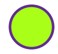
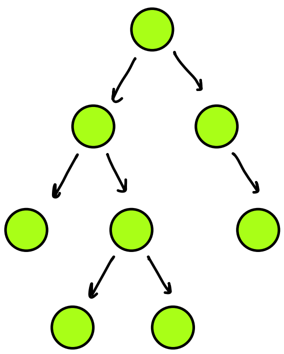
Introduction to Graphs

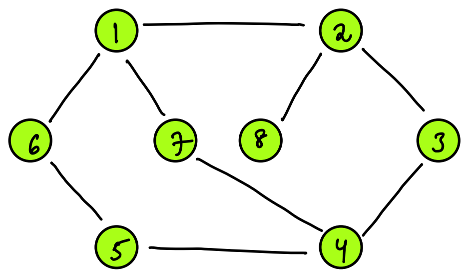


Graph: It is bunch of nodes connected via Edges

Ex1: Tree



Graph



Main difference between tree & Graphs

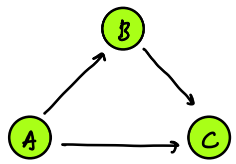
1) Tree is hirarchical data Structure, unlike graphs

2) No: of Edges in N node Tree = N-1

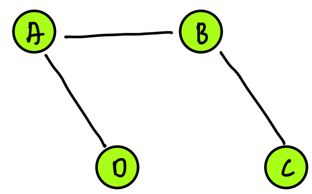
Classification of Graphs

Case - I :

Directed Graph



Undirect graph



Facebook:

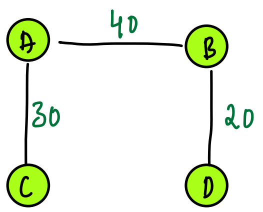
A — B

Instagram:

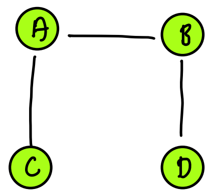
A → B

Case - II :

Weighted Graph

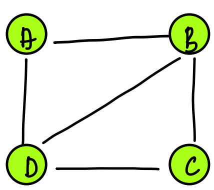


Un Weighted Graphs

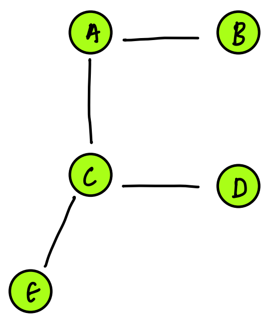


Case - III

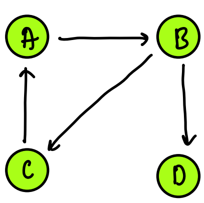
undirected cyclic graph



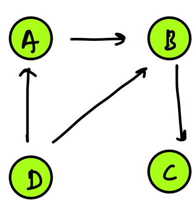
undirected acyclic graph



directed cycle graph



directed acyclic graph



→ a graph can be combination of multiple things

→ Type of graph is always in Question.

How Graph is Given as Input?

↳ collection nodes connected with Edges

Q1) Given an ~~u~~ndirected graph with N Nodes & M Edges

Inputformat:

1st line,

#Nodes #Edges

N E

Followed E lines

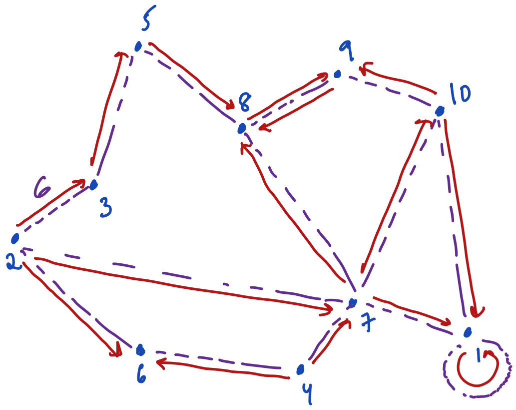
u v w

indicating Edge from u — v

u & v are nodes

w indicate weight of Edge between u & v

N	E	
10	14	
u	v	w
2 —	3	6
4 —	7	.
8 —	9	.
2 —	7	.
7 —	8	
10 —	1	
4 —	6	
5 —	8	
2 —	6	
10 —	9	
7 —	10	
3 —	5	
7 —	1	
1 —	1	



In Question:

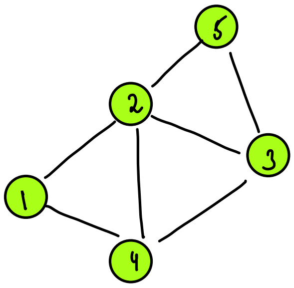
a) Undirected vs Directed

b) Weighted or UnWeighted

Information which won't be given

a) Cyclic or Acyclic

Storing a graph



Input:

N	E
5	7
1	4 ✓
2	5 ✓
3	2 ✓
4	3 ✓
2	4 ✓
3	5
1	2

App: 1 → Adj Matrix :

int mat[6][6] ⇒ 1 based index

	0	1	2	3	4	5
0						
1		0	1	0	1	0
2		1	0	1	1	1
3		0	1	0	1	1
4		1	1	1	0	0
5		0	1	1	0	0

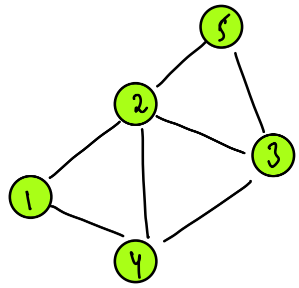
In general : N Nodes → int g[N+1][N+1] → TC: O(E) : edges

→ SC: O(N²) : Space wastage

Classification : u ——— v

	unweighted	weighted, u, v, w
undirected	g[u][v] = 1 g[v][u] = 1	g[u][v] = w : g[v][u] = w
directed	g[u][v] = 1	g[u][v] = w

In general weights are are non-zero



10:20 → 10:40

Input:

N	E
5	6
1	4
2	5
3	2
4	3
2	4
3	5

Way: 2 Adj list

list<int> g[6] → array of lists / lists of lists → list<list<int>>

In your language of choice plean check it

g[6]

- g[0]
- g[1] → 4
- g[2] → 5, 3, 4
- g[3] → 2, 4, 5
- g[4] → 1, 3, 2
- g[5] → 2, 3

Eg: list<pair<int, int>> g[4]

N E → directed graph

g[4]

N	E	
3	5	
1	2	5
1	3	2
2	3	4
2	1	6
3	1	7

- g[0]
- g[1] → <2, 5>, <3, 2>
- g[2] → <3, 4> <1, 6>
- g[3] → <1, 7>

Classification: (u) ——— (v) → TC: O(E) SC: O(E)

unweight graph ⇒ N Node : list<int> g[N+1]

Weighted graph ⇒ N Node : list<pair<int, int>> g[N+1]

	unweighted	weighted, u, v, w
undirected	g[u].add(v) g[v].add(u)	g[u].add({v, w}) g[v].add({u, w})
directed	g[u].add(v)	g[u].add({v, w})

vertex v

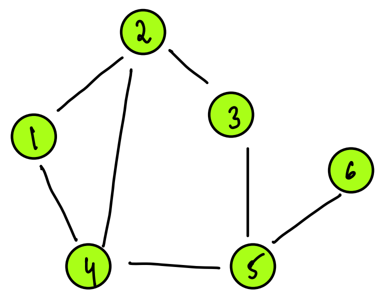
weight from u-v

Note: For every edge we do 1 or 2 insertion based on graph

1Q: Given a undirected graph & Source Node & Dest Node
Check if node can be visited from Source Node?

S		D	
1	→	6	{return True}

Graph:



Input:

N	E
6	7

	u[]	v[]
0	1	2
1	1	4
2	2	4
3	2	3
4	3	5
5	5	6
6	4	5

list<int> g[7]

g[0]
g[1] → 2, 4
g[2] → 1, 4, 3
g[3] → 2, 5
g[4] → 1, 2, 5
g[5] → 3, 6, 4
g[6] → 5

S = 1, D = 6

Obs: a node add only once

bool vis[7] = {F}

0	1	2	3	4	5	6
F	~~F~~	~~F~~	~~F~~	~~F~~	~~F~~	~~F~~
	T	T	T	T	T	T

~~1~~	~~2~~	~~4~~	~~3~~	~~5~~	~~6~~

Operations
- → delete at Start
- → insert at back

Queue is used

Idea:

Repeat till Queue Empty
Step1: Get front node from node & remove it
Step2: Go to adj list of node, and add all unvisited neighbours into Queue & make it as visited

// N → Nodes, E → Edges $\underbrace{u[\,],\ v[\,]}$: Edge Connections →

```
bool BFS( int N, int E, int u[], int v[], int s, int d){

    list<int> g[N+1] // Creating adj list            TC: O(E)
    for(int i=0; i<E; i++){                          SC: O(E)
        // u[i], v[i], i^th edge from u[i] → v[i]
        g[u[i]].add(v[i])
        g[v[i]].add(u[i])   → If unidirected graph   TC: O(E)
                                                     SC: O(N+E)
    }                                                  E>>N
                                                     → O(E)
    Queue<int> q; q.insert(s)
    bool vis[N+1] = F; vis[s] = True                 TC: O(E)
    int lev[N+1] = -1, lev[s] = 0                    SC: O(N)
    int par[N+1] = -1; par[s] = -1
    while(q.size()>0){
        // Step1: get front Node from q
        int cu = q.front(); q.delete() → // delete front

        // Step2: Traverse on adj list of cu
        for( int i=0; i< g[cu].size(); i++){
            int cv = g[cu][i]
            if( vis[cv] == false ){
                // not yet visited
                vis[cv] = True
                q.add(cv)
                lev[cv] = lev[cu]+1 // update level since node cu pushing node cv
                par[cv] = cu // updating parent
            }
        }
    }
    return vis[d], return lev[d],
}
```

Note: If cv == d, we can return true, here itself

// Say N nodes:

cu	g[cu]
1	g[1]
2	g[2]
3	g[3]
⋮	
N	g[N]

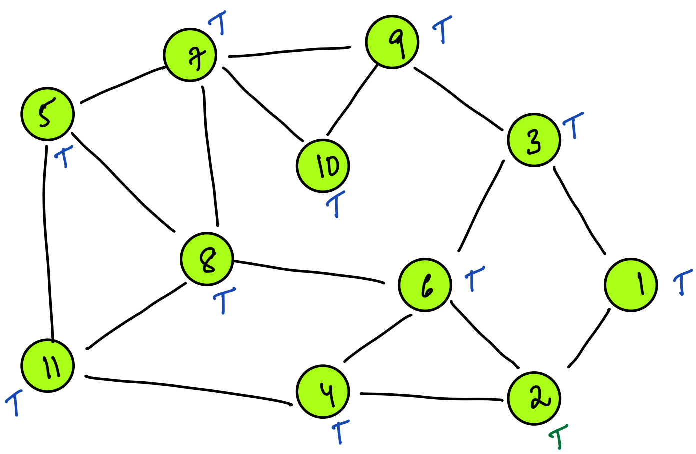
Sum of all this = $O(E)$

$2 \times$ [Total no: of Edges] ⇒ undirected

[Total no: of Edges] ⇒ directed

$g[0] + g[1] + \dots g[N]$ = Size of Adj list $= O(E)$

Tracing: S: 10 D: 2

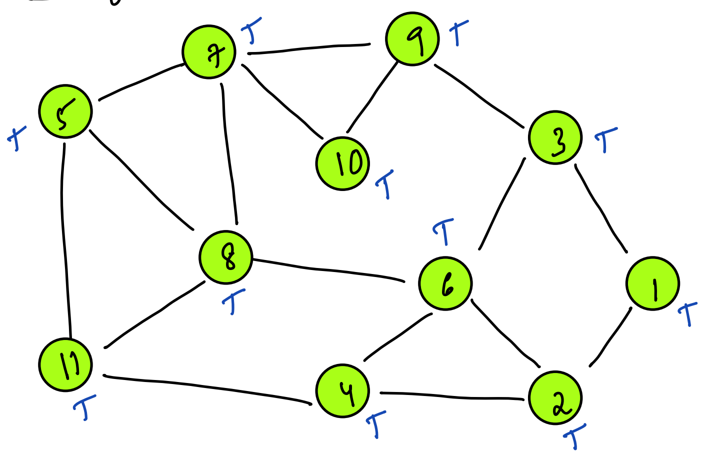


level	0	1	2	3	4
	10	9 7	3 8 5	1 6 11	2 4

Final obs:

BFS also gives you length of shortest path from source to all Nodes

Tracing: S: 10 D: 2



par[12] =

0	1	2	3	4	5	6	7	8	9	10	11
	3	1	9	11	7	8	10	7	10	-1	5

→ path from $\frac{S}{10} \to \frac{D}{11}$

par[11] par[5] par[7] par[10]

11 ⟶ 5 ⟶ 7 ⟶ 10 ⟶ -1

↳ shortest path S → D

→ path from $\frac{S}{10}$ $\frac{D}{4}$

par[4] par[11] par[5] par[7] par[10]

4 ⟶ 11 ⟶ 5 ⟶ 7 ⟶ 10 ⟶ -1

↳ shortest path S → D

// get S → D

$\left.\begin{array}{l}\text{1) Fill par[N+1]} \\ \text{2) list<int> path;} \\ \quad \text{while(d != -1) \{} \\ \qquad \text{path.add(d)} \\ \qquad \text{d = par[d]} \\ \quad \text{\}}\end{array}\right\}$ // shortest path from S → D