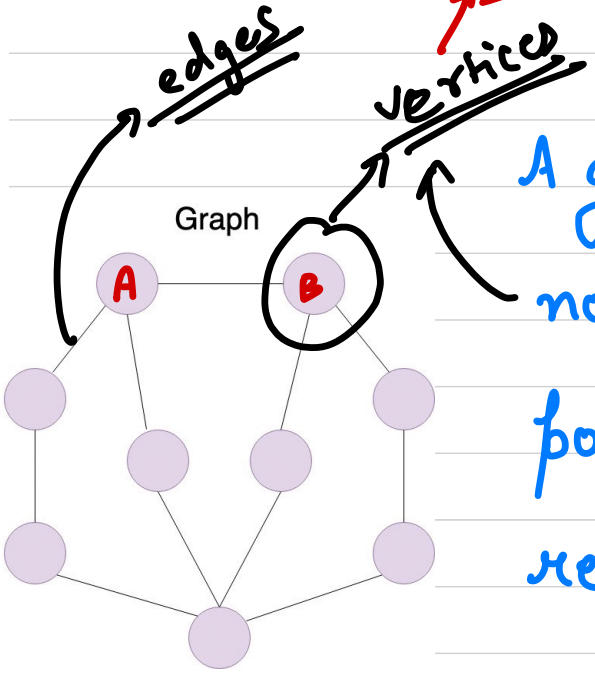
Graph (discrete maths)

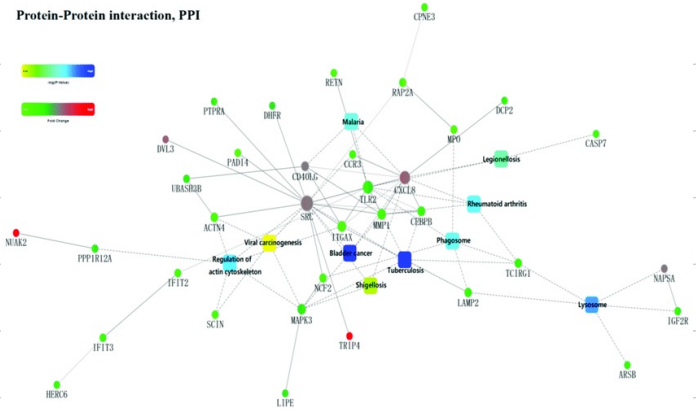
node → some data we will store



A graph is a collection of nodes where each node might point to other nodes. The nodes represent real life entities and are connected by edges representing relationship between the entities / node.

Social media

Applications Of Graphs



Protein - Protein Interaction

Google Maps

1) Computer Science
   - Network flow
   - Shortest path
   - Minimum Spanning Trees
2) Electrical Engg
   - Circuit Organization
3) Networking
   - Routing algorithms
   - Vertex coloring for freq assignment
4) Biology
   - Protein - Protein interaction
   - Metabolic Networks

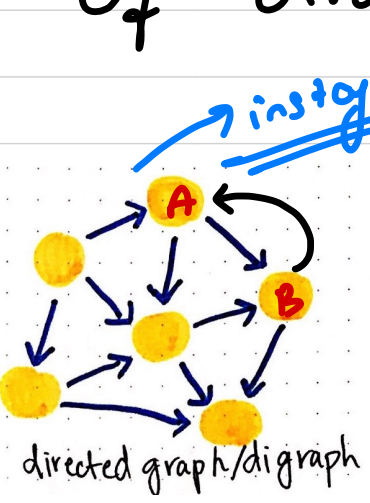
Types Of Graphs

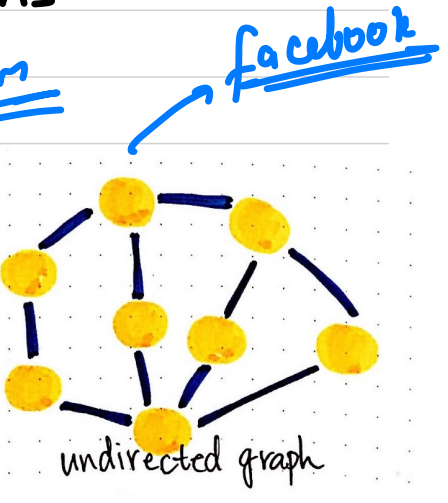
1) Undirected Graph

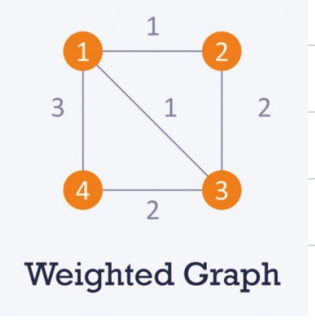
2) Directed Graph

3) Weighted Graph

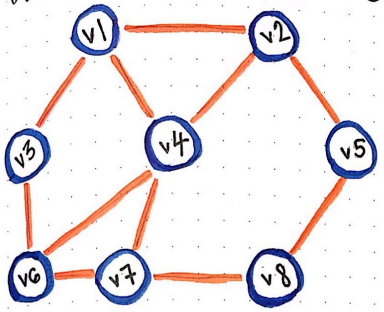
4) Unweighted graphs







(Formally) Defining a Graph



8 vertices/nodes

11 edges/links

$G = (V, E)$ → pair

$V = \{v1, v2, v3, v4, v5, v6, v7, v8\}$

set

$E = \{\{v1, v2\}, \{v1, v3\}, \{v1, v4\}, \{v2, v4\}, \{v2, v5\}, \{v3, v6\}, \{v4, v6\}, \{v4, v7\}, \{v5, v8\}, \{v6, v7\}, \{v7, v8\}\}$

set of pairs

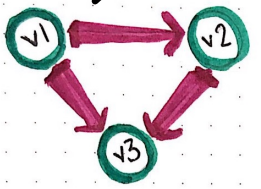
these edge definitions are unordered pairs!

→ $G = (V, E)$ is the formal mathematical notation for defining graphs.

→ A graph G is an ordered pair of a set V vertices and E, a set of edges.

→ An ordered pair is a pair of mathematical objects in which the order of objects in the pair matters.

directed graph



But what about a directed graph?

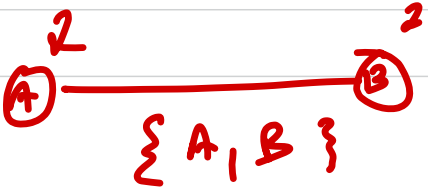
$G = (V, E)$ → how would our edge objects be different?

$V = \{v1, v2, v3\}$

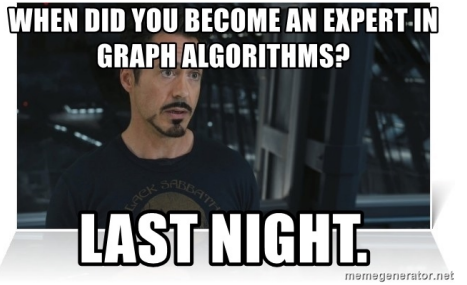
$E = \{(v1, v2), (v1, v3), (v2, v3)\}$

pair ordering matters

→ these edge definitions are ordered pairs, because direction matters!



Representation Of Graphs



1) Adjacency List → array of LL

2) Adjacency Matrix → 2D array

3) Adjacency Map → array of hashmap / hashmap of mapping

4) Incidence Matrix → 2D array

Adjacency List

[[ ]]

→ space optimum

→ add at head O(1)

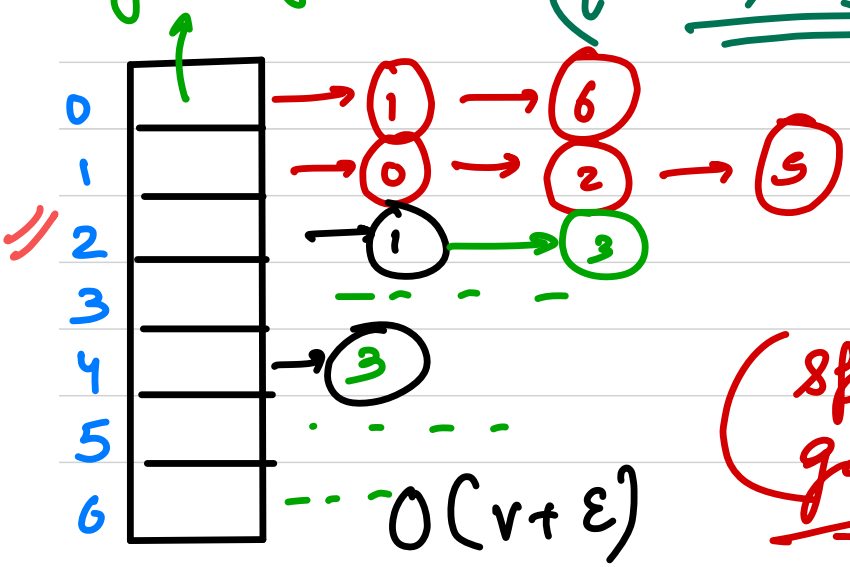
Vertices

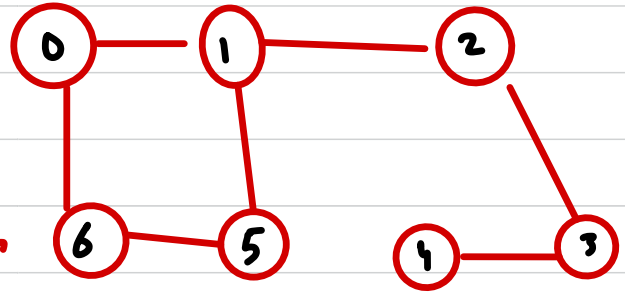
Array of linked list.

→ edges

{vertex, wt}

we store immediate neighbour of ith node





→ hashmap

Sparse graph

→ ll

→ bucket

Adjacency Matrix

simple

Dense graphs

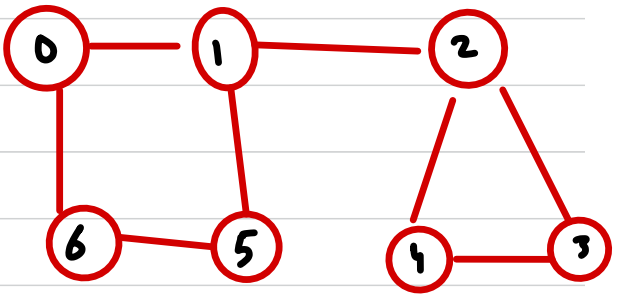
2d array

$$A_{ij} = \begin{cases} 1 & \text{there is an edge between } i \text{ \& } j \\ 0 & \text{else} \end{cases}$$

2 Q

A =

	0	1	2	3	4	5	6
0	0	1	0	0	0	0	1
1	1	0	1	0	0	1	0
2	0	1	0	1	1	0	0
3	0	0	1	0	1	0	0
4	0	0	1	1	0	0	0
5	0	1	0	0	0	0	1
6	1	0	0	0	0	1	



$O(V^2)$

$V \times V$

$V_1 — V_2$

m[v1][v2] = 1

2D matrix

Incidence Matrix

$M_{ij} = \begin{cases} 1 & \text{if } i^{th} \text{ vertex belongs to } j^{th} \text{ edge} \\ 0 & \text{else} \end{cases}$

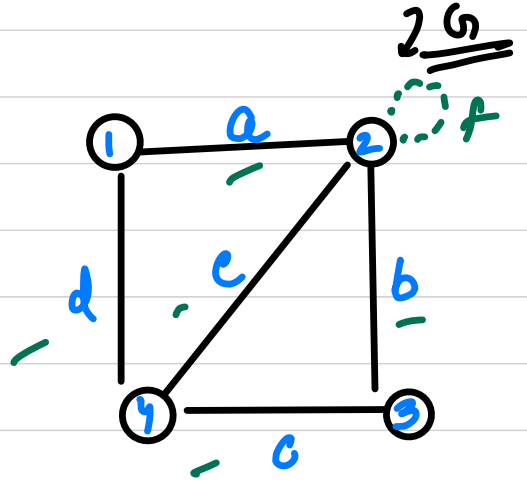
$M =$

	a	b	c	d	e
1	1	0	0	1	0
2	1	1	0	0	1
3	0	1	1	0	0
4	0	0	1	1	1

t

2 → column represent edges

rows represent vertices



row sum = degree

column sum = 2

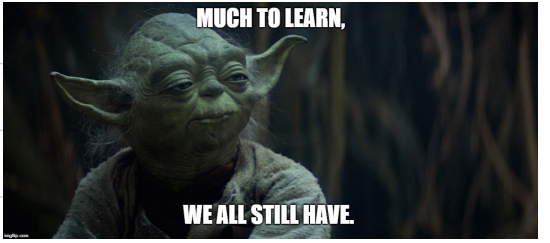
Let's Implement Graphs

→ Vectors / Array list

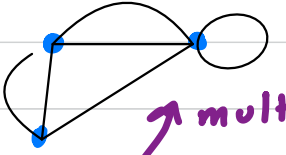
→ Linked List

→ Structs / Classes

Graph Terminologies



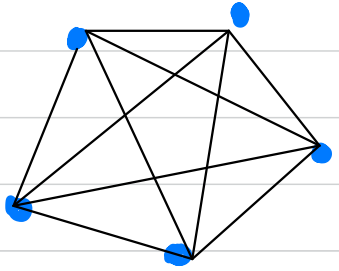
Multi Graph → An undirected graph in which multiple edges are allowed between any 2 vertices.



multigraph with multi edge and loop.

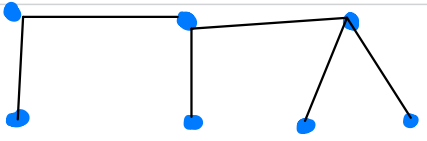
Simple Graph → An undirected graph in which both multiple edges and loops are not allowed.

Complete Graph → A graph in which every vertex is directly connected to every vertex.



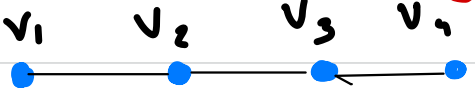
↙ example of complete graph.

Connected Graph → A connected graph has a path from every vertex to all other vertex.



↙ connected graph.

Path → A path P_n is a graph whose vertices can be arranged in a sequence.



$V = \{v_1, v_2, v_3 \ldots\ldots v_n\}$ Such that edge set of graph is

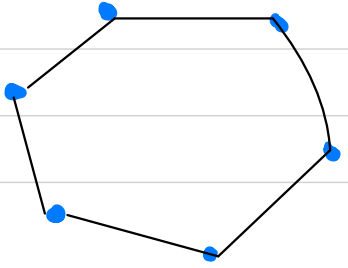
↳ Path

$$E = \{v_i v_{i+1} \quad \forall\, i \in [1, n-1]\}$$

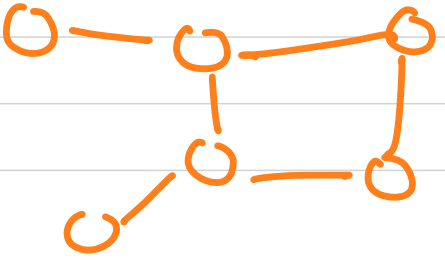
Cycle → A cycle C_n is a graph whose vertices can be arranged in a cyclic sequence

$V = \{v_1, v_2, v_3 \ldots\ldots v_n\}$ such that the edge set is

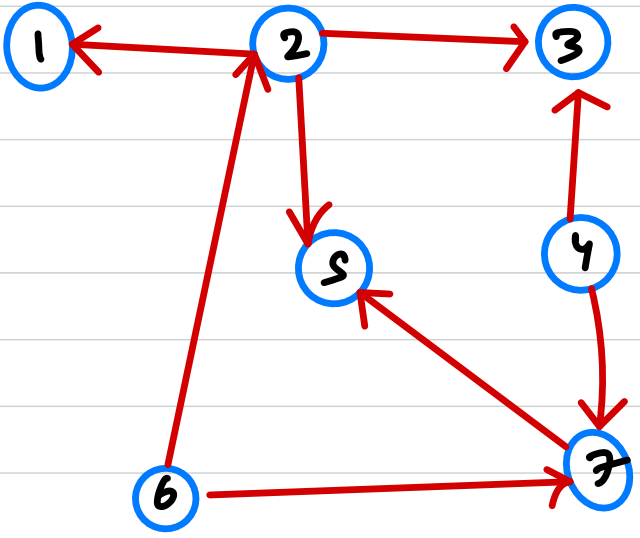
$$E = \{v_i v_{i+1} \quad \forall i \in [1, n-1]\} \cup \{v_1 v_n\}$$

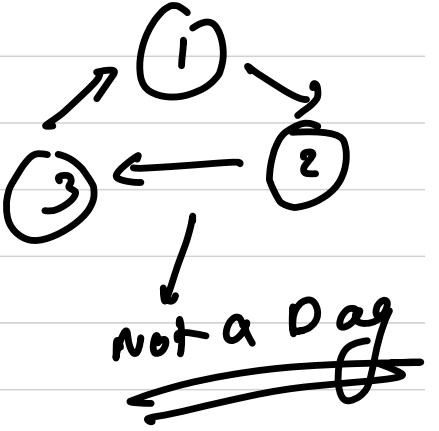


↙ cycle

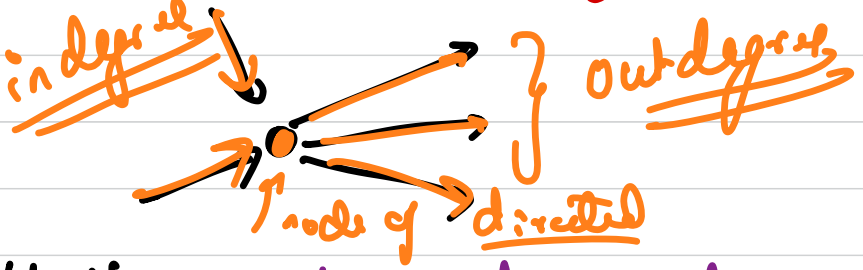


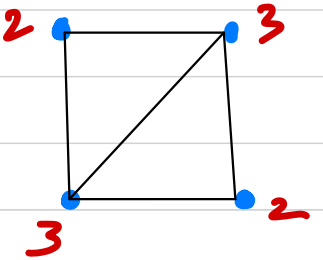
DAG (Directed Acyclic Graph)





Degree → Degree of a vertex in a graph is the total no. of edges incident to it / associated with it (for undirected graph)

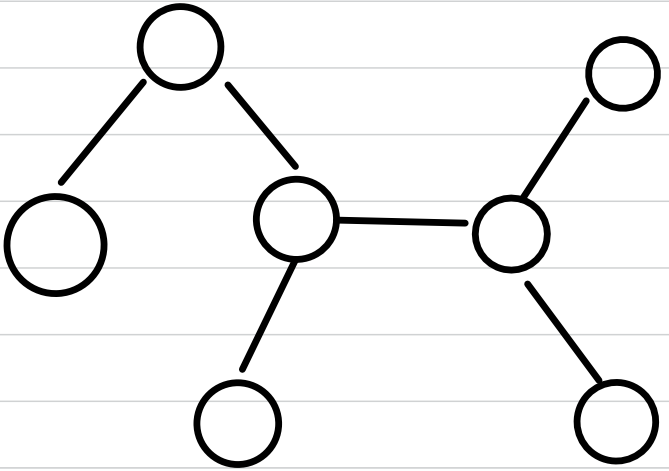




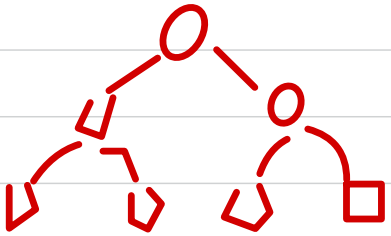
Note → In a directed graph, the out degree of a vertex is total no. of outgoing edges & indegree is total no. of incoming edges.

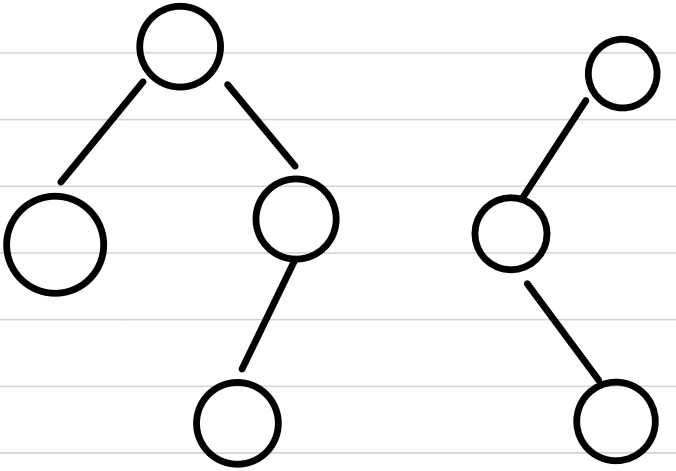
Tree → It is a connected graph with no cycles. If we remove all cycles from a graph we will get a tree. And if we remove an edge from a tree, it is called forest.

forest → An undirected graph where any 2 vertices are connected by atmost 1 path.



Tree





forest

Relationship between number of edges & vertices

total edges

# Directed	$\|E\| = \|V\| \times (\|V\|-1)$	(max)
# Undirected	$\|E\| = {}^{\|V\|}C_2$	(max)
# Connected	$\|E\| = \|V\|-1$	(min)
# Tree	$\|E\| = \|V\|-1$	
# forest	$\|E\| = \|V\|-1$	
# Complete	$\|E\| = {}^{\|V\|}C_2$	

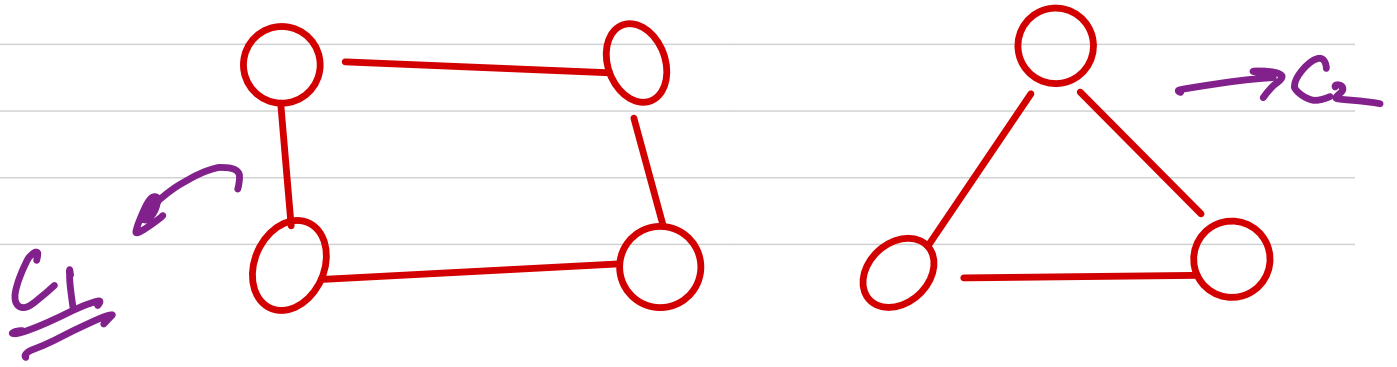
${}^{n}C_r$

${}^{|V|}C_2$

max

Connected Components

* Component. → If there is a dis-connected graph, then the set of vertices which are connected forms a component..



How To Read Graphs

As graphs are non-linear data structure, we need some mechanism to read graphs. These mechanisms are also called as Graph Traversals.

1. Depth first Search (DFS) → Recursive
2. Breadth first Search (BFS) → Iterative

Depth First Search

Lets take a motivation problem :

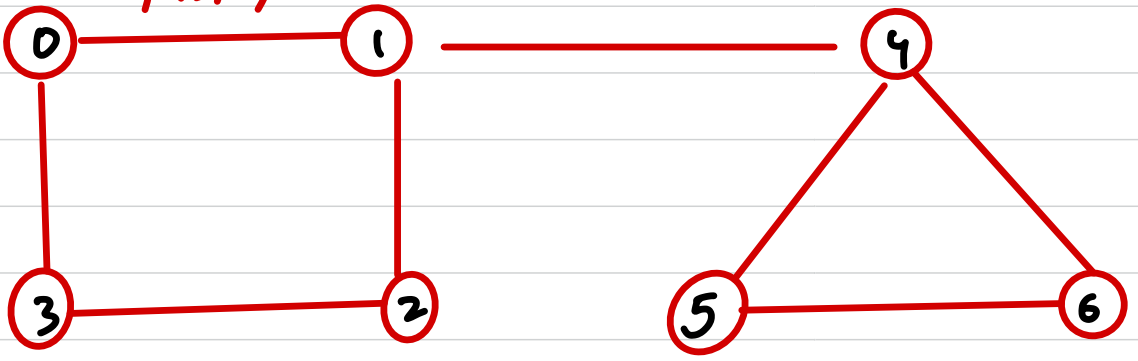
Given a graph calculate all paths between 2 vertices

OR

Given a graph check whether there is a path between any 2 vertices.

f(0,5) → f(1,5), f(3,5)

visited



0 – 5 (any path from 0 to 5)

0 – 1 ← direct edge

0 – 3 ↵

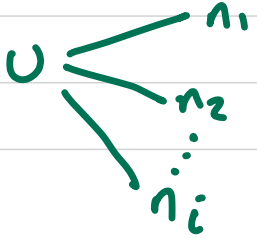
$f(u, v)$ = $\begin{cases} f(n_1, v) \\ \text{or} \\ f(n_2, v) \\ \text{or} \\ f(n_3, v) \\ \text{or} \\ \vdots \\ \text{or} \\ f(n_i, v) \end{cases}$

Recursive

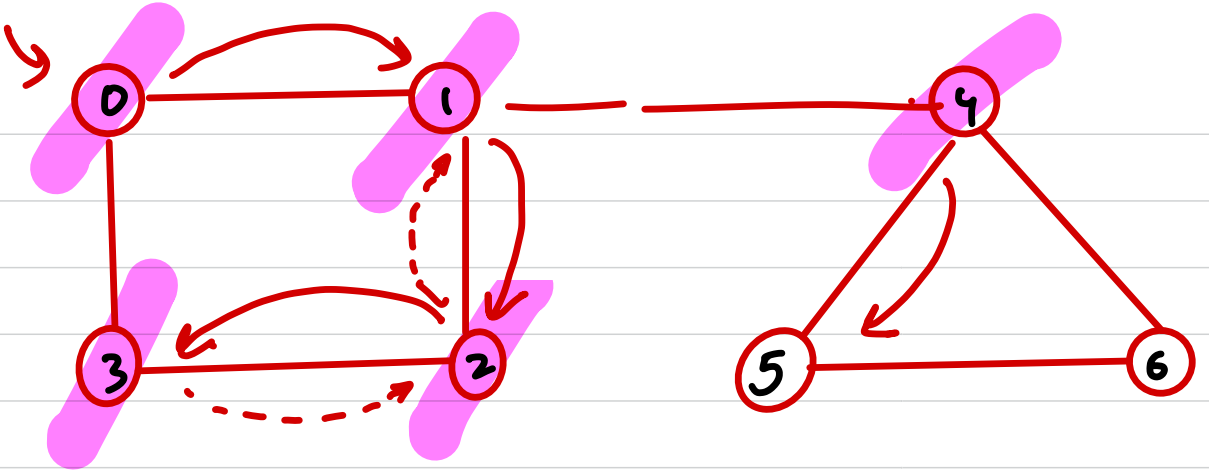
[ ]

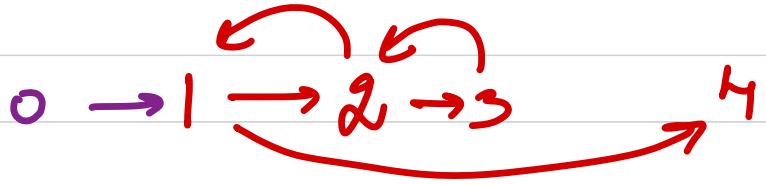
$f(u, v)$ ↓

whether there is a path from u to v or not



$n_1, n_2 \dots n_i$ are immediate neighbour of u.





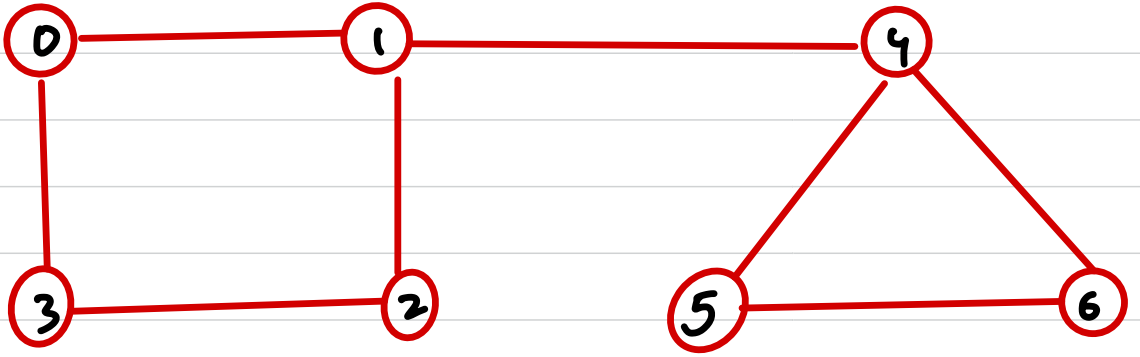
boolean [ ]

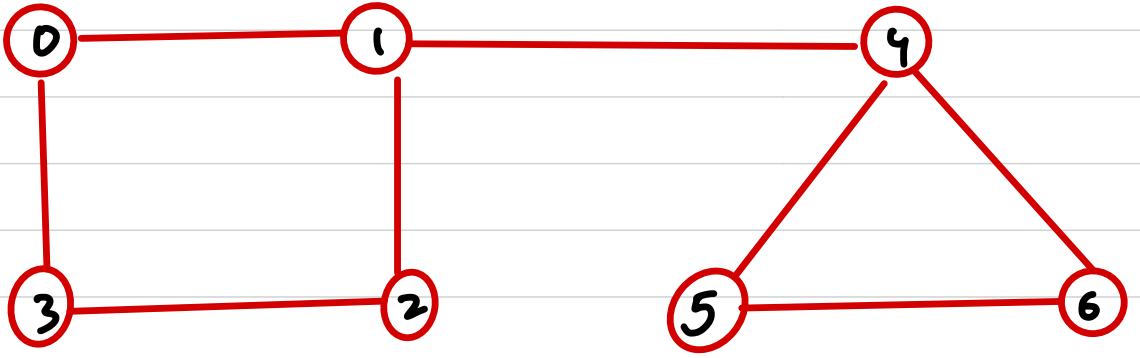
hashmap

→ visited

Recursion

Stack





g(0,5)

0, 1, 4, 5

0, 1, 4, 6, 5

0, 3, 2, 1, 4, 5

0, 3, 2, 1, 4, 6, 5

$g(v, v) = v + g(n_j, v)$

$g(v, v)$ → all paths from $v \to v$

$j \in [0, i]$

n_i → neighbour of v

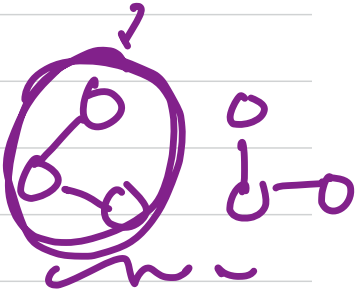

0
1
4
3
2
5
6

```cpp
bool has_path(std::vector<std::list<int> > &graph, int start, int end, std::vector<bool> &

    if(start == end) return true;
    visited[start] = true;
    for(auto &neighbour : graph[start]) {
        if(not visited[neighbour]) {
            bool result = has_path(graph, neighbour, end, visited);
            if(result) return true;
        }
    }
    return false;
}

bool validPath(int n, std::vector<std::vector<int>>& edges, int start, int end) {
    std::vector<std::list<int> > graph(n, std::list<int> ());
    for(auto &edge : edges) {
        int u = edge[0];
        int v = edge[1];
        graph[u].push_back(v);
        graph[v].push_back(u);
    }
    std::vector<bool> visited(n, false);
    return has_path(graph, start, end, visited);
}
```

Recursion

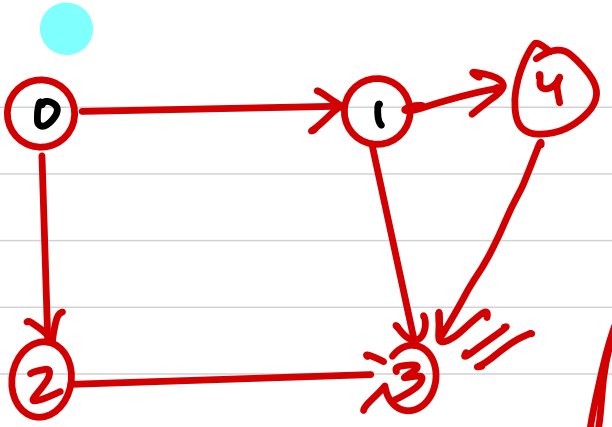
DFS

Call stack



Time → O(V+E)

DFS



visited

[0, 2, 3]

{0, 2, 3

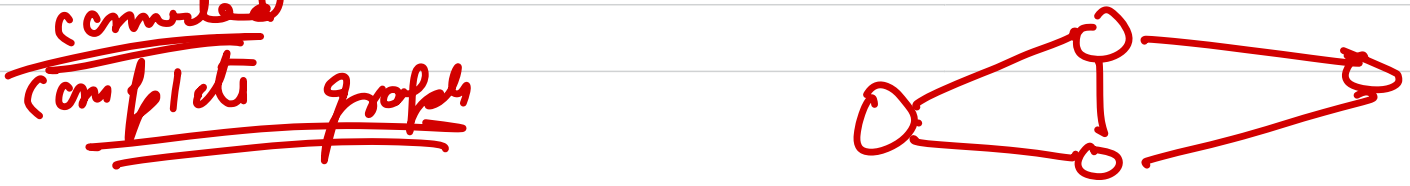
path

{

{0, 1, 3}
{0, 1, 4, 3}
{0, 2, 3}

count

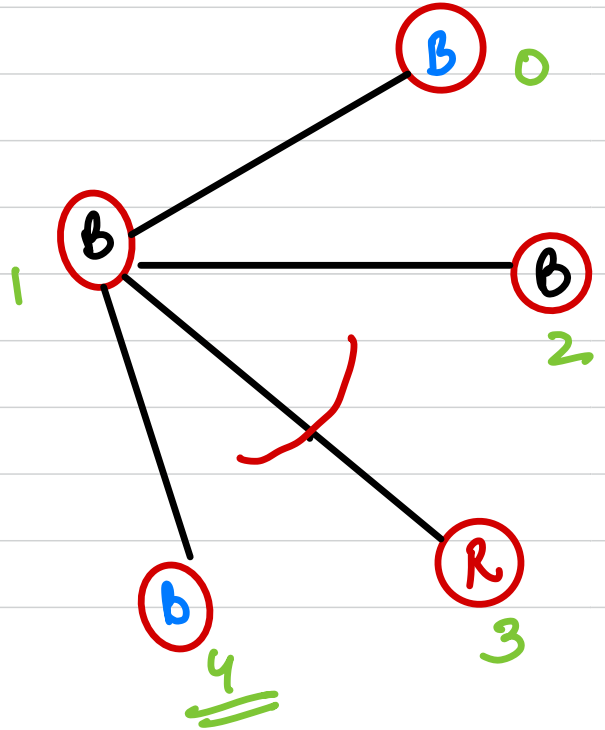
```cpp
std::vector<std::vector<int> > result;

void all_path(std::vector<std::vector<int>>& graph, int start, int end, std::vector<int> &path
    if(start ==  end) {
        path.push_back(end);
        result.push_back(path);
        path.pop_back();
        return;
    }

    path.push_back(start);
    visited[start] = true;
    for(auto &neighbour : graph[start]) {
        if(not visited[neighbour]) {
            all_path(graph, neighbour, end, path, visited);
        }
    }
    visited[start] = false;
    path.pop_back();

}

std::vector<std::vector<int>> allPathsSourceTarget(std::vector<std::vector<int>>& graph) {
    std::vector<int> path;
    int n = graph.size();
    std::vector<bool> visited(n, false);
    all_path(graph, 0, n-1, path, visited);
    return result;
}
```



TREE CUTTING - Codeforces (Div 3 - F1) (1800)

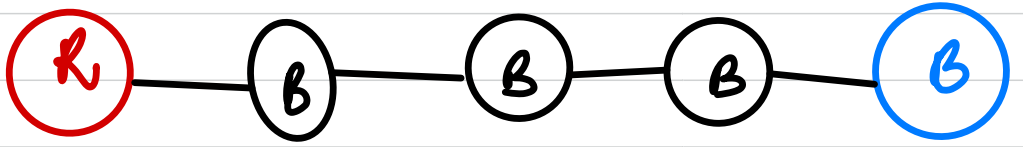
R ✓
B ✓
R - black ✓
b - black ✓
R - atleast

Example

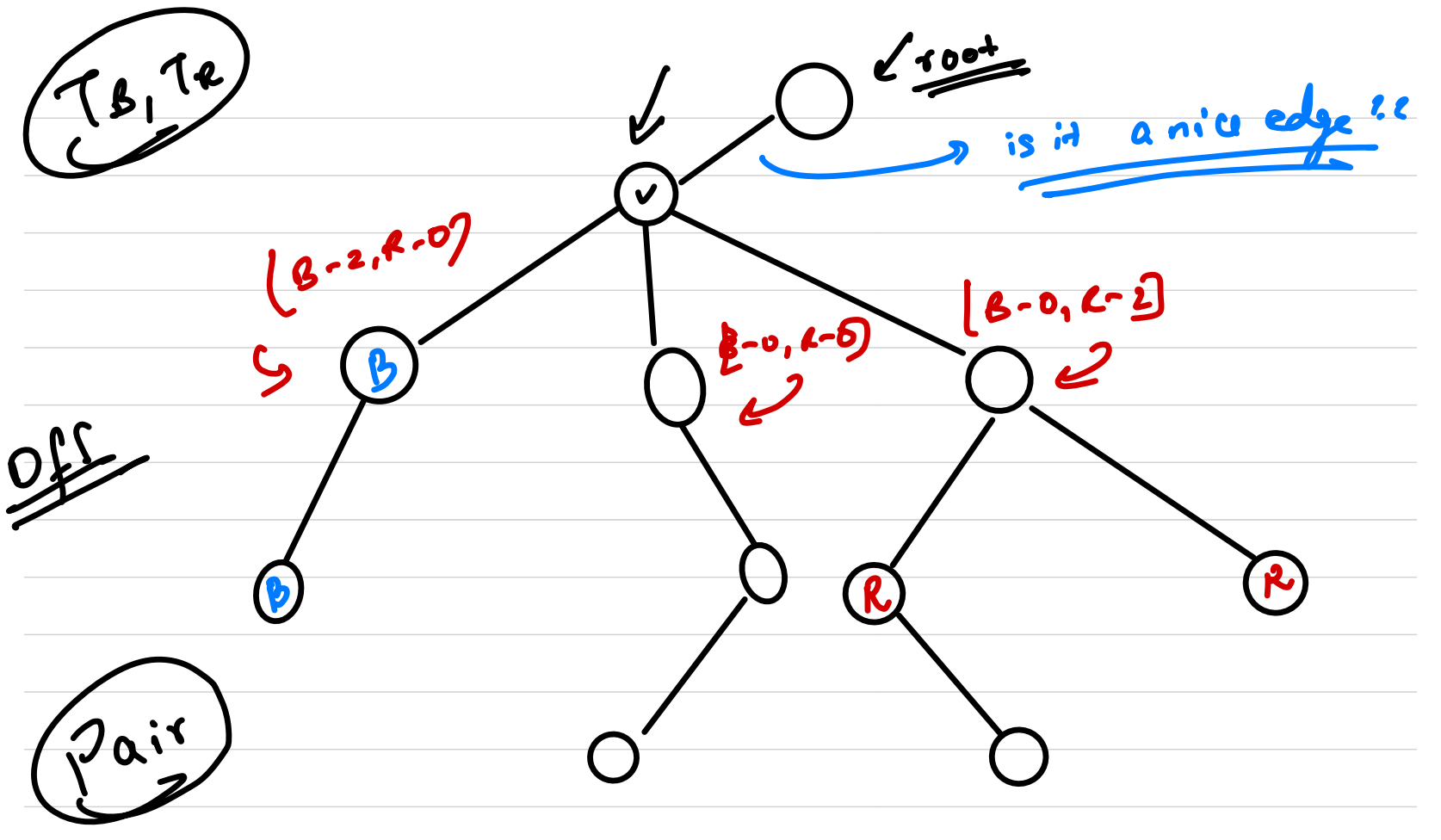


1

Example



→ Tree → no cycles


T_B, T_R
root
v
is it a nice edge ??
[B-2, R-0]
B
[B-0, R-0]
[B-0, R-2]
B
R
R
Off
Pair

We need to get the distribution of Red and Blue nodes on a subtree

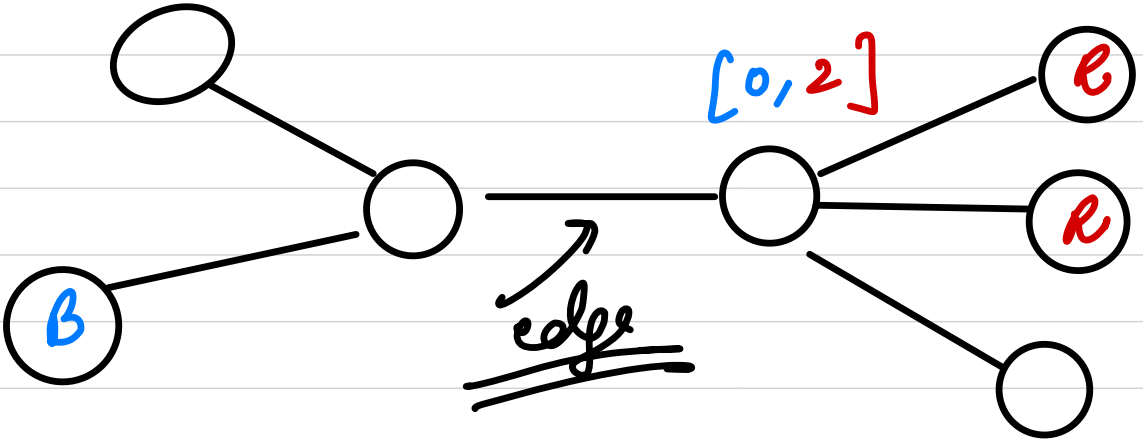
$< R, B >$

$R > 0$ and $B > 0$ → then the edge is not tie

$f(v, c) = \sum_{u} f(u, c)$ [u ∈ neighbours of v]

$+ (v.\text{color} == c)$

$f(v, c)$ returns the distribute of blue colored nodes for the tree rooted at v.



$T_R - 2$

$T_B - 1$

Problem: Proove that a tree with n nodes has n-1 edges.

Principal of Mathematical Induction → (PMI)

→ f(n)

This func^n returns the no. of edges in a tree with n nodes

1) Base Case $n = 1$

$f(1) \rightarrow 0$ (Proved for base case)

2) Assumption

$f(n) \rightarrow n-1$ (we assume that $f(n)$ works completely fine)

3) Proof (self work)

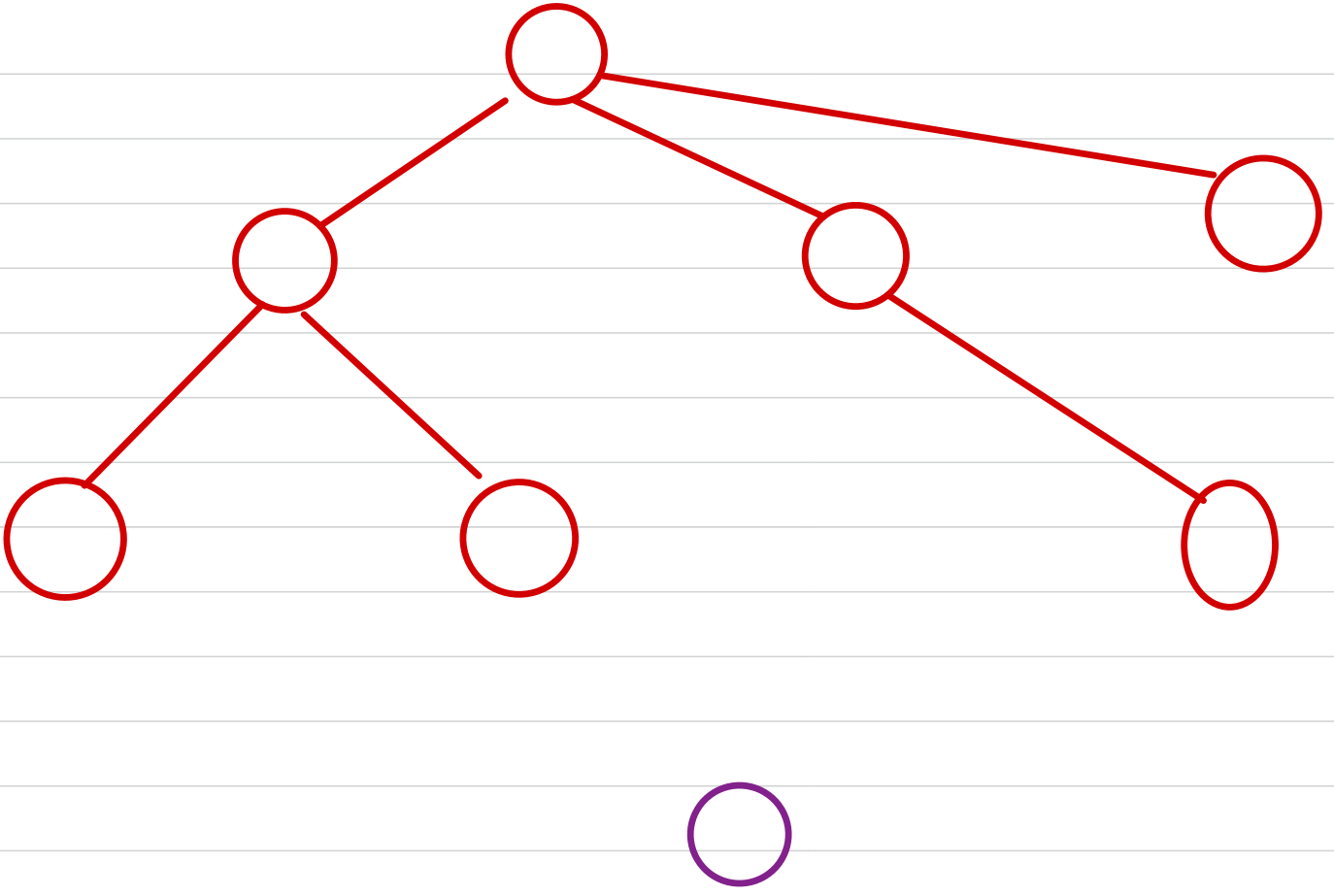
We will try to calc the value of

$f(n+1)$

$(n+1)$ → tree is having $n+1$ nodes

$f(n) = n-1$

$n+1$ node tree is equal to a tree with n nodes & one more node gettly attached to the tree.



$f(n+1) = f(n)$ + no. of edges reqd to add one more node to the tree

$f(n+1) = (n-1) + 1$

$f(n+1) = n$

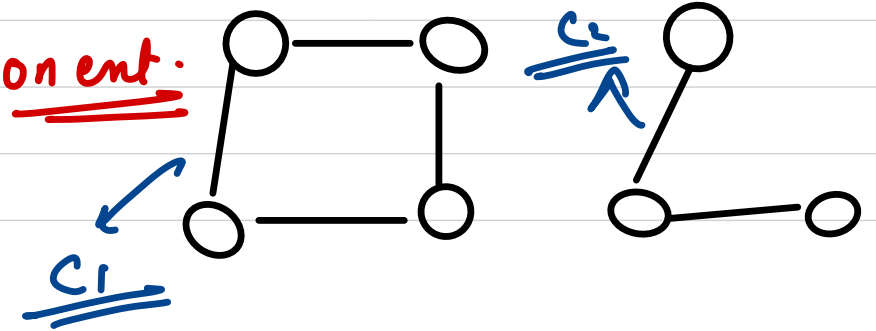
H.P

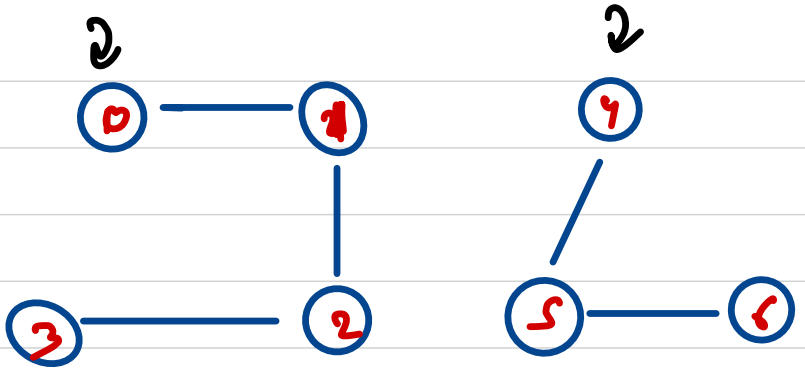
What is Connected Component ??

In a graph a set of vertices connected with each other forms one connected component.

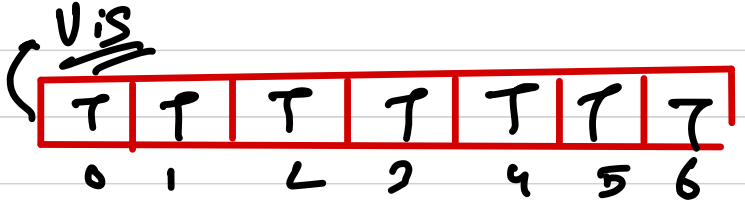
If the graph is disconnected then it will be having more than one connected component.

else only one component.





CC = 2



algo → the no of times we start doing dfs or BFS will be the no. of CC.

Clone A graph

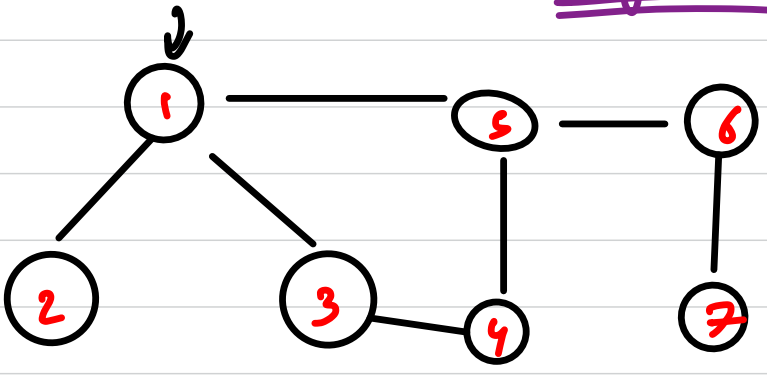
→ given → one single node.

→ To create the clone, we should read the graph atleast once.

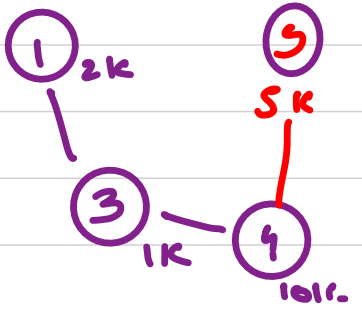
- DFS
- BFS

→ O(V+E)

adj list → new cloned graph



Original graph



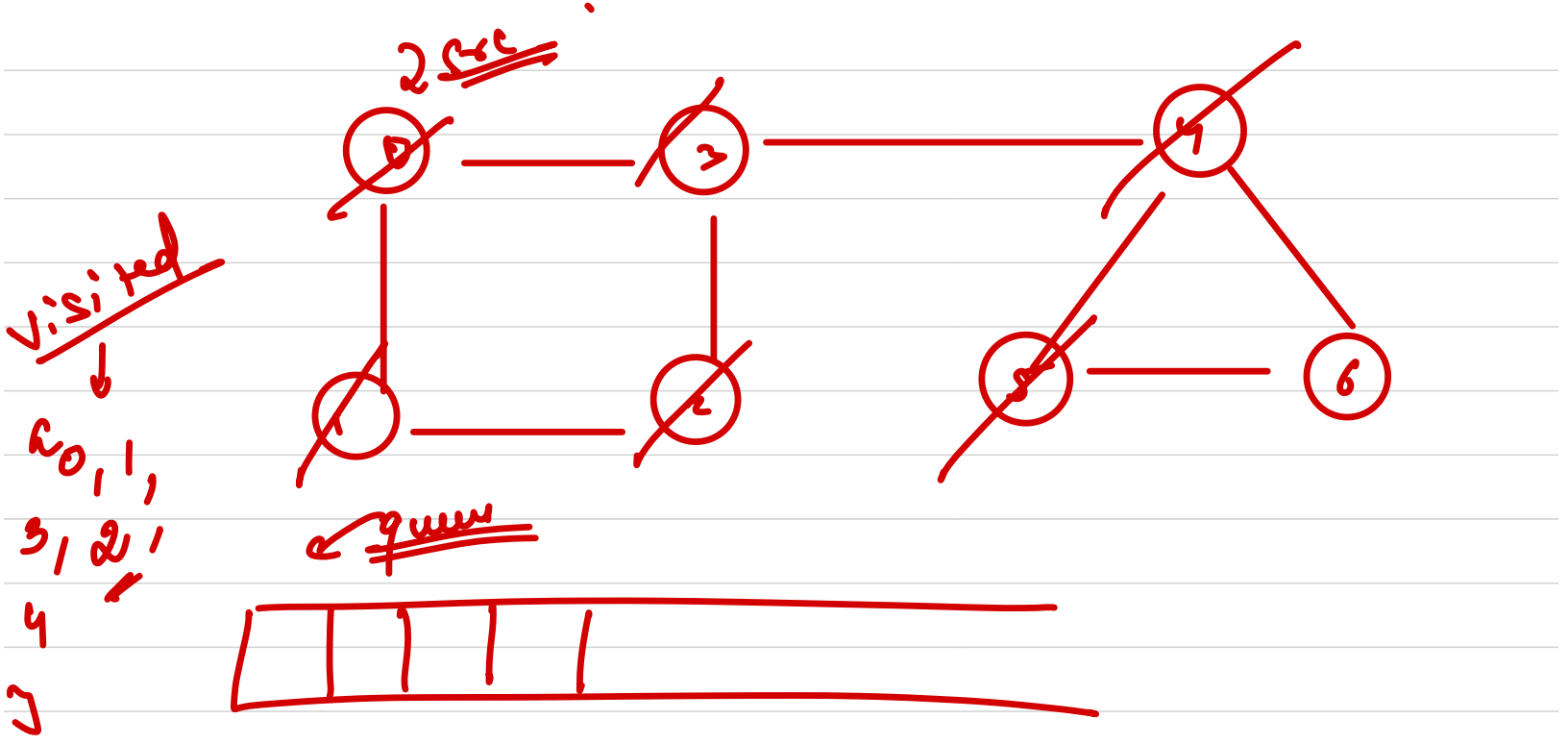
[1-2K, 3-1K, 4-10K, 5-5K]

array / hashmap

Start reading the graph from node 1 using dfs

prepare a visited to keep track of nodes

$$ {}^{n}C_{r} = \frac{n!}{r!\,(n-r)!} = \frac{{}^{n}P_{r}}{r!} $$


2 src
0
3
4
1
5
6
visited
0, 1,
3, 2,
4
5
queue