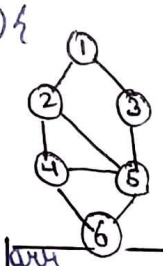
GRAPH

1) BFS - Breadth first Search
→ O(E)

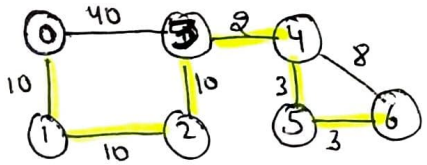
```
void bfs(vector<vector<int>>& arr, int v){
    vector<bool> vis(v,0);
    queue<int> q;
    q.push(src); vis[src]=1;
    while(!q.empty()){
        int x= q.front(); q.pop();
        cout<<x<<" ";
        for(int nbr : arr[x])
        {
            if(vis[nbr]) continue;
            q.push(nbr);
            vis[nbr]=1;
        }
    }
```



arr
1- 2 3
2- 1 4 5
3- 1 5
4- 2 5 6
5- 2 3 4 6
6- 4 5

traversal
1 2 3 4 5 6

2) PRIMS → Algo for MST → Greedy Algo
→ remains connected while calc. ans.
→ E log V

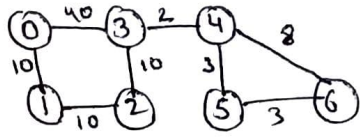


ans= 0 + 10+10+10+2 +3 +3

Order of Removal
1 (0,0)
2 (1,10)
(3) (3,40) vis
3 (2,10)
4 (3,10)
5 (4,2)
6 (5,3)
(8) (6,8) NIS
7 (6,3)

```
int prims(vector<vector<pp>> arr,
  int v, int src){
priority-queue<pp,vector<pp>,cmp>
    pq;
pq.push({src,0}); //node,cost
while(!pq.empty()){
    pp p=pq.top(); pq.pop();
    if(vis[p.first]) continue;
    vis[p.first]=1
    ans+=p.second;
for(auto nbr: arr[p.first])
{
    if(vis[nbr.first]==0) pq.push({nbr.first,
                                   nbr.second});
}}return ans;
```

3) Dijkstra → Shortest path from source to all vertices
only +ve edge weight ← → Greedy
→ E log V



Order of removal
1 (0,0)
(3) (3,40) vis
2 (1,10)
3 (2,20)
4 (3,30)
5 (4,33)
6 (5,35)
7 (6,38)

```
void dijkstra(vector<vector<
  pp>>& arr, int v, int src){
priority-queue<pp,vector<pp>,
    cmp> pq;
pq.push({src,0});
vector<bool> vis(v,0);
vector<int> dis(n, INT_MAX); // shortest distance
                             // from source to ith vertex
vector<int> par(n); // ith vertex ka parent
dis[src]=0; par[src]=-1;
while(!pq.empty())
{
    pp p= pq.top(); pq.pop();
int nod= p.first, cost= p.second;
if(vis[nod]) continue;
    vis[nod]=1;
for(auto nbr: arr[nod])
{
  int nbrnode=nbr.first, nbrcost=nbr.second;
  if(vis[nbrnode]) continue;
  pq.push({nbrnode, cost+nbrcost});
  if(dis[nbrnode]>dis[node]+nbrcost){
  dis[nbrnode]=dis[node]+nbrcost;
   parent[nbrnode]=node;
  }
}}
```

If shortest dist. from source to destination
`if(node==dest) return cost;`

→ dis vector contains distance from source to ith node
→ par contains from where we came so that path is minimum.

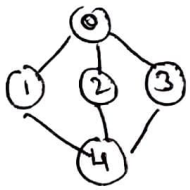
To get path

```
vector<int> path;
while(dest != -1)
{
  ans.push_back(dest);
   dest = par[dest];
}
reverse(path.begin(), path.end());
```

4) **DFS** → O(E)

→ goes deeper until no further node is nbr.

```
void dfs(vector<vector<int>>& arr, vis,
         int src){
  if(vis[src]==0) continue
  cout << src << " ";   vis[src] = 1;
  for(int nbr: arr[src])
  {
    if(!vis[nbr])
      dfs(arr, v, nbr);
  }
}
```



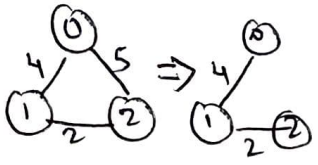
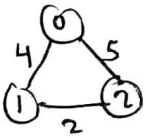
0 1 4 2 3

5) **Topological Sorting**

3 COLOURING ALGORITHM → O(E)

→ Topological sorting
→ Check cycle in Directed graph
→ Vertices that form cycle.

```
bool dfs(vector<vector<int>>& arr,
   vector<int>& vis, int src)
{
if(vis[src]==2) return 1; // no cycle, src
                          // has completed
                          // all its calls
if(vis[src]==1) return 0; // cycle src has not
  // completed its calls & you visited it again
```

Prims	Dijkstra
→ MST [any weight]	→ Shortest path [+ve edge weight]
→ pq.push({nbr, cost of nbr})	→ pq.push({nbr, cost + cost of nbr});
	
→ get shortest path	src:0 dest:2 cost:5 
	src:0 dest:1 cost:4 
	→ minimise overall weight

```
vis[src] = 1;  //start visiting src
for(int nbr: arr[src])
{
  if(dfs(arr, vis, nbr)==0) return 0;
                  //cycle is present.
}
vis[src] = 2; //source has visited all
              his child nodes w/o cycle.
ans.push_back(src);
return 1;  //no cycle found;
```

→ pushing into my ans when all outdegrees are 0 or visited.

```
void toposort(vector<vector<int>>& arr, int v)
{
  vector<bool> vis(v,0);
  for(int i=0; i<v; i++)
  {
    int x = dfs(arr, vis, i);
    if(x==0) // cycle
  }
```

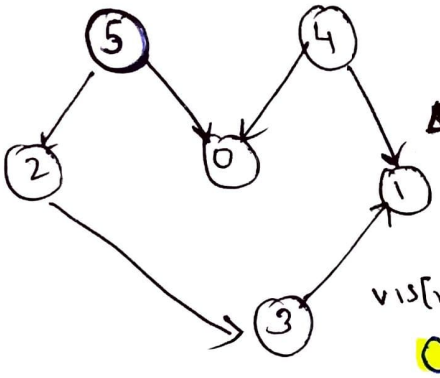
If cycle is present

```
  for(int i=0; i<v; i++)
  {
    if(vis[i]==1) cout << i << " ";
  }
```

Vertices in cycle

```
  reverse(ans.begin(), ans.end());
  for(int it: ans) cout << it << " ";
}
```

Topological sorting



45 23 10

vis[n] = 0|1|2

0 ⇒ unvisited

1 ⇒ visiting

2 ⇒ visited

BFS (KAHN'S ALGO) - O(E)

↳ detect cycle

↳ Topological sorting

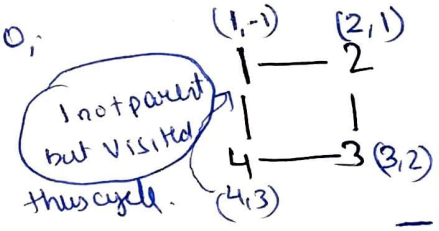
```
queue<int> q;
for (auto it : arr)
{
    for(int i: it) indgr[i]++;
}
for(int i=0; i<V; i++)
{
  if(indgr[i]==0) q.push(i); // 5,4
}
while(!q.empty())
{
  int n= q.front(); q.pop(); // 5
  ans.push_back(n);
  for(int i: arr[n])
  {  indgr[i]--;          // 0-1
                          // 2-0
    if(indgr[i]==0) q.push(i); // 2
  }
}
```

indgr 5-0 4-0
0-2 2-1
3-1 1-2

- If any indgr ≠ non 0 → cycle is present

6) Cycle Detection in Undirected Graph

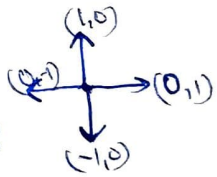
```
bool dfs(vector<vector<int>> &arr, vector vis, int src,
{ vis[src]=1;                          int par)
  for(int nbr: arr[src]){
    if(!vis[nbr])
    {
      if(dfs(arr,vis,nbr,src)) return 1;
    }                                  // cycle
  else if(nbr!=par) return 1;
        // ie nbr is visited & not the
        // parent ie from where we came.
  }
return 0;
}
```



— O(E)

7) Flood fill

```
int dx[4]= { 0, 0, 1, -1 }
int dy[4]= { 1, -1, 0, 0 }
```




```
for(k=0; k<4; k++)
  x= i+dx[k];  y= j+dy[k];
```

```
int dx[8]={ 0,0,1,-1, 1,-1, 1,-1 }
int dy[8]={1,-1,0,0, 1, 1,-1,-1 }
```

8) DSU - Disjoint set Union
- Union And

Uses → Cycle Detection [only in undirected]
→ Kruskal → Shortest Path Algo
→ Number of connected components
→ Number of nodes in a component using size

```
class DSU{
public:
  int n; // no. of vertex, components
vector<int> parent;
vector<int> size;
 DSU(int n)
 {
    this->n=n;
   for(int i=0; i<n; i++)
    {
       parent.push_back(i);
        // or parent[i]=i;
      size.push_back(1);
       // or size[i]=1;
   }
int findparent(int x)
{
   if(parent[x]==x) return x
  return parent[x]=findparent(parent[x]);
}
void unite(int x, int y)
{
  int px=findparent(x), py=findparent(y);
  if(px==py) return;
  if(size[py] < size[px]) swap(px,py);
  // as I want to make py parent of px
  // so it must have more nodes
    parent[px]=py;
     size[py]+=size[px];
     n--; // component ↓ with every unite
}
};
```

Path Compression

#1 Check if cycle is present

x=1, y=3


```
if(findparent(x)==findparent(y)
  // there would be cycle if
  // we unite
else
   unite(x,y)
```

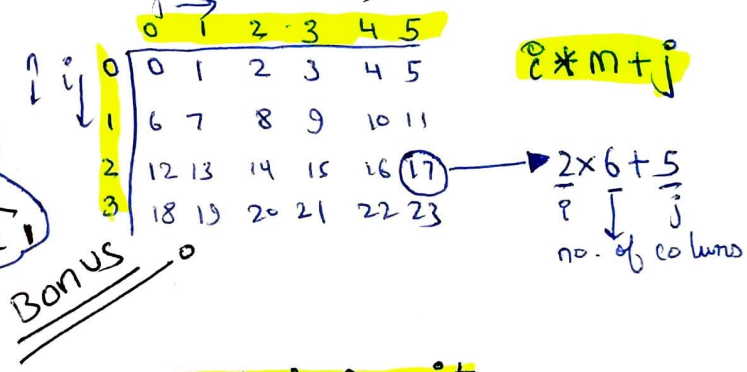
#2 No. of connected components

dsu.n

#3 No. of nodes in connected component

dsu.size[findparent(x)]

Static Graph	Dynamic Graph
DFS > DSU	DFS < DSU
O(E) O(ElogV)	O(q*E) O(q+ElogV)



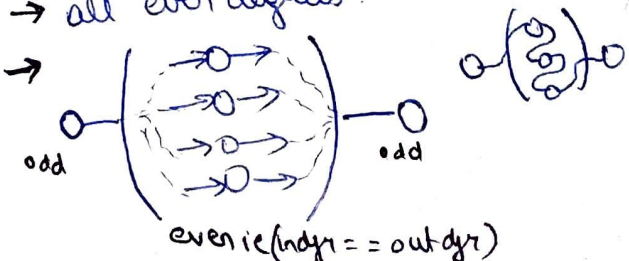
9) Euler Path & Circuit

Undirected & Directed [U & D]

Euler Path (EP) → • visit edge only once
• visit vertex more than once
• start & end at diff. vertex

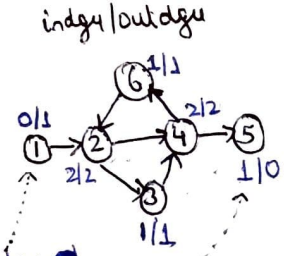
Euler Circuit (EC) → • same as euler
• start & end at same vertex

UEP → only 2 vertex odd degrees
UEC → all even degrees
DEP →



- for all nodes
  indgr == outdgr
  except 2 nodes

indgr/outdgr



1st node → indgr = outdgr + 1
2nd node → indgr + 1 = outdgr

DEC →

10) **KRUSKAL** → MST → Greedy
→ ElogV → graph/tree may or may not be connected
→ using DSU

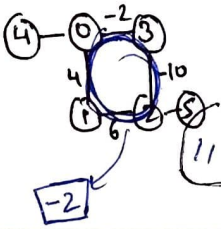
```
class DSU {
  -----(8)-----
};
int Kruskal(vector<vector<int>> &arr) {
  sort(arr.begin(), arr.end(), cmp);   // sort on weights
  DSU dsu(                             // number of vertices.
  int ans = 0;
  for(auto it : arr)
  {
if(findparent(it[0]) != findparent(it[1]))   // check cycle
    {
      ans += it[2];   // add weight
      unite(it[0], it[1]);
    }
  }
  return ans;
}
```

11) **Bellman ford** → Single Source shortest path
→ O(V×E) → `vector<int> path(v, INT_MAX); path[src] = 0;`

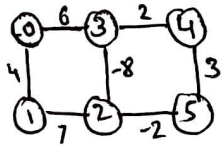
```
for(int j=1; j<=V; j++){
  for(int i=0; i<E; i++)
  {
    int u = arr[i][0], v = arr[i][1], wt = arr[i][2];
    if(path[u] == INT_MAX) continue;
    path[v] = min(path[v], path[u] + wt);
  }
}
```

So, to detect -ve wt cycle,
at **j = V** ie Vth time

```
if(path[v] > path[u] + wt)
{
  if(j == V) return 1;   // cycle of -ve weight, as v-1 time mein update hojana tha but ek or baar normalise hua ie there is -ve wt cycle.
  path[v] = path[u] + wt;
}
```



Djikstra	Bellman Ford
ElogV	EV
can't handle -ve wt.	can handle -ve wt
can't detect -ve cycle [no -ve edge LOL]	can detect -ve cycle

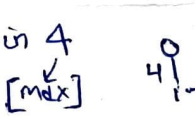


path

0	1	2	3	4
0	∞	∞	∞	∞
0	4	11	8	5
0	4	11	8	5
0	4	11	8	5
0	4	11	8	5

0-4 shortest path

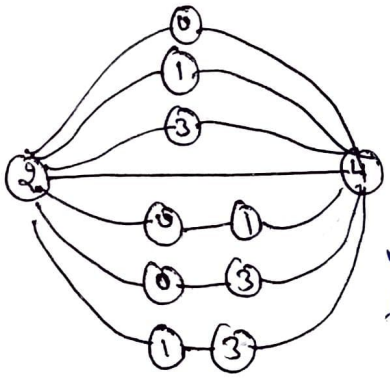
■ We'll get to '4' in 4 iterations [max]



- So for every vertex (v-1) would be req. as path length can't be greater than v-1 as we can't go in cycle (coz of no-ve cycle).
- for -ve wt. finding we run loop on edges once more & if they are relaxed ie there is cycle, as done above.

12) **Floyd Warshall**
→ All pair shortest path
→ O(V³)
→ adjacency matrix is helpful here

Option → Path from [2-4]



minimum we can get path from anywhere.

(1,2) ⇒ min((1,2), (1-0)+(0-2))

via

(3,2) ⇒ min((3,2), (3-1)+(1-2))

(1-2) (1-0-2)

(3-2) ≡ (3-1-0-2)

```
for(int k=0; k<V; k++)
{
    for(int i=0; i<V; i++)
    {
        for(int j=0; j<V; j++)
        {
if(i==j or i==k or j==k) continue;
if(arr[i][k]==∞ or arr[k][j]==∞) continue;
arr[i][j] = min(arr[i][j], arr[i][k]+arr[k][j]);
        }
    }
}
```

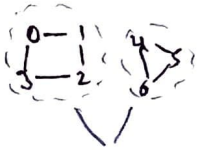
13) Strongly Connected Components

#Kosaraju Algorithm#

Every vertex is reachable from every other vertex.

Undirected

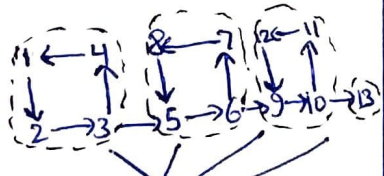
[Connected components]



2 comp.

Directed

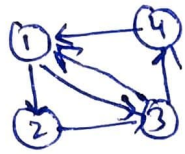
[Strongly connected comp.]



4

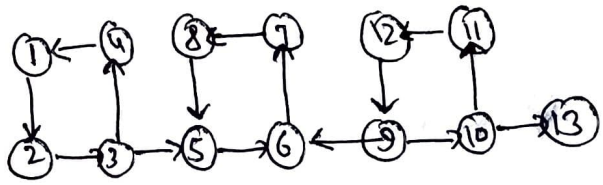
[3-5] we can't come back

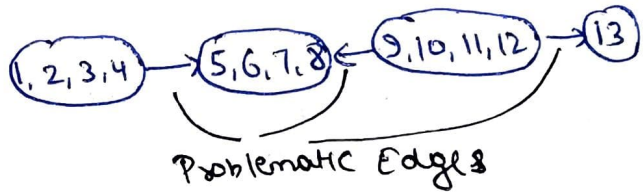
Note: Count of cycle is wrong approach



2 cycles but 1 strongly connected component.

[3-5] se wapas aa nahi possible so ek specific DFS lgega.





Problematic Edges

Kosaraju Algo.

Step1: Full stack in order of departure (DFS)

Step2: Reverse the graph

Step3: DFS on reversed graph ~~with~~ on unvisited stack elements.

```
void dfs(arr, vis, src, stack<int> &s) {
    vis[src]=1;
    for(int nbr: arr[src])
    {
      if(vis[nbr]==0)
        dfs(arr, vis, nbr, s);
    }
    s.push(src); // similar to topological sorting, order of departure
}

int kosaraju(arr, int v)
{
    stack<int> s;
    vector<bool> vis(v,0);
    for(i=0->v)
    {  if(vis[i]==0) dfs(arr, vis, i, s); }

    for(i=0->v){
        for(int nbr: arr[i]) rev[nbr].push_back(i);
    }                    // reverse graph
                         // to solve problem edge
    int ans=0;
    fill(vis.begin(), vis.end(), 0);
    stack<int> mouse;
```

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

s for above graph

```
while(!s.empty()){
  int x=s.top(); s.pop();
  if(vis[x]==0)
  {
    dfs(rev, vis, x, mouse);
    ans++;
  }
}
return ans;
}
```

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

ans=0

dfs(9) visited
ans++; 9,10,11,12

dfs(13) visited
ans++; 13

dfs(1) visited
ans++; 1,4,3,2

dfs(5) visited
ans++ 5,8,7,6

ans=4
↳zeher

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

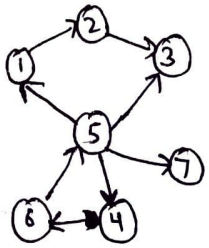
⇒ Sort cell nodes in order of finishing time
↳sim. to toposort

⇒ Transpose Graph

⇒ DFS acc. to finish time.

14) Mother Vertex

↳vertices that can reach all other vertices of graph



Step 1 of kosaraju.

```
void dfs(--)
{ . . . .
  . . .
  st.push(src);
}
```

run dfs(4)
↳if all get visited
this is answer

else no such vertex.
↳Thus return -1

Note: There can't be any answer in rest of stack as jaha wale vaha se stack bhute hain so top pe starting node hogi jisne baki sb pahuchne ki help ki

15) Articulation Point (TARJAN'S ALGO)

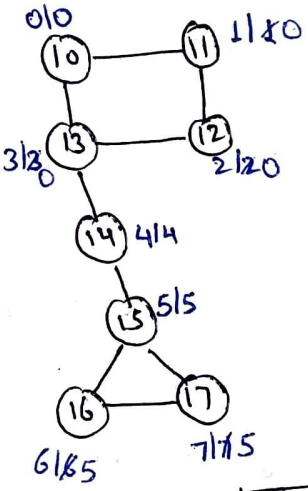
CUT VERTEX

↳Undirected graph
↳point removed disconnects graph
↳graph divides into 2 or more comp.

```
if(nbr==par[src]) continue;
if(dis[nbr]!=-1)
  low[src]=min(low[src], dis[nbr]);
else
{
  dfs(nbr);
  low[src]=min(low[src], low[nbr]);
}
```

Category(nbr)
parent → continue
vis
unvis

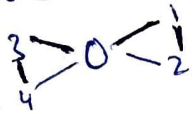
discovery / low



- we need to find the order of vertices from earliest to latest to detect back edges.
  ∴ we use timestamp to mark nodes with increasing value
  ∴ we do this by assigning discovery time.
- we need to maintain earliest possible node accessible for given node which will indicate if we have back edge

we reach our ancestor w/o parent

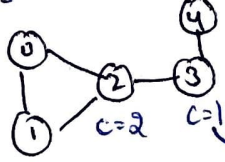
Now, for source vertex ⇒ edge case
2 children



⇒ if src vertex has more than one children than if removed divides graph into 2 or more parts, thus it's articulation point

Note: If you think for complete graph finding child > 1 would give AP ✗

eg.



AP: 2, 3
but this is AP

so children is edge case only for source vertex.

```
vector<int> disc, low, par, ap;

void dfs(arr, int src)
{
    static int time=0;
    dis[src] = low[src] = time++;
    int child=0;
    for(int nbr: arr[src])
    {
        if(nbr == par[src]) continue;
        if(disc[nbr] != -1)  // visited nbr
            low[src] = min(low[src], dis c[nbr]);
        else {
            child++;
            par[nbr] = src;
            dfs(arr, nbr);
            low[src] = min(low[src], low[nbr]);
        }
    if(par[src] == -1 and child > 1)  // source
            ap[src] = 1;              // vertex

    if(par[src] != -1 and low[nbr] >= disc[src])
            ap[src] = 1;  // more nbr more bad hi
                          // find karege ie they are dependent
                          // on me thus, on my removal
                          // graph disconnects.
    }
}

int main()
{
    par.resize(v, -1); disc.resize(v, -1); low.resize(v, -1); ap.resize(v, 0);

    arr[u].push_back(v);
    arr[v].push_back(u);

    dfs(arr, 0);

    for(int i=0; i<v; i++)
    {   if(ap[i]) cout<<i<<" ";
    }
```

→ all articulation point in graph

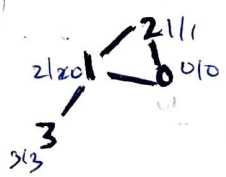
Note

#1 Shortest Path Algo → BFS edge count
- Prims, Kruskal → overall graph minimise
- Dijkstra (+ve edge wt.) (ElogV)
- Bellman Ford (ExV)
- Floyd Warshall (V3)

#2 Dijkstra, Bellman Ford, Floyd Warshall all work for directed & undirected.

bridge
→ that edge jisko remove krne se we get 2 diff components.

low[nbr] > disc[src]
↓
only change baki same.



ie edge mujhse (src) se bhi ho nhi milna chahiye uska bad vale find pe mile than it's bridge