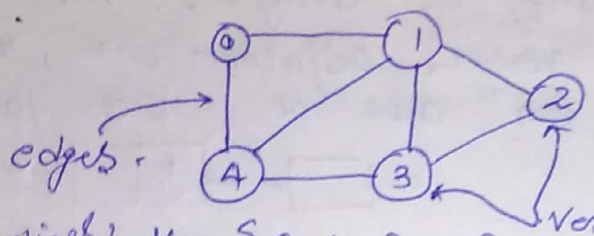
Graph Data Structure & Algorithms

A graph is non-linear data structure consisting of nodes (vertices) and edges (line).



Set of Nodes (vertices) $V = \{0, 1, 2, 3, 4\}$

Set of edges (lines) $E = \{01, 12, 13, 14, 23, 34, 04\}$

It is used to solve real-life problem.

Graph is data structure that consists of two components:-

A) Finite set of vertices know as nodes.

B) Finite set of ordered pair of form (u,v) called as edge. pair (u,v) indicates that is an edge from vertex u to vertex v. Edges may contain weight / value / cost.

∴ following two are most commonly used representation of graph:-

1) Adjacency matrix
2) Adjacency list

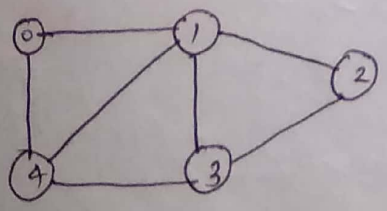
Adjacency matrix

It is 2D array of size V x V where V is number of vertices (nodes) in graph. let array be adj[][], the slot adj[i][j]=1 indicated that there is an edge from vertex i to vertex j.

Adjacency matrix for undirected graph is always symmetric.

Adjacency matrix is also used to represent weighted graphs. if adj[i][j] = w then there is an edge from vertex i to vertex j with weight w.

Undirected graph with 5 vertices

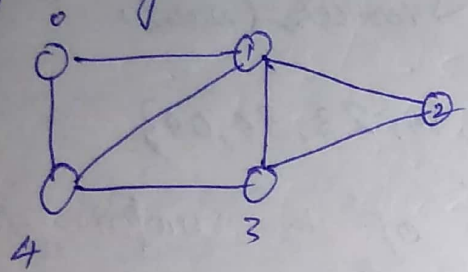


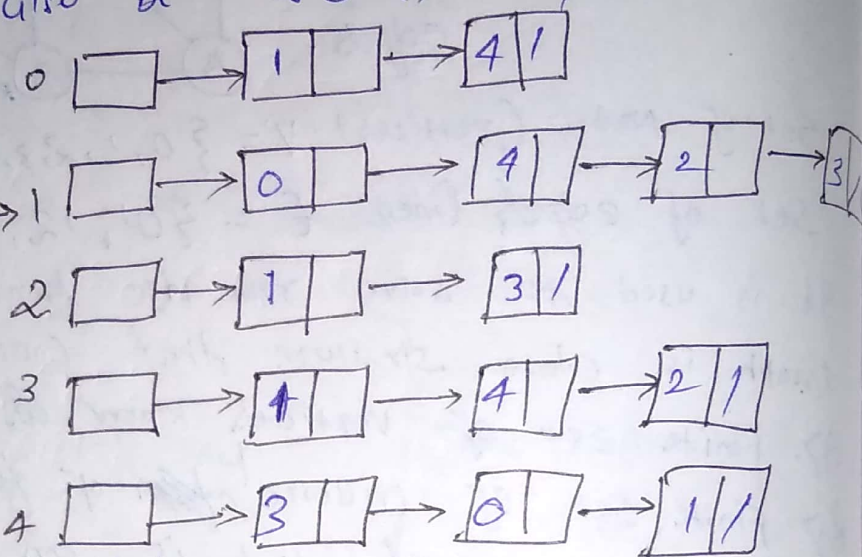
adjacency matrix

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency list

An array of list used. Size of array is equal to number of vertices. let array be array[]. An entry array[i] represents list of vertices adjacent to i^{th} vertices. This representation can also be used to represent a weighted graph.



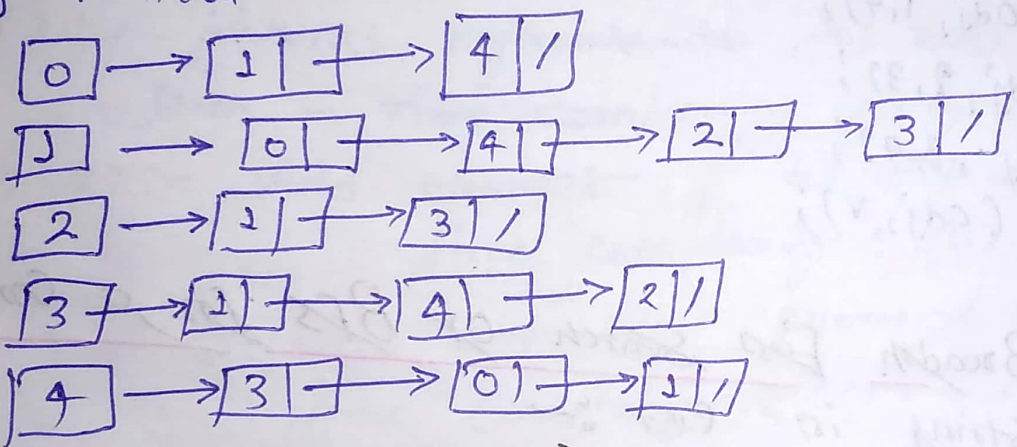


Note that in below implementation, we use dynamic arrays (vector in C++/ ArrayList in Java) to represent adjacency list.

vertex take $O(V^2)$ time.

Adjacency List:-

Array of list is used. Size of array equal to number of vertices.




```
#include <bits/stdc++.h>
using namespace std;
void addEdge (vector<int> adj[], int u, int v)
{ adj[u].push_back(v);
  adj[v].push_back(u);
}
```

```
void printGraph(vector<int> adj[], int V)
{
    for (int v=0; v<V; v++)
    {
        for (int i=0; i<adj[v].size(); i++)
            { cout << "->" << adj[v][i];
            }
        cout << endl;
    }
}
int main()
{
    int V=5;
    vector<int> adj[V];
    addEdge(adj, 0,1);
    addEdge(adj, 0,4);
    //  (adj, 1,2);
    //  (adj, 1,3);
    //  (adj, 1,4);
    //  (adj, 2,3);
    //  (adj, 3,4);
    printGraph(adj, V);
    return 0;
}
```

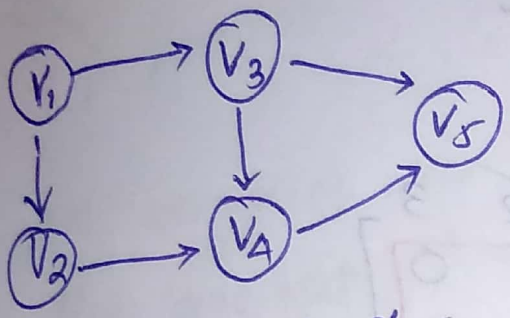
Output:-

->1->4
->0->2->3->4
->1->3
->1->2->4
->0->1->3

Breadth First

Introduction to Graph

- undirected graph Eg:- Social network (friend)
- directed " Eg:- World wide web (page 1 has link to page 2).



Undirected graph

Outdegree (V_3) = 2

↓

No. of edge outgoing.

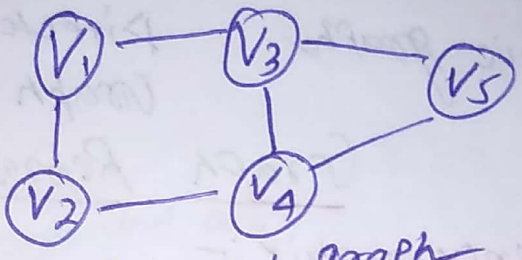
indegree (V_3) = 1.

Sum of indegree = |E|

Sum of Outdegree = |E|

maxm no. of edges = $^{V}C_2$

$= |V| * (|V|-1)$



Directed graph

degree (V_3) = 3

↓

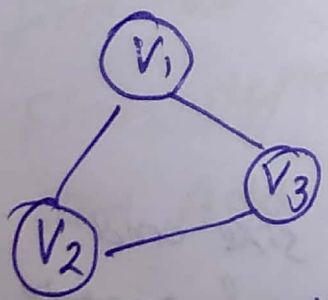
No. of edge connected to this vertices.

Sum of degree = 2 |E|

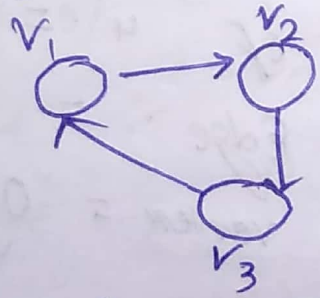
maxm no. of edges = $^{V}C_2$

$= \frac{|V|(|V|-1)}{2}$

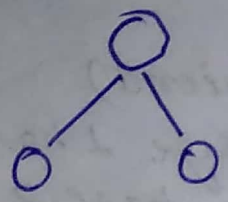
Cyclic :- There exists a walk that begins and ends at same vertex.



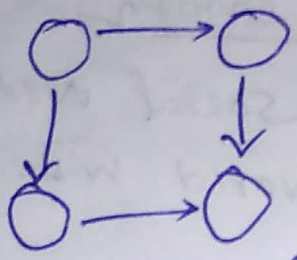
Cyclic undirected



directed cyclic graph



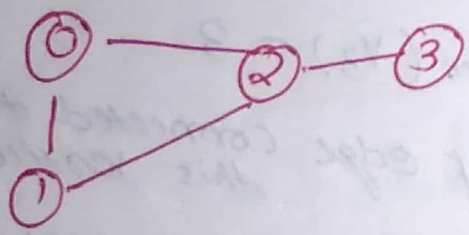
Undirected Acyclic graph



Directed Acyclic Graph (DAG)

Graph Representation

Adjacency matrix :-



	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

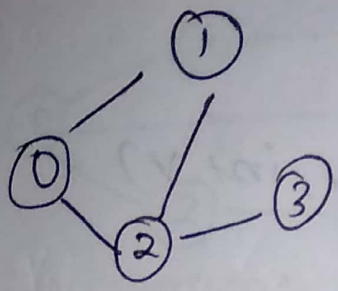
Size of matrix = $|V| \times |V|$
↳ No. of vertices

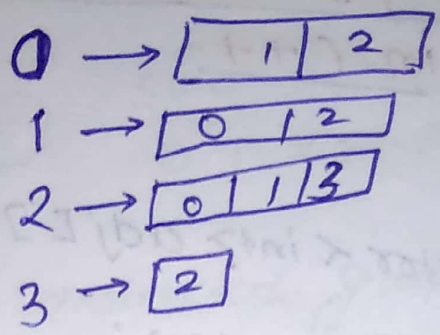
For undirected graph it is symmetric matrix.

matrix[i][j] = 1 → if there is an edge from i to j.
0 → otherwise

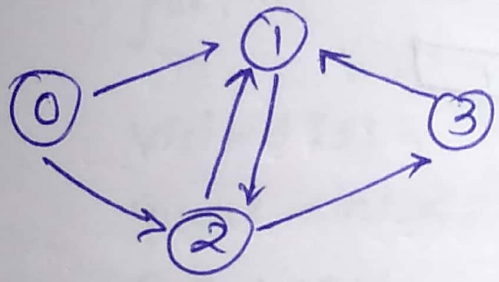
Operations :- Check if u and v are adjacent = O(1)
Find all vertices adjacent to u = O(V)
Find degree of u = O(V)
Add/remove an Edge = O(1)
Add/remove a vertex = $O(V^2)$
↳ Reallocate size and Copy previous matrix value.

Adjacency List :-

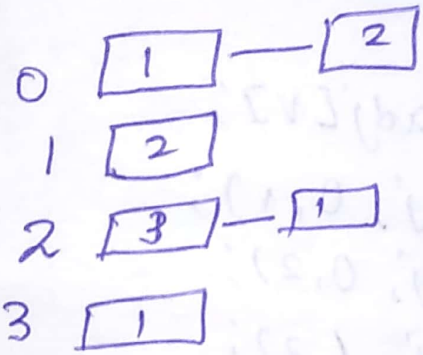




An array of list where list are properly represented as
i) Dynamic sized arrays
ii) Linked Lists.



Directed graph

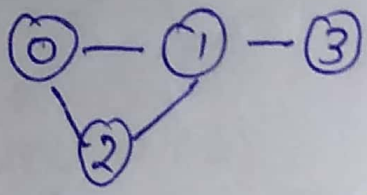


Space :- $O(V+E)$ → for directed
$O(V+2E)$ → for undirected

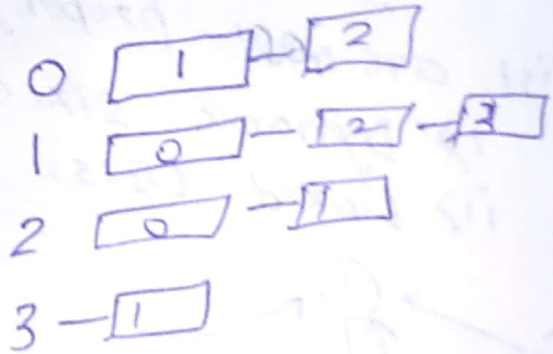
Operations :-

Check if there is an edge from u to v : $O(V)$
Find all adjacent of u :- $O(degree(u))$
Find degree of u :- $O(1)$
~~Find~~ Add an edge :- $O(1)$
Remove an Edge :- $O(V)$

Graph Adjacency list representation in C++



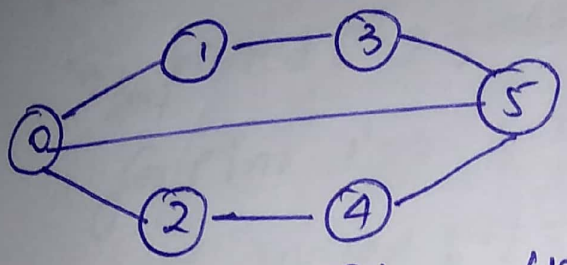

```
Void addEdge ( vector<int> adj[], int u, int v)
  { adj[u].push_back(v);
    adj[v].push_back(u);
  }
int main ()
{ int V = 4;
  vector<int> adj[V];
  addEdge (adj, 0, 1);
  addEdge (adj, 0, 2);
  addEdge (adj, 1, 2);
  addEdge (adj, 1, 3);
  return 0;
}
```



Comparision of adjacency list and matrix

	list (used for sparsh)	matrix (Dense)
memory	O(V+E)	O(V x V)
Check if there is an edge from u to v	O(V)	O(1)
Find all adjacent of u	O(degree(u))	O(V)
Add an Edge	O(1)	O(1)
Remove an Edge	O(V)	O(1)

Breadth First Search (BFS)



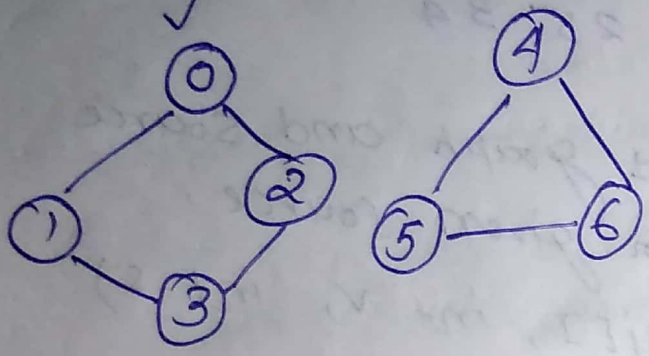
S=0

0 1 2 5 3 4

First version:- Given undirected graph and source vertex 's' print BFS from given source.

```
Void BFS (vector<int> adj[], int v, int s)
{ bool visited[v];
  for(int i=0 ; i<v; i++)
        Visited[i] = false;
  visited[s] = true;
  queue<int> q;
  q.push(s);
  while (q.emptyc)=false)
  { int u=q.fronte);
    q.pop();
    Cout<<u<<' ';
    for( vector<int>::iterator it = adj[u].begine);
                  it!=adj[u].ende) ; it++)
      { if (visited[*it] == false)
        { q.push (*it);
          visited[* it] = true;
        }
      }
  }
}
```

Second Version :- No source Given and graph may be disconnected.

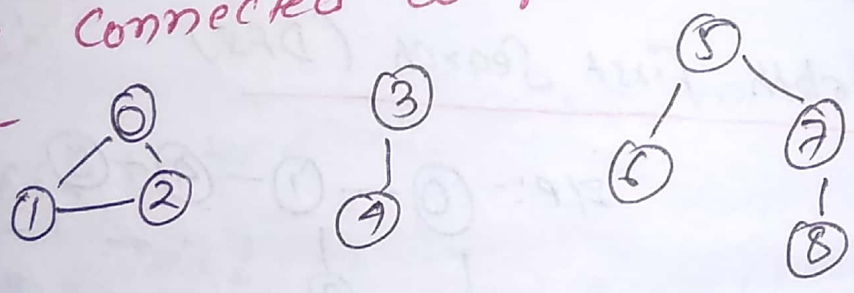



```
Void Bfs ( vector<int> adj[ ], int v, int s, bool
                                        visited[ ])
{ queue<int> q;
  q.push(s);
  visited[s] = true;
  while ( q.empty() == false)
    { int x = q.front();
      q.pop();
      cout << x << ' ';
      for( vector<int>:: iterator it = adj[x].begin();
                        it != adj[x].end(); it++)
        {  if ( visited[*it] == false)
           { q.push(*it);
             visited[*it] = true;
           }
        }
    }
}
```

```
Void BfsDfn{ vector<int> adj[], int v)
{ bool visited[v];
for(int i=0 ; i<v; i++)
    visited[i] = false;
for(int i=0; i<v ; i++)
    if( visited[i] == false)
        Bfs(adj, v, i, visited);

}
```

time complexity O(V+E);

Counting Connected compent in an undirected graph :-



Ans :- 3.

In bfsDfn :- change :-

```
int count=0;
for(int i=0; i<v; i++)
{   if( visited[i] == false)
    { count++;
      Bfs(adj, v, i, visited);
    }
}
Cout<< count<<endl;
```

Application of Bfs

i) Shortest path in ~~undirected~~ Unweighted graph

ii) Crawlers in search Engine.

iii) peer to peer network

iv) social networking search.

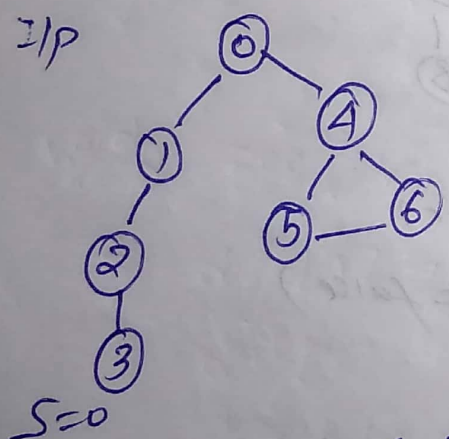
v) In Garbage Collection (cheney's Algorithm)

vi) Cycle detection. (both bfs and DFS)

vii) ford fulkerson Algorithm

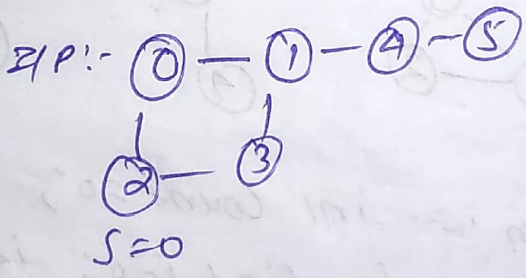
viii) Brodcasting.

Depth First Search (DFS)



S=0

O/p:- 0 1 2 3 4 5 6



S=0

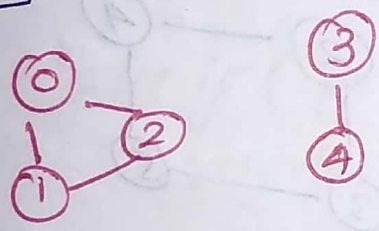
O/p:- 0 1 3 2 4 5

Case:-1 Simple undirected and connected graph

```
void dfsRec( vector<int> adj[], int v, int s, bool visited [])
{ visited [s] = true;
  cout << s << ' ';
  for(int u:adj[s])
```

```
        if (visited[u] == false)
            dfsRec(adj, v, s, visited);
    }
void dfs(vector<int> adj[], int v, int s)
{
    bool visited[v];
    memset(visited, false, sizeof(visited));
    dfsrec(adj, v, s, visited);
}
```

Case-2 disconnected graph




```
Void dfs(vector<int> adj[], int v)
{   bool visited[v];
    memset(visited, false, sizeof(visited));
    for(int i=0; i<v; i++)
        if(visited[i]==false)
            dfsrec(adj, v, i, visited);
}
```

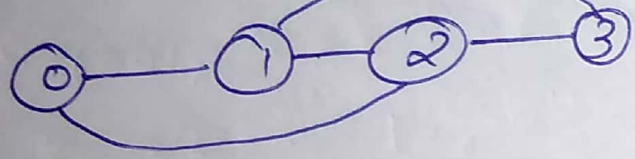
Time Complexity :- O(V+E)

Application of DFS

i) Cycle detection (DFS and BFS)
ii) topological sorting (make file utility)
iii) Strongly connected components
iv) Solving maze and similar puzzle
v) Path Finding → min no. of edge

Sortest path in Unweighted graph (BFS)

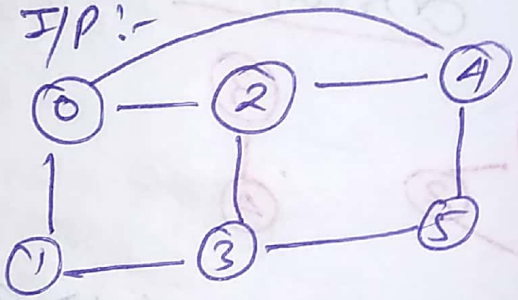
I/P



Source = 0

O/P :- 0 1 1 2

I/P :-

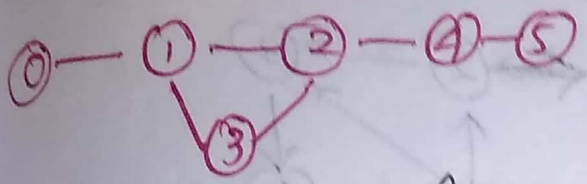


Source = 0

O/P :- 0 1 1 2 1 2

```
void BFS ( vector<int> adj[], int V, int S)
                                    visited[],
{ dist[V] = { int_MAX };
  dist[S] = 0; visited[S] = true;
  q.push(S);
  while ( q.empty() == false )
  { int u = q.front();
    q.pop();
    for ( int x : adj[u] )
      { if ( visited[x] == false )
        { visited[x] = true; q.push(x);
          dist[x] = dist[u] + 1;
} } }
```

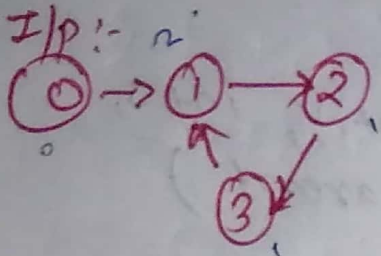
Detect cycle in an Undirected Graph




```
bool DfsRec( vector<int> adj[], int v, int s,
             int visited, int parent )
{ visited[s] = true;
  for (int x : adj[s])
  { if ( visited[x] == false )
        if ( DfsRec(adj, v, u, visited, s) == true)
            return true;
    else if ( parent != x)
        return true;

  }
  return false;
}
bool DFS ( vector<int> adj[], int v )
{bool visited[v] = {false};
  for(int i=0 ; i< v; i++)
      if (visited[i] == false)
          if (DFS( adj, v, i, visited, -1) == true)
              return true;

  return false;
}
```

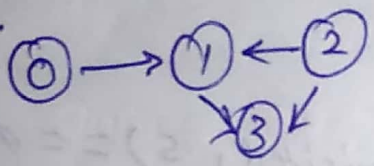
Detect cycle in undirected graph (DFS)

I/P :-



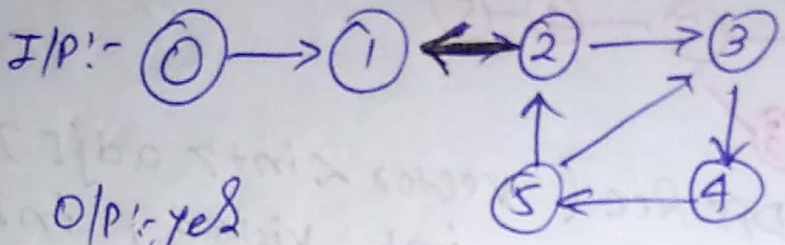
O/P :- yes

I/P :-



O/P :- NO

I/P :-



O/P :- yes

```
bool
DFSRec(adj, s, visited[], recst[])
{   visited[s] = true;
    recst[s] = true;
    for (int x : adj[s])
    {   if ( visited[s] == false && dfsrec(adj, x, visited, recst)
                                    == true)
            return true;
        else if ( recst[x] == true)
            return true;
    }
    recst[s] = false;
    return false;
}
bool dfs (vector<int> adj[], int v)
{   bool visited[s] = {false};
    bool recst[s] = {false};
```

```
for(int i=0; i<v; i++)
    if (visited[i]==false && dfsrec(adj, i, v,
                                    visited, recst)
                                    ==true)
        return true;
return false
}
```

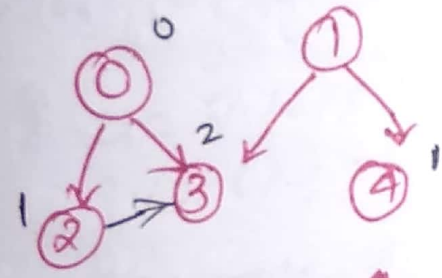
Topological Sorting (Kahn's Algorithm) (only for Non-cyclic)

I/P

Job 1 and 2 dependent job 0 so it must be printed after 0.

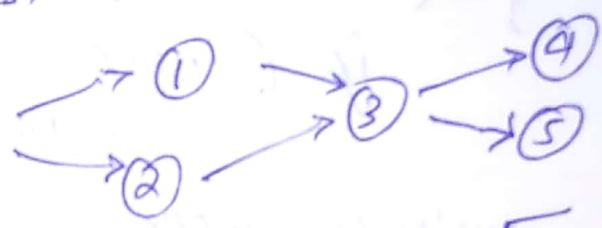
O/P 0 1 2

I/P



O/P :- 0 1 2 3 4

I/P :-



O/P :- 0 1 2 3 4 5

BFS Based Solution :-

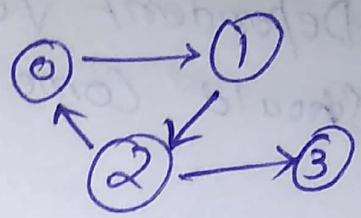
i) Store indegree of every index.
ii) Create a queue q.
iii) add all 0 indegree vertices to q.
④ while (q.empty == false)

```
{ u = q.front();
  q.pop();
  cout << u << ;
  for every v in u :-
      i) reduce indgree of v by 1.
      ii) if indgree of v becomes 0
          add v to q.
}
```

```
Void topological(vector<int> adj[], int V)
{ bool visited[V] = {false};
  int indegree[V];
  for(int i=0; i<V; i++)
     for(int x: adj[i])
        indegree[x]++;
  queue<int> q;
  for(int i=0; i<V; i++)
     if(indegree[i]==0)
        q.push(i);
  while(q.empty()==false)
  { int u = q.front();
    cout<<u<<' ';
    q.pop();
    for(int x: adj[u])
    { indegree[x]--;
      if(indegree[x]==0)
        q.push(x);
    }
  }
}
```

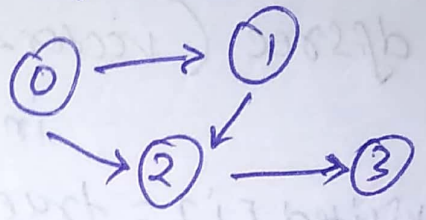
Cycle detection in Directed graph using Kahn's Algorithm

I/P :-



O/P :- yes

I/P



O/P :- NO

1) Store indegree of every vertex.
2) Create a queue q
3) Add all 0 indegree vertices to q
4) Count = 0
5) while (q.empty == false)

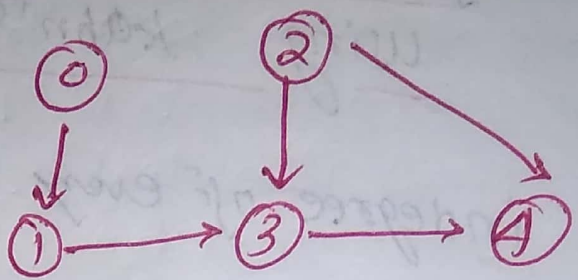
```
{ u = q.top()
  q.pop();
  for every v of u
      i) reduce indegree of v by 1
      ii) if indegree of v becomes 0
          push v to q.

  count++;
}
```

6) return (count != v);

}

Topological Sorting using DFS

Dependent job
Should come later.



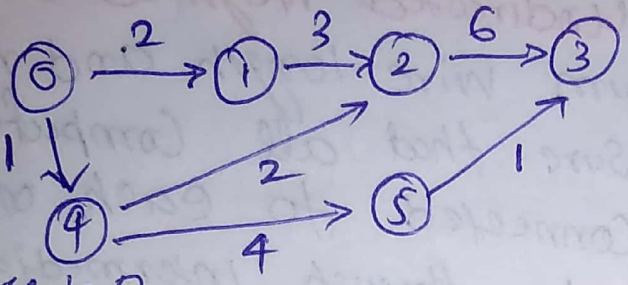
O/P :- 0 2 1 3 4

```
Void dfsrec (vector<int> adj[], int V, bool visited[],
                int i, stack<int> &S)
{ visited[i] = true;
  for(int x: adj[i])
      if (visited[x] == false)
          dfsrec(adj, V, visited, x, S);
  S.push(i);
}
Void dfs (vector<int> adj[], int V)
{ bool visited[V] = {false};
  stack<int> S;
  for(int i=0; i<V; i++)
      if (visited[i] == false)
          dfsrec(adj, V, visited, i, S);
  while (S.empty() == false)
  { cout<<S.top()<<" ";
    S.pop();
  }
}
```

Shortest path in Directed Acyclic Graph

weighted

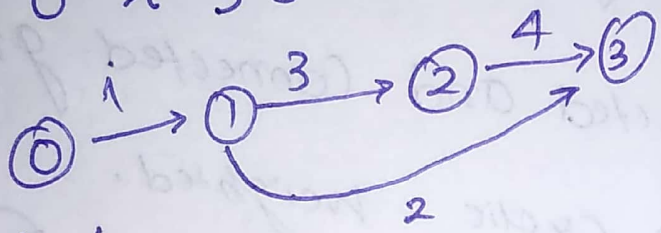
I/P :-



source :- 0

O/P :- 0 2 3 6 1 5

I/P :-



source :- 1

O/P :- INF 0 3 2

Using topological sort time complexity O(V+E)

1) Initilize dist[V] = {inf, INF, -- INF}

ii) dist[s] = 0;

iii) Find a topological sort of graph

iv) for every vertex u in topological sort.

a) for every vertex v in u

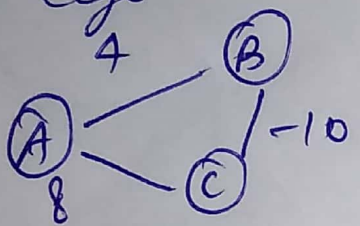
if dist[v] > dist[u] + weight(u,v)

dist[v] = dist[u] + weight(u,v);

Dijkstra's Algorithm for weighted Undirected graph

Shortest path

(1) Does not work for negative weight edge

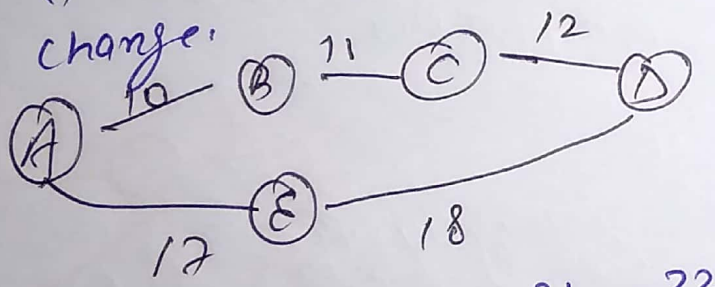


	A	B	C
A	0	4	8
B	0	4	-6
C	0	4	-6
	0	4	8

But ans :-

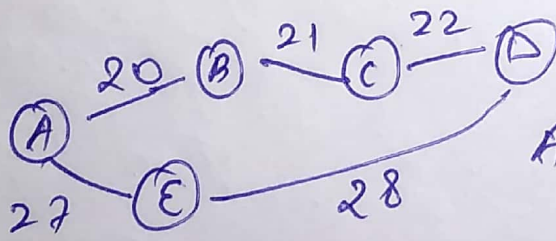
(2) Does shortest path change if add a weight to all edge of original (graph)

Ans :- change:



Shortest path from A to D :- 10+11+12 = 33

Add 10 to all



Shortest path A to D = 55

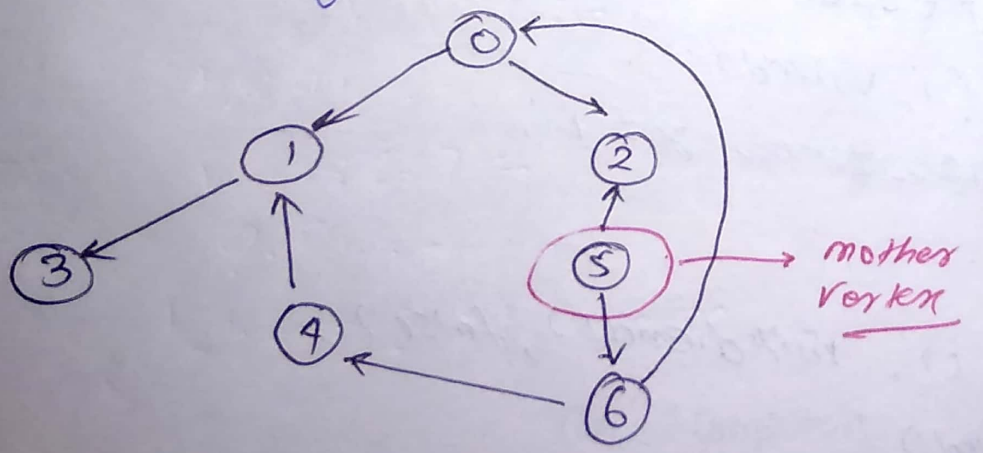
```
Void Graph::shortestPath(int src)
{ priority_queue< pair<int,int>, vector<pair<int,int>>,
                  greater<pair<int,int>>> pq;

vector<bool> visit(V, false);
vector<int> dist(V, INF);
pq.push(make_pair(0,src));
dist[src]=0;
```

```
while (pq.empty())
{ int u = pq.top().second;
    pq.pop();
    visit[u] = true;
  for (pair<int,int> p : adj[u])
      { if (visit[p.first]==false && dist[p.first] >
                                    dist[u]+p.second)
          { dist[p.first] = dist[u] + p.second;
            q.push(make_pair(dist[p.first], p.first));
          }
      }
}
Return dist;
}
```

Find a mother vertex in a Graph

→ A mother vertex in graph $G = (V, E)$ is vertex v such that all other vertices in G can be reached by a path from v.

This is for only directed - connected graph.



Naive approach :- We perform DFS / BFS on each vertices and check all vertices reach or not. Time Complexity $O(V(E+V))$ — for each vertices.

Better :-

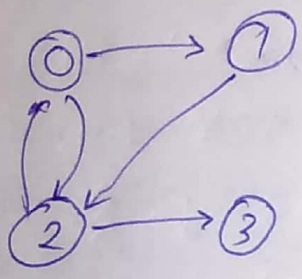
Mother vertices are always vertices of source Component in Connected ; Time :- $O(V+E) + O(V+E)$

```
Void Graph::DFSutil(int v, vector<bool> &visited)
{   visited[v] = true;
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); i++)
      { if(!visited[*i])
            DFSutil(*i, visited);
      }
}
int Graph::findMother()
{   vector<bool> visited(V, false);
                          size  initialisation
    int x=0; // for store last finished vertex (for mother vertex)
    for(int i=0; i<V; i++)
      { if(visited[i]==false)
          { DFSutil(i, visited);
            x = i;
          }
      }
    fill(visited.begin(), visited.end(), false);
    DFSutil(x, visited);
```

```
for (int i=0; i<V; i++)
  { if (visited[i]==false)
      return -1;
  }
return x;
}
```

Transitive Closure of Graph using DFS

The reach-ability matrix is called transitive closure of graph.



transitive closure :-

```
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
```

```
Void Graph::transitiveClosure()
{  bool tc[V][V];
   memset(tc, 0, sizeof(tc));
   for(int i=0; i<V; i++)
     { DFSUtil(i, tc); }
}
Void Graph::DFSUtil(int x, bool tc[][V])
{  tc[x][x] = 1;
   list<int>::iterator it
   for(it = adj[x].begin(); it!=adj[x].end(); it++)
     { if(tc[x][*it]==false)
         DFSUtil(x, tc, *it);
     }
}
```

Time complexity O(V²)

Count Pairs with given Sum

Given an arrays of integers, and a number 'sum', find number of pairs of integer in array whose sum is equal to 'Sum'.

Eg arr[] = {1,5,7,-1}
Sum = 6
Output :- 2 (1,5) and (7,-1)

arr[] = {1,5,7,-1,5}
Sum = 6
Output :- 3 (1,5) (1,5), (7,-1)

Simple solution :-

```
int getPairsCount (int arr[], int n, int sum)
{ int count=0;
  for(int i=0; i<n; i++)
     for(int j= i+1; j<n; i++)
        { if (arr[i] + arr[j] == sum)
             count++;
  return count;
}
```

Time :- O(n²)

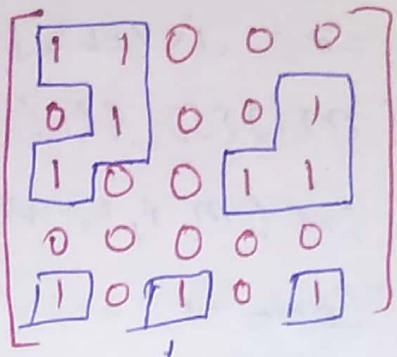
Better Solution :- O(n)

```
int getPairsCount (int arr[], int n, int sum)
{ unordered_map <int, int> mp;
  for(int i=0; i<n; i++)
     { mp[arr[i]]++; }
int count=0;
  for( int i=0 ; i<n; i++)
     { count += mp[sum-arr[i]];
       if (sum == 2 * arr[i])
          count--;
     }
return count/2;
}
```

Find number of islands using DFS

input :-

```
mat[][] = { { 1, 1, 0, 0, 0 },
            { 0, 1, 0, 0, 1 },
            { 1, 0, 0, 1, 1 },
            { 0, 0, 0, 0, 0 },
            { 1, 0, 1, 0, 1 } }
```



Output = 5

No. of Connected sub groups

```
#include <bits/stdc++.h>
using namespace std;
#define Row 5
#define Col 5
int issafe(int m[][Col], int row, int col, bool visit[][Col])
{ return (row >= 0) && (row < Row) && (col >= 0) && (col < Col)
    && m[row][col] && !visit[row][col];
}
void DFS(int m[][Col], int row, int col, bool visited[][Col])
{ visited[row][col] = true;
  static int rowNbr[] = {-1, -1, -1, 0, 0, 1, 1, 1};
  static int colNbr[] = {-1, 0, 1, -1, 1, -1, 0, 1};
  for (int k=0; k<8; k++)
    { if (issafe(m, row+rowNbr[k], col+colNbr[k], visited))
      DFS(m, row+rowNbr[k], col+colNbr[k], visited);
    }
}
int CountIslands(int m[][Col])
{ bool visited[Row][col];
  memset(visited, 0, sizeof(visited));
```

```
int count=0;
for(int i=0; i<Row; i++)
  for(int j=0; j<COL; j++)
    if (m[i][j] && !visited[i][j])
    { DFS(m, i, j, visited);
      count++;
    }
  return count;
}
int main()
{ int m[][COL] = { {1, 1, 0, 0, 0},
                   {0, 1, 0, 0, 1},
                   {1, 0, 0, 1, 1},
                   {0, 0, 0, 0, 0},
                   {1, 0, 1, 0, 1} };
  Cout << countIslands(m);
  return 0;
}
```

Sanake and Ladder Problem

use Breadth first serach to get minm no. of dice through.

```
#include <bits/stdc++.h>
Using namespace std;
#define siz 30
class Cell
  { int v;
    int distance;
  };
int Solve(int move[])
{ queue <Cell> q;
  Cell x, y;
```

```
x.v=0;  x.distance=0;
q.push(x);
while(q.size())
{ y=q.front();
  if(y.v==siz-1)
      break;
  q.pop();
  for(int i=(y.v+1); i<=(y.v+6) && i<siz; i++)
      { cell z;
        z.distance = y.distance+1;
        if(move[i]!=-1)
            z.v = move[i];
        else
            z.v=i;
        q.push(z);
      }
}
return y.distance;
}
int main()
{ int t; cin>>t;
  while(t--)
  { int move[siz]; int n;
    cin>>n;
    memset(move, -1, sizeof(move));
    int x,y;
    for(int i=0; i<n; i++)
      { cin>>x>>y;
        move[x-1]=y-1;
      }
    cout<<solve(move)<<endl;
  }
  return 0;
}
```

Count of Inversion of Array

Two element of array form an inversion if $a[i] > a[j]$ $i < j$.

input :- 2 4 1 3 5

output :- 3. (2,1), (4,1), (4,3)

Simple :- $O(n^2)$

```
int count=0;
for (int i=0; i<n; i++)
    for (int j=i+1; j<n; j++)
        if (arr[i] > arr[j])
            count++;
```

Best approach $O(n \log n)$ using merge sort :-

```
#include<bits/stdc++.h>
using namespace std;
int merge ( int arr[], int l, int mid, int r)
{ int n= mid-l+1; int m=r-mid;
  int x[n], y[m];
  for (int i=0; i<n ; i++)
      x[i] = arr[l+i] ;
  for (int i=0; i<m; i++)
      y[i] = arr[mid+i+1];
  int count=0;
  int i=0, j=0, k=l;
  while (i<n && j<m)
  { if (x[i] > y[j])
      { count +=(n-i);
        arr[k] = y[j];
        k++;  j++;
      }
    else arr[k++] = y[i++];
  }
```

```
    while (i<m)
      { arr[k++] = x[i++];
      }
    while (j<m)
     { arr[k++] = y[j++];
     }
    return count;
}   int solve(int arr[], int l, int r)
    { if (l>=r)
        return 0;
      int mid = (l+r)/2; int count=0;
      int count = solve(arr, l, mid);
      count += solve(arr, mid+1, r); count += merge(arr, l, mid+1, r); retu
      return count;
    }
int main()
{   int n; cin>>n; int arr[n];
    for(int i=0; i<n; i++)
        cin >> arr[i];
    cout << solve(arr, 0, n-1) << endl;
}
```

let

then,

int

{

Given

Small

be cut

power