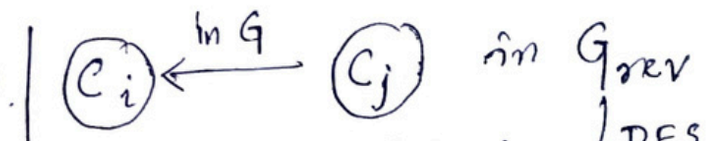
The maximum finish (·) would be for a sink strong component (that has no out-edges).

Say, C_K has the max f(·).



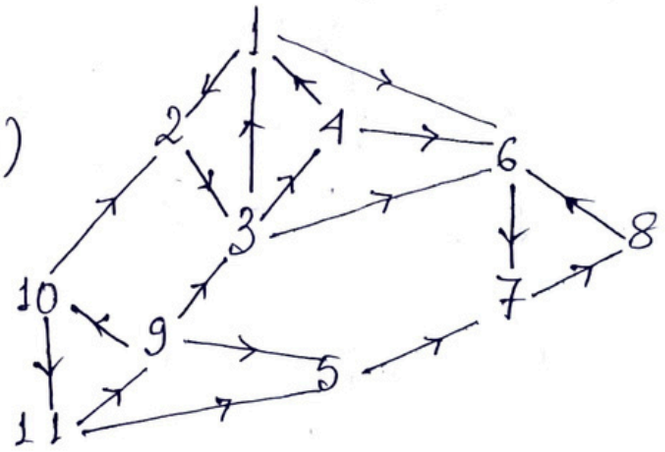
$f(C_p) > f(C_K)$. contradiction!

$\Rightarrow$ So, no out-edges.



Example (graph on the back 5)

↳ G_{rev} of G -



Start DFS on any node (in G_{rev})

DFS(1)

1 → 2 → 3 → 4 → 6 → 7 → 8

0/13 1/12 2/11 3/10 4/9 5/8 6/7

DFS(10)

10 → 11 → 9 → 5

14/21 15/20 16/19 17/18

Finish time in dec. order.

node no.	1	2	3	4	5	6	7	8	9	10	11
	10	11	9	5	1	2	3	4	6	7	8

Now take node 10 & DFS(10) in G.

(10 → 9 → 11) this is a strong component.

Ignore 11, 9 from array now.

Pick, 5' DFS(5) in G. (5) strong comp. ignore 5 now.

DFS(1) (1, 4, 3, 2)

DFS(6) (6, 8, 7)

Algo (Strong Component)

1. Let G_{rev} = G with directions of edges reversed.

2. Run DFS - loop on G_{rev} (determine finish time of all nodes).

3. Run DFS - loop on G in decreasing order of finish times determined in (2).

4. Output the nodes in each tree in the DFS forest obtained in (3) as a separate strongly connected component.

8. Finding lexicographical first path in the graph from source to all vertices.

9. Check if a vertex in a tree is an ancestor of some other vertex.

v_i is an ancestor of v_j iff

discovery (i) < discovery (j)

finish (i) > finish (j)

v_i
↓
v_j

10. Finding lowest common ancestor (LCA) of 2 vertices.

11. Finding bridges in an undirected graph:

First, convert the given graph into a directed graph by running a series of DFS & making each edge directed as we go through it, in the direction we went. Second, find the strong components in the directed graph. Bridges are the edges whose ends belong to different strong components.

Shortest Path Algorithms.

* In a shortest path problem, we are given weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights. The weight of path $p = \langle v_0, v_2, \ldots, v_k \rangle$ is the sum of the weighted of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the shortest-path weight from u to v by

$$\delta(u,v) = \begin{cases} \min \{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there's path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u,v)$.

* Shortest path algorithms typically rely on the property that a shortest path between 2 vertices contains other shortest paths within it.

* Types of shortest path problem.

i) Single source shortest path problem:

Shortest path from a given a source vertex to all other remaining vertices is computed. Dijkstra's algorithm (Greedy approach) & Bellman Ford algorithm (Dynamic programming) are the famous algorithms used for solving single-source shortest path problem. Many other problems can be solved by the algorithm for the single-source problem, including other 3 variants of shortest path problem.

ii) Single-pair shortest path problem.

Shortest path between a given pair of vertices is computed. A* search algorithm is a famous algorithm used for solving this. No algorithms for this problem are known that run asymptotically faster than the best single source algorithms in the worst case.

iii) Single-destination shortest path problem

Shortest path from all the vertices to a single destination vertex. By reversing the direction of the edges in the graph, we can reduce this problem to a single-source problem.

iv) All pairs shortest path problem.

Shortest path for every pair of vertices u & v. Floyd-Warshall algorithm & Johnson's algorithm are famous for solving this problem.

→ Optimal substructure of a shortest path:

Subpaths of shortest paths are shortest paths.

* Negative Weight Edges:

If the graph $G(V, E)$ contains no negative weight cycles reachable from the source s, then for all $v \in V$, the shortest path weight $\delta(s, v)$ remains well-defined, even if it has a negative valued weight.

If the graph contains a negative-weight cycle reachable from s, however shortest-path weights are not well-defined. No path from s to a vertex on the cycle can be a shortest path; we can always find a path with lower weight by following the proposed shortest path & then traversing the negative-weight cycle. If there is a negative-weight cycle on some path from s to v, we define $\delta(s,v) = -\infty$. (CLRS 582)

NB 1. Dijkstra's algorithm assumes that all edges' weights in the input graph are nonnegative.

*2. Bellman-Ford algo. allows ~~non~~-negative weight edges in the input graph & produce a correct answer as long as no negative-weight cycles are reachable from the source. If there's such a negative weight cycle, the algorithm can detect & report its existence.

* Relaxation.

Dijkstra's & Bellman-Ford use the technique relaxation.

For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a shortest path estimate. We initialize the shortest path estimates & predecessors by the following $\Theta(V)$ time procedure —

Initialize_single_source (G, s) $\quad \Theta(V)$

```
for each vertex v ∈ V[G]
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0.
```

$\pi[v]$ - predecessor of node v if followed the best path

After initialization $\pi[v] = NIL$ for all $v \in V$, $d[s] = 0$ & $d[v] = \infty$ for $v \in V - \{s\}$.

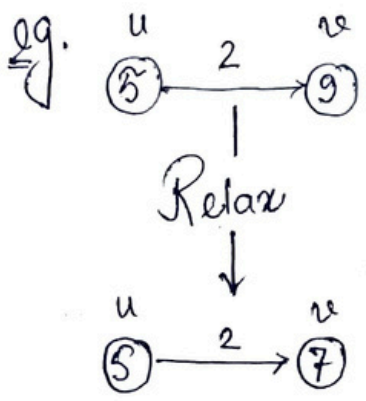
The process of relaxing an edge (u,v) consists of testing whether we can improve the shortest path to v found so far by going through u & , if so, updating $d[v]$ and $\pi[v]$.

Following performs a relaxation step on edge (u,v) in $O(1)$ time.

Relax (u, v, w) $\quad O(1)$

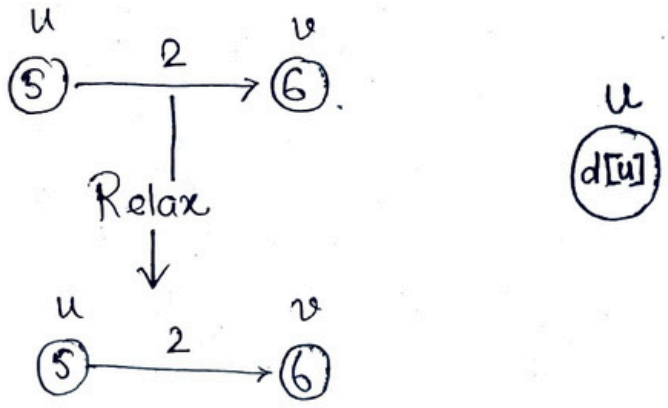
```
if d[v] > d[u] + w(u, v)
   then d[v] ← d[u] + w(u, v)
        π[v] ← u.        // setting predecessor of
                         // v to u.
```

Eg.



$d[v] > d[u] + w(u,v)$

$d[v]$ value decreases



$d[v] \leq d[u] + w(u,v)$

* Both D's & BF algos call Initialize_single-source & repeatedly relaxes edges. Moreover, relaxation is the only means by which shortest path estimates d[v] and predecessors π[v] change. In D's algo, each edge is relaxed only (exactly) once. In BF algo, each edge is relaxed many times.

* Dijkstra's Algorithm. - Greedy (GV)

Solves the single source shortest path problem on a weighted, directed graph G(V,E) for the case in which all edge weights are nonnegative. We assume $w(u,v) \geq 0$ for each edge $(u,v) \in E$.

Dijkstra's algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V-S$ or Q with the minimum shortest path estimate, adds u to S, and relaxes all edges leaving u. We use a min-priority queue Q of vertices, keyed by their d values.

Dijkstra's algorithm is like Prim's algo in that both algos use a min-priority queue to find the lightest vertex outside a given set (the set S in D's algo & the tree being grown in Prim's).

Dijkstra (G, w, s)

TC:
avg $O(|E| \log |V|)$
worst $O(|V|^2)$

SC:
worst $O(|V|+|E|)$

```
1.  Initialize_single_source (G, s)
2.  S ← ϕ
3.  Q ← V[G]
4.  while Q ≠ ϕ
5.      do u ← Extract_min (Q)
6.         S ← S ∪ {u}
7.         for each vertex v ∈ Adj [u]
8.             do Relax (u, v, w).
```

Line 3 initializes the min-priority queue Q to contain all the vertices in V.

Line 5 extracts a vertex u from $Q = V - S$ & line 6 adds it to set S. Lines 7-8 relax each edge (u,v) leaving u, thus updating the estimate $d[v]$ & the predecessor $\pi[v]$ if the shortest path to v can be improved by going through u.

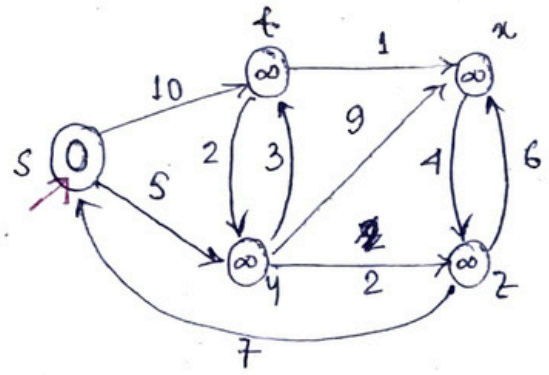
Because Dijkstra's algorithm always chooses the closest vertex, in $V-S$ to add to set S, we say that it uses a greedy strategy. Greedy strategies don't always yield optimal results in general, but Dijkstra's algorithm does indeed compute shortest paths. Each time a vertex u is added to set S, we have $d[u] = \delta(s,u)$.

- Keys in min-heap: nodes in $V-S$

  for $v \in V-S$, key $[v]$ = smallest value of greedy parameters $(d_u + l(u,v))$ of an edge (u,v) s.t. $u \in S$; if no such edge then $d'(v) = +\infty$

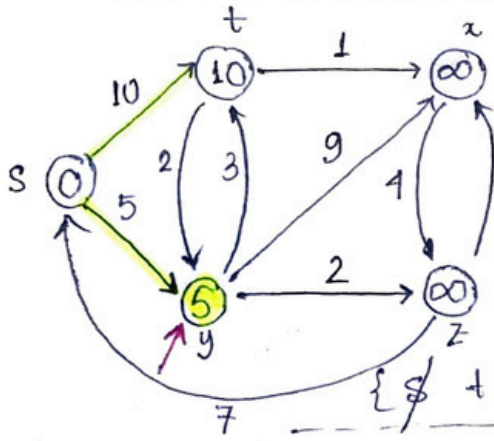
$d'(\cdot)$

Eg.



$S = \{s\ t\ x\ y\ z\}$ $S = \phi$

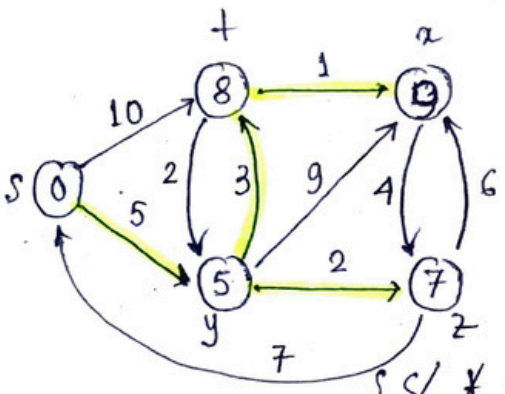
(s, t)
(s, y)



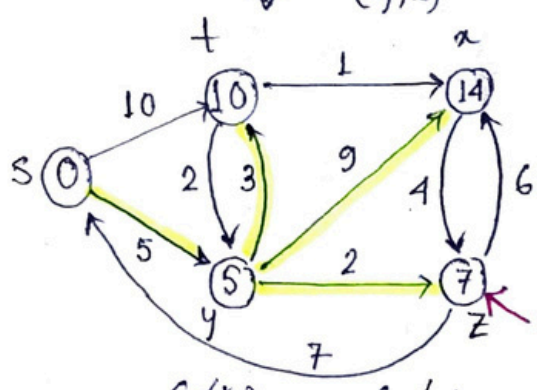
$S = \{s, y\}$

$\{s\ t\ x\ y\ z\}$

(y, t) (y, z)
(y, x)



$\{s, t\ x\ y\ z\}$

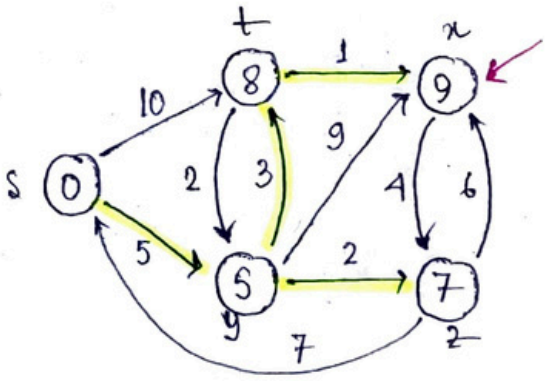


$S = \{s, y\}$

$\{s\ t\ x\ y\ z\}$

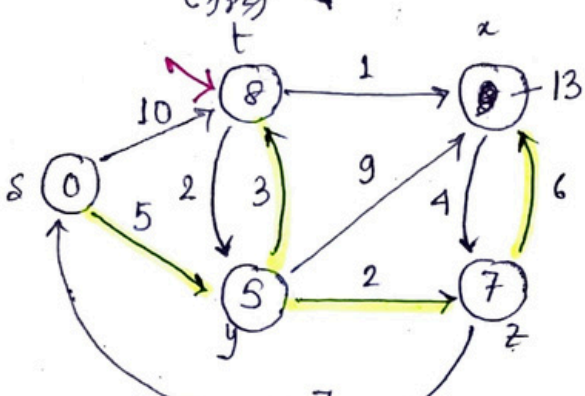
$S = \{s, y, z, t, x\}$

(z, x)



(t, x).

$\{s\ t\ x\ y\ z\}$



$S = \{s, y, z, t\}$

$S = \{s, y, z\}$

Now the set S by adding $\min(d_u + l(u,v))$ $u \in S, v \in V \setminus S$

Analysis. (how min-PQ is implemented)

Decrease_key is implicit in Relax. Insert is invoked once per vertex, as in Extract_min. Because, each vertex $v \in V$ is added to set S exactly once, each edge in the adjacency list adj[v] is examined in the for loop of lines 7-8 exactly once during the course of the algo. Since, the total number of edges in all the adjacency lists is $|E|$, there are a total of $|E|$ iterations of this for loop, & thus a total of at most $|E|$ Decrease_key operations.

1. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$. We simply store $d[v]$ in the vth entry of an array.

a) Each Insert & Decrease_key takes $O(1)$.

b) Each Extract_min takes $O(V)$ time.

c) Total time $O(V^2 + E) = O(V^2)$.

2. If we use Binary heap to maintain min-priority queue,

a) Each Extract_min takes $O(\log V)$.

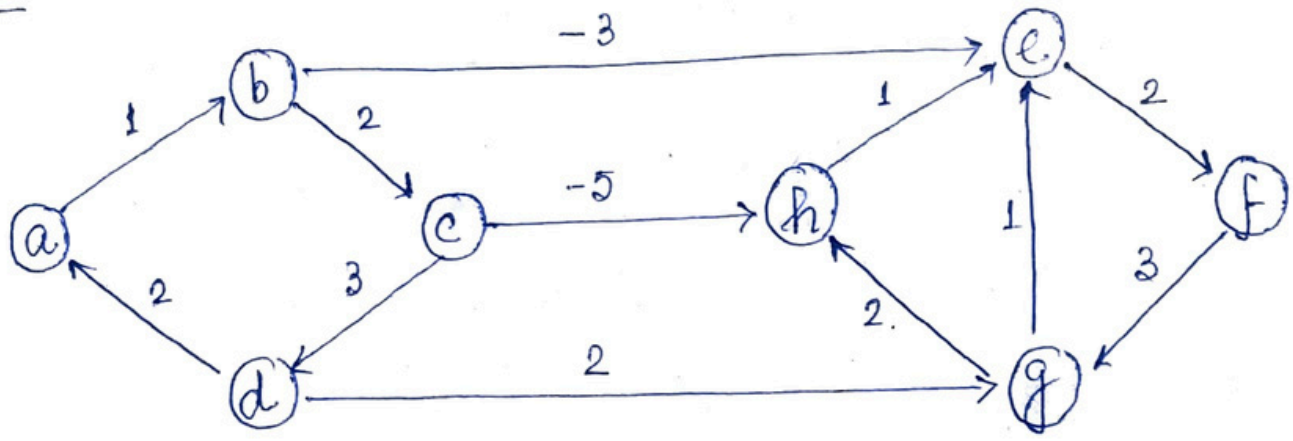
There are $|V|$ such operations. Time to build the binary min-heap is $O(V)$.

b) Each Decrease_key operation takes $O(\log V)$ time & there are still at most $|E|$ such operations.

c) Total running time is $O(V \log V + E \log V)$ which is $O(E \log V)$ if all vertices are reachable from the source.

3. If we use Fibonacci heap, Extract_min op^n takes $O(\log V)$ & Decrease_key $O(1)$, total TC will be $O(V \log V + E)$.

Q. G'08.



Dijkstra's single source shortest path algorithm when run from vertex a, computes the correct shortest path distance to

a) Only vertex a

b) Only vertices a, e, f, g, h

c) Only vertices a, b, c, d

✓ d) All the vertices.

→ Just by simulating D's, it works for this graph though D's is not guaranteed to work for graphs with -ve weight edges.

- D's works only for connected graphs.
  D's works for directed as well as undirected graphs.

* Bellman-Ford Algorithm.

Solves the single-source shortest-path problem in the general case in which edge weights may be negative.

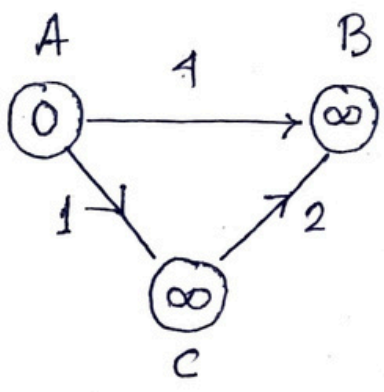
Given a weighted, directed graph $G(V, E)$ with source s and weight function $w : E \rightarrow \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no sol^n exists. If there is no such cycle, the algorithm produces the shortest paths & their weights.

Time complexity of Dijkstra's algo is better than BFA.

- Given n nodes in a graph, the shortest path between 2 nodes will have at most n-1 edges. This is why, in BFA, we relax the edges in the graph n-1 times. Relaxing edges n-1 times implies we are finding the shortest path ~~length with~~ weights when the path length is at most n-1. After n-1 times relaxation, we relax the edges

One more time to check if there is any -ve edge cycle. If after nth relaxation, the weights get decreased, it means there must be some -ve weight loop.

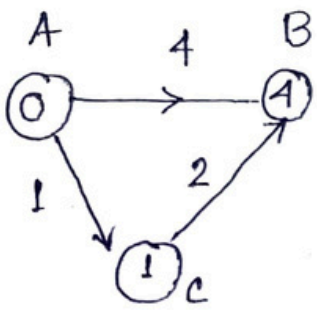
eg.



3 nodes

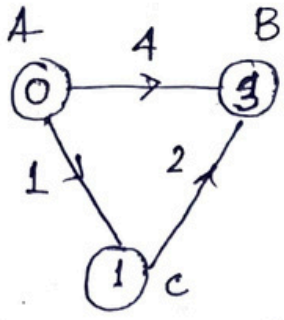
2 relaxations.

1st relaxation



Finding shortest path weights of length 1.

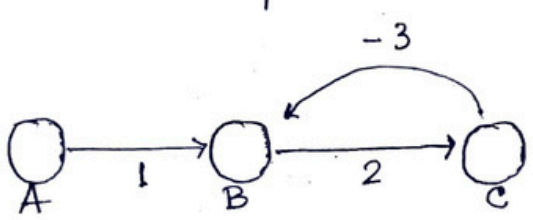
2nd relaxation.

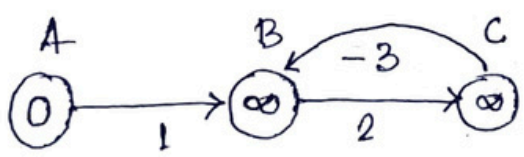


Finding shortest path weights of length 2.

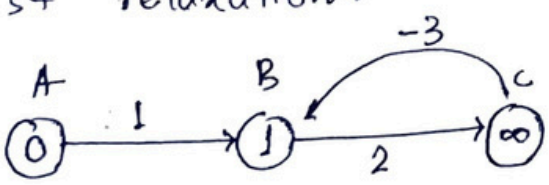
In the 3rd relaxation weights in the node remain same as no -ve edge cycle is present.

eg.

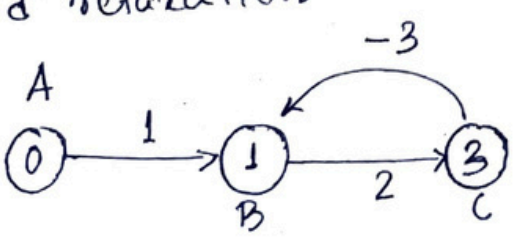




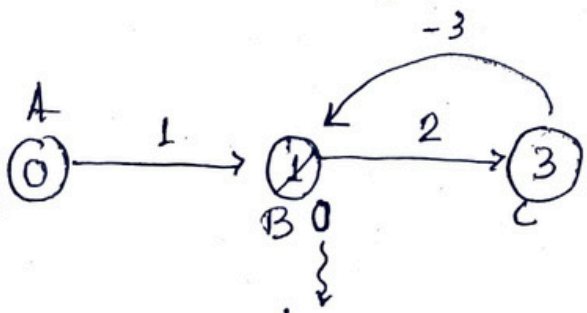
1st relaxation.



2nd relaxation



3rd relaxation to confirm about -ve edge cycle.



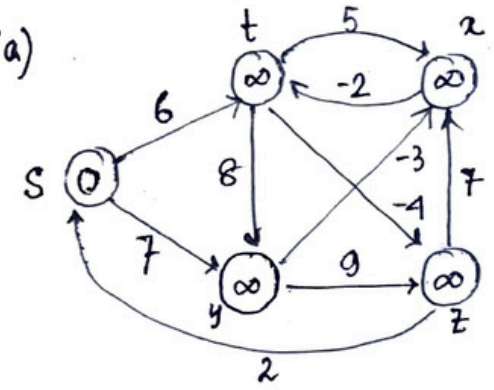
decrease

⇒ -ve edge cycle is present.

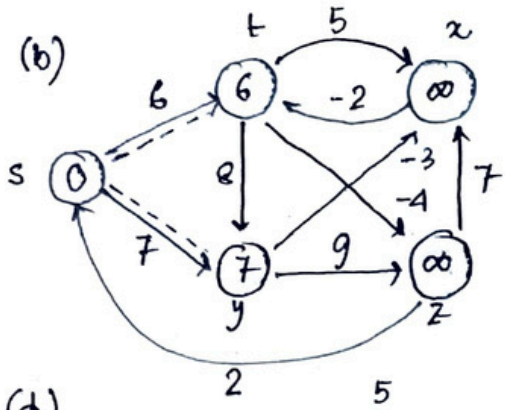
BFA runs for $|V| \times |E|$ relaxations. So, time complexity of BFA is $O(VE)$.

→ Execution of BFA.

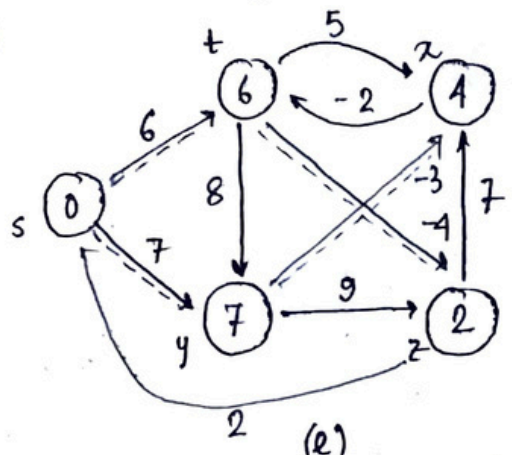
(a)



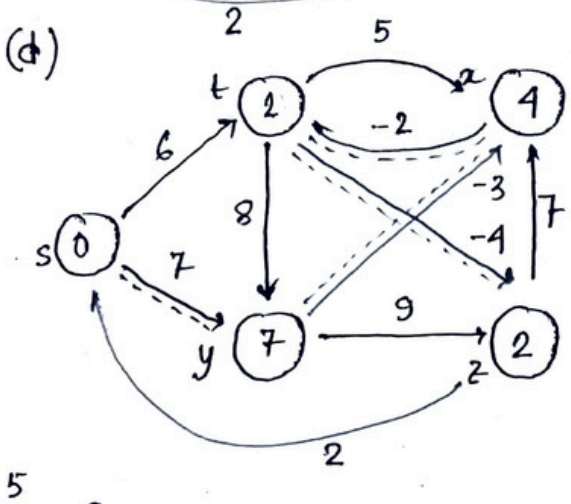
(b)



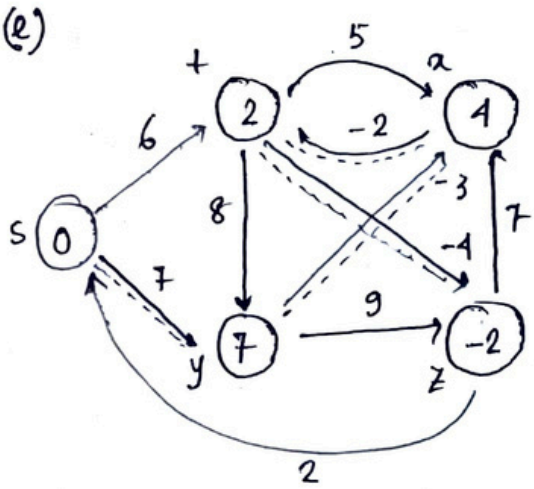
(c)



(d)



(e)



final value of d & π.

Each pass relaxes the edges in the order —

(t,x) (t,y) (t,z) (x,t) (y,x) (y,z) (z,x) (z,s) (s,t) (s,y)

Shortest paths from (e) figure

s to t - s y x t (2)

s to x - s y x (4)

s to y - s y (7)

s to z - s y x t z (-2).

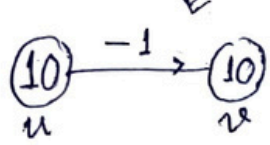
- Algorithm.

The algorithm relaxes edges, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest path weight $\delta(s, v)$.

TC:
avg, worst $O(|E||V|)$
SC: $O(|V|)$

BELLMAN_FORD (G, w, s)

```
1.  Initialize_single_source (G, s).
2.  for i = 1 to |G.v| - 1
3.      for each edge (u, v) ∈ G.E
4.          RELAX (u, v, w) ---- O(1) as we take
5.  for each edge (u, v) ∈ G.E          an array.
6.      if v.d > u.d + w (u, v)
7.          return False
8.  return True.
```



Line 1 initializes the d and π for all vertices.
Line 2 for loop iterates $|V|-1$ times. Each pass is one iteration of the for loop of lines 2-4 & consists of relaxing each edge
(TC) of ~~deg~~ the graph once. After that, lines 5-8 check for a -ve weight cycle & return the appropriate boolean value.

Initialization in line 1 takes $\Theta(V)$ time, each of the $|V|-1$ passes over the edges in lines 2-4 takes $\Theta(E)$ time & the for loop of lines 5-7 takes $O(E)$ time. So, time complexity is $O(VE)$.

* Shortest path in DAG (Directed Acyclic Graph)

- Single source shortest path
- By relaxing the edges of a weighted DAG $G(V,E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\Theta(V+E)$ time.
- Shortest paths are always well defined in DAG, since even if there are -ve weight edges, no -ve weight cycles can exist.
- The algorithm starts by topologically sorting the DAG to impose a linear ordering on the vertices. If the DAG contains a path from u to v, then u precedes v in the topological sort. We make just one pass over the vertices in the topologically sorted order. As we process, each vertex, we relax each edge that leaves the vertex.

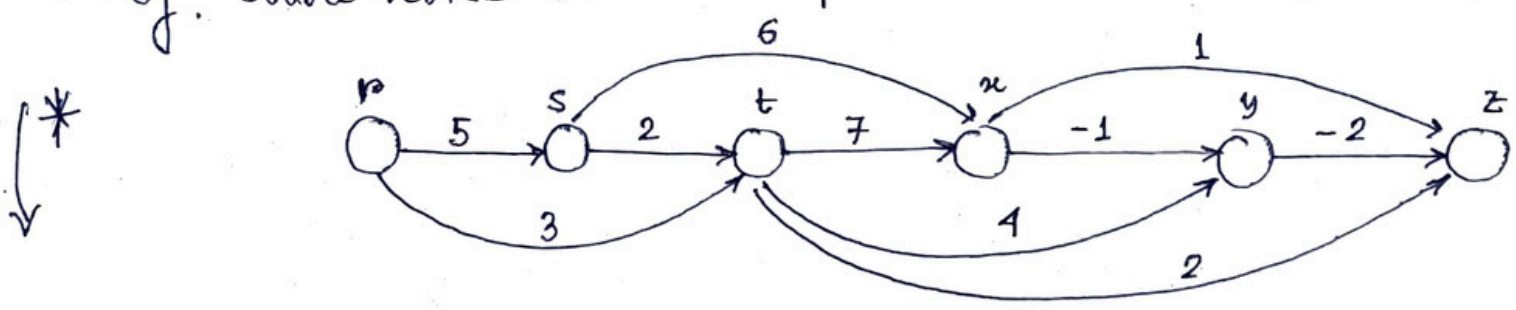
- DAG_SHORTEST_PATHS

```
1  topologically sort the vertices of G
2  Initialize_single_source (G, s)
3  for each vertex u, taken in topologically sorted or
4  |    for each vertex v ∈ G.Adj [u]
5  |    |  RELAX (u, v, w)
```

- Topological sort of line 1 takes $\Theta(V+E)$ time. Line 2 takes $\Theta(V)$ time. for loop of 3-5 makes one iteration per vertex. Altogether, the for loop of 4-5

relaxes each edge exactly once. Because each iteration of the inner loop takes $\Theta(1)$ time, the total running time is $\Theta(V+E)$, which is linear in the size of an adjacency-list representation of the graph.

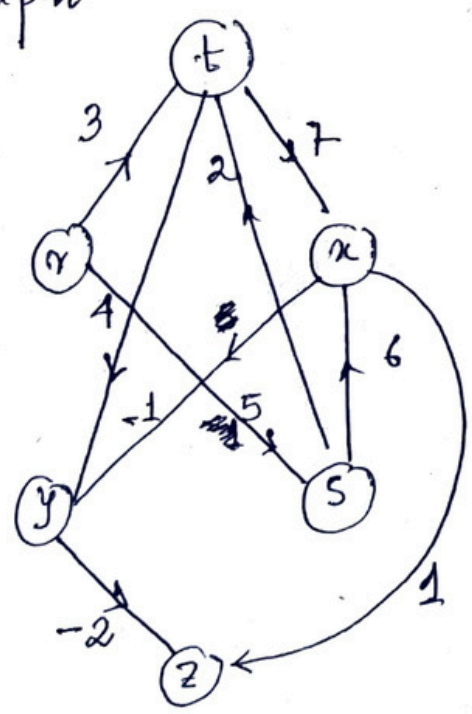
eg. Source vertex s. top-ordered



	r	s	t	x	y	z
	∞	0	∞	∞	∞	∞
Relaxing for r	∞	0	∞	∞	∞	∞
for s	∞	0	2	6	∞	∞
for t	∞	0	2	6	6	4
for x	∞	0	2	6	5	4
for y	∞	0	2	6	5	3
for z	∞	0	2	6	5	3 (final)

---

* Actual graph

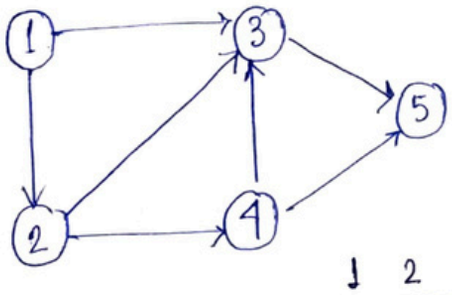


⇒ top. ordering

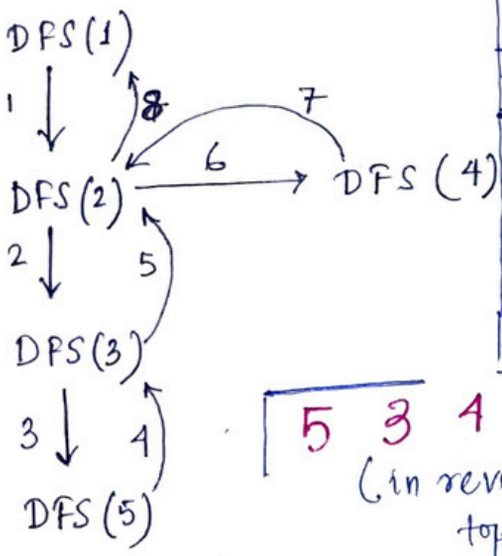
r, s, t, x, y, z

- Topological Sort. (for DAG) $O(V+E)$

Use finishing time for DFS



	1	2	3	4	5
visited	✓	✓	✓	✓	✓



	Entry	Exit
1	0	9
2	1	8
3	2	5
4	6	7
5	3	4

Based on the last finishing time of DFS.

1, 2, 4, 3, 5

5 3 4 2 1 (in reverse order - top. sorted)

- topologically sorted

- Defn: Linear ordering of vertices such that for each directed edge uv for vertex u to v, u comes before v in the ordering. Every DAG has at least one top. sorting order.

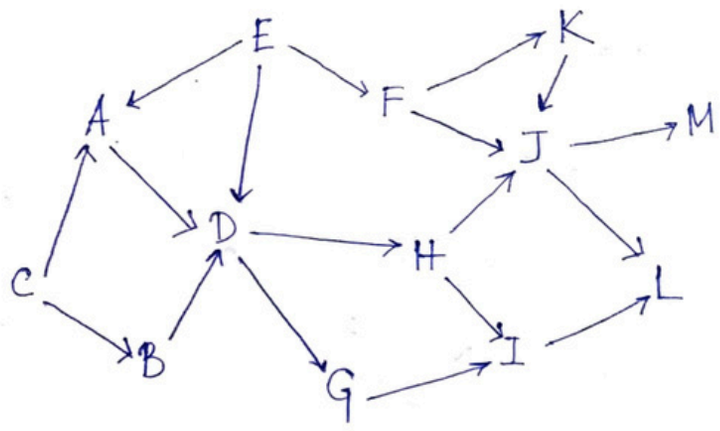
- Algo: a) Pick an unvisited node.

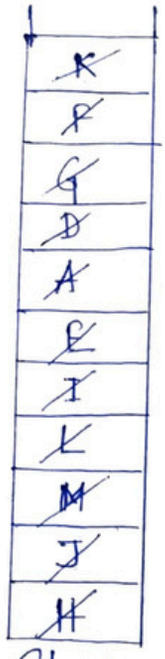
b) Beginning w the selected node, do a DFS exploring only unvisited nodes.

c) On the recursive callback of the DFS, add the current node to the topological ordering in reverse order.

```
function dfs(at, V, visitedNodes, graph):
    V[at] = true
    edges = graph.getEdgesOutFromNode(at)
    for edge in edges:
        if V[edge.to] == false:
            dfs(edge.to, V, visitedNodes, graph)
    visitedNodes.add(at)
```

eg.





Pick unvisited node — H 1st
E 2nd.
C 3rd

Top. ordering (in reverse) ↓

M - L - J - I - H - G - D - A - K - F - E - B - C.

Visited.
H J M L I E A D G F K C B

* Pseudocode. Graph stored as adj. list

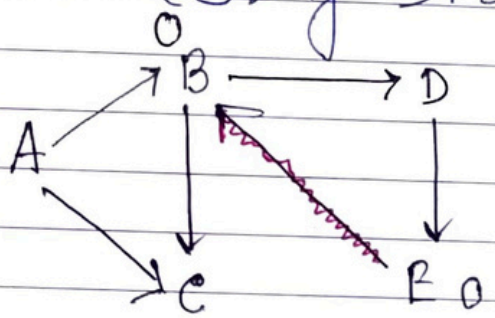
```
function topsort (graph):
    N = graph.numberOfNodes();
    V = [false, ..., false]   # length N
    ordering = [0, ..., 0]    # length N
    i = N-1 ; # index for ordering array
    for (at = 0 ; at < N ; at++):
        if V[at] == false:
            visitedNodes = []
            dfs (at, V, visitedNodes, graph)
            for nodeId in visitedNodes:
                ordering[i] = nodeId
                i = i-1
return ordering.
```

dfs on back

233-132
34th Week

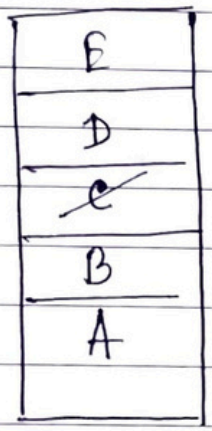
Thursday 21

✱ Detect cycle in Directed Graph (Using DFS).



flag : -1 unvisited
0 visited, in stk
1 visited, popped from stk

Stack



Visited set

A B C D E

Parent map

Vertex	Parent
A	-
B	A
C	B
D	B
E	D

If any vertex finds its adj. vertex to have flag 0, then there's cycle.

Cycle : E → B, D → E, B → D

checking parent of E

checking parent of D

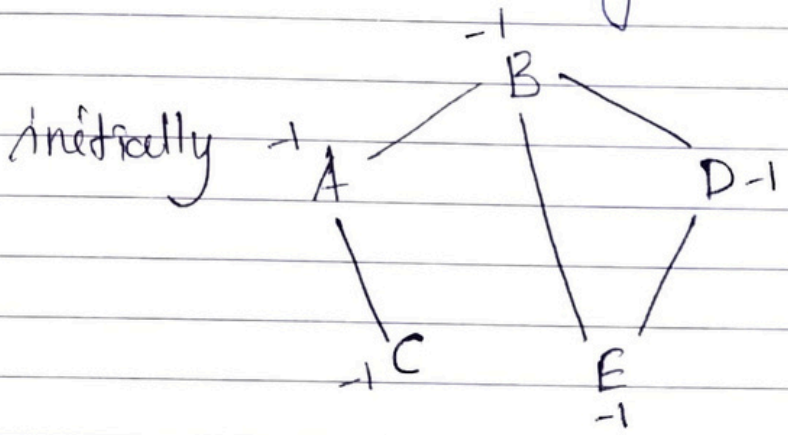
B → D → E

Essential | Job to do | Phone No.

235-130
34th Week

Saturday 23

※ Detect cycle in Undirected Graph (Using BFS)

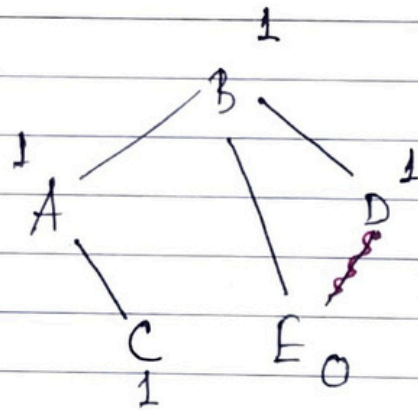


flag:

-1 unvisited
0 in queue
1 traversed

Queue

A	B	C	D	E



Visited
A B C D

24 Sunday

If found a vertex, whose adj. vertex with flag 0, then contains cycle.

Essential	Job to do	Phone No.

Dynamic Programming

- Solve the problem by combining the solutions to smaller subproblems.
- In DP, the subproblems overlap & the solutions to inner problems are stored in memory. This avoids the work of repeatedly solving the innermost problem.
- DP is most often used to solve combinatorial optimization problems, where we are looking for the best possible input to some function chosen from an exponentially large search space.
- DP is applicable when the subproblems are not independent. Subproblems share ~~subp~~ subproblems. Divide & conquer does more work than necessary repeatedly solving common subproblems.
- 2 methods of storing the result in memory -

1. Memoization (Top-down)

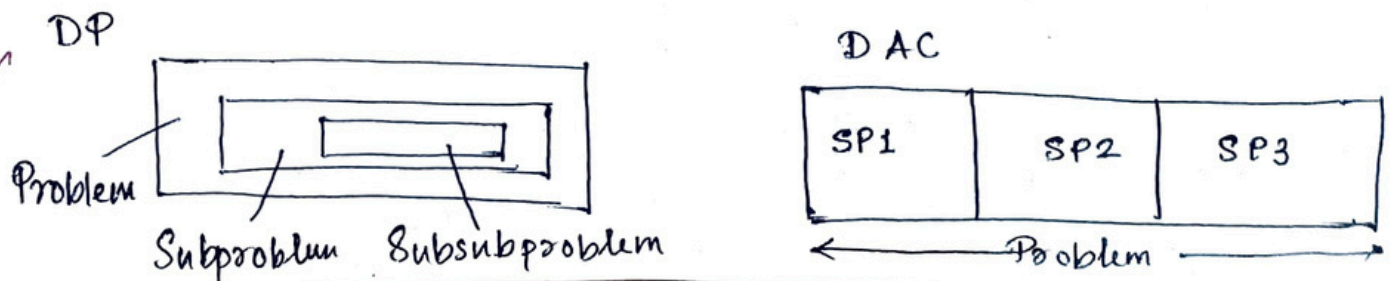
Whenever we solve a problem first time we store it & reuse it next time (make memo).

2. Tabulation (Bottom-up)

We precompute the solutions in linear fashion & later use them.

Overlapping subproblems
Optimal substructure

- In Divide & Conquer, subproblems are disjoint.



• The key feature that a problem must have in order to be amenable to DP is that of optimal substructure : the optimal solution to the problem must contain optimal solutions to subproblems.

- Overlapping subproblems.

- Longest path problem doesn't have optimal substructure property.

* Example with n^{th} Fibonacci number.

$$F(n) = F(n-1) + F(n-2)$$

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

Naive recursive approach is -

```
int fib (int n).
{
    if (n < 2)
        return 1 ;
    else
        return fib (n-1) + fib (n-2) ;
}
```

$$T(n) = T(n-1) + T(n-2) + \theta(1) = \Theta(a^n)$$

a is golden ratio (1.618033...)

Problem is we keep recomputing values of fib that we've already computed. We avoid this by memoization, where we wrap our solution to a memoizer that stores previously computed solutions in a hash table.

STEPS IN DP

Optimal Substructure
↓
Recursive Eqn
↓
Recursion Tree
↓
Overlapping subproblems ?
↓ Yes
Memoise / Tabulate

```
int memoFib (int n) {
    int ret ;
    if ( hashContains (FibHash, n)) {
        return hashGet (FibHash, n) ;
    } else {
        ret = memoFib (n-1) + memoFib (n-2) ;
        hashPut ( FibHash, n, ret) ;
        return ret;
    }
}
```

Bottom up DP. ~

```
int fib2 (int n) {
    int *a ;
    int i, ret ;
    if (n < 2) {
        return 1;
    } else {
        a = malloc (sizeof (*a) * (n+1)) ;
        assert (a) ;
        a[1] = a[2] = 1 ;
        for (i = 3 ; i <= n ; i++) {
            a[i] = a[i-1] + a[i-2] ;
        }
    }
    ret = a[n] ;
    free (a) ;
    return ret ;
}
```

$\Theta(n)$.

* Matrix Chain Multiplication.

Two matrices of size $p \times q$ & $q \times r$ when multiplied, we need to do $(p \times r) \times q = p \times q \times r$ no. of scalar multiplications.

– Given a sequence of n matrices $\langle A_1, A_2, \ldots, A_n \rangle$ to be multiplied, & we wish to compute the product $A_1 A_2 \cdots A_n$.

– If the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$ then we can fully paranthesize the product in 5 distinct ways.

$(A_1 \ (A_2 \ (A_3 A_4)))$.	$((A_1 \ (A_2 A_3)) A_4)$.
$(A_1 \ ((A_2 A_3) A_4))$	$(((A_1 A_2) A_3) A_4)$.
$((A_1 A_2) \ (A_3 A_4))$	

– We state the matrix chain multiplication problem as follows : given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully paranthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

– Counting no. of parenthesizations. :

Total no. of ways = $(n-1)^{th}$ Catalan number

$$= \binom{2(n-1)}{(n-1)} \Big/ n$$

$$= \frac{(2n)!}{(n+1)!\, n!}$$

For $n = 5$ matrices, # ways $= \frac{(2 \times 4)!}{5!\, 4!} = 14$.

- No. of alternative paranthesizations of a sequence of n matrices be $P(n)$. When $n = 1$, we have just one matrix. When $n \geq 2$, a fully paranthesized matrix product is the product of two fully paranthesized matrix subproducts, & the split between the two subproducts may occur between the k^{th} & the $(k+1)^{st}$ matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 \end{cases}$$

Soln to this recurrence being $\Omega(2^n)$, brute force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

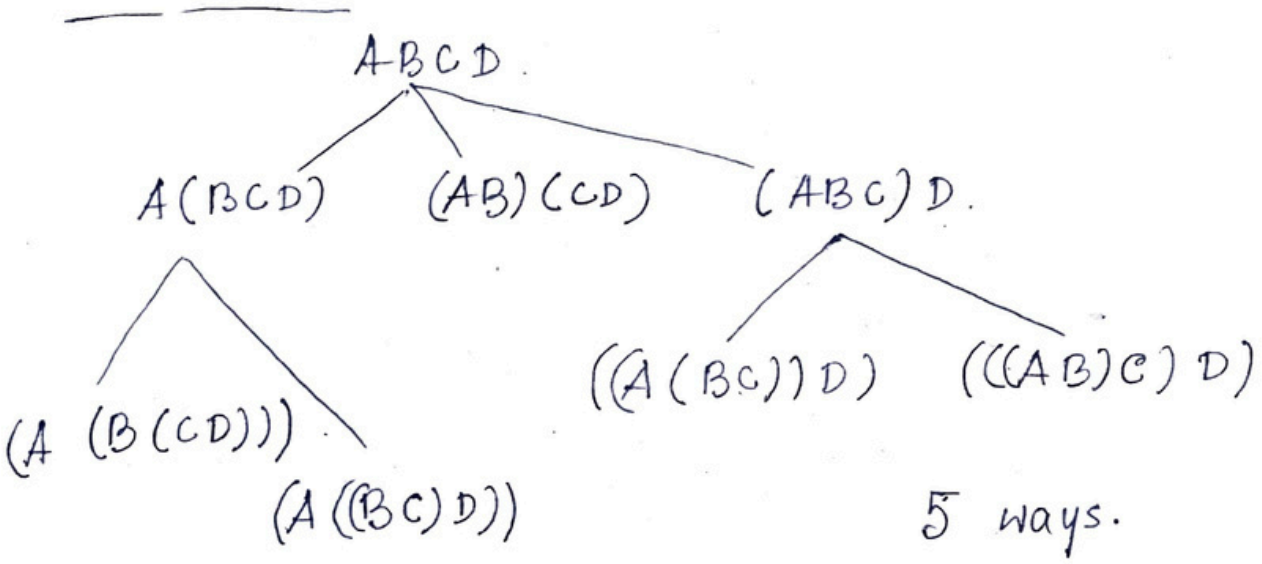
- eg. Paranthesization -

3 matrices

ABC

A(BC) (AB)C 2 ways.

4 matrices



5 ways.

# of scalar multiplications -

A	2 x 1
B	1 x 2
C	2 x 3
D	3 x 1.

$(A(B(CD))) \rightarrow (2\times3\times1) + (1\times2\times1) + (2\times1\times1)$

$= 6 + 2 + 2 = (10)$

$(A((BC)D)) \rightarrow (1\times2\times3) + (1\times3\times1) + (2\times1\times1)$

$= 6 + 3 + 2 = (11)$

$(AB)(CD) \rightarrow (2\times1\times2) + (2\times3\times1) + (2\times2\times1)$

$= 4 + 6 + 4 = (14)$

$((A(BC))D) \rightarrow (1\times2\times3) + (2\times1\times3) + (2\times3\times1) = 6 + 6 + 6 = (18)$

$(((AB)C)D) \rightarrow (2\times1\times2) + (2\times2\times3) + (2\times3\times1) = 4 + 12 + 6 = (22)$.

So, (A(B(CD))) is the optimal parenthesization way to minimise # of scalar multiplications.