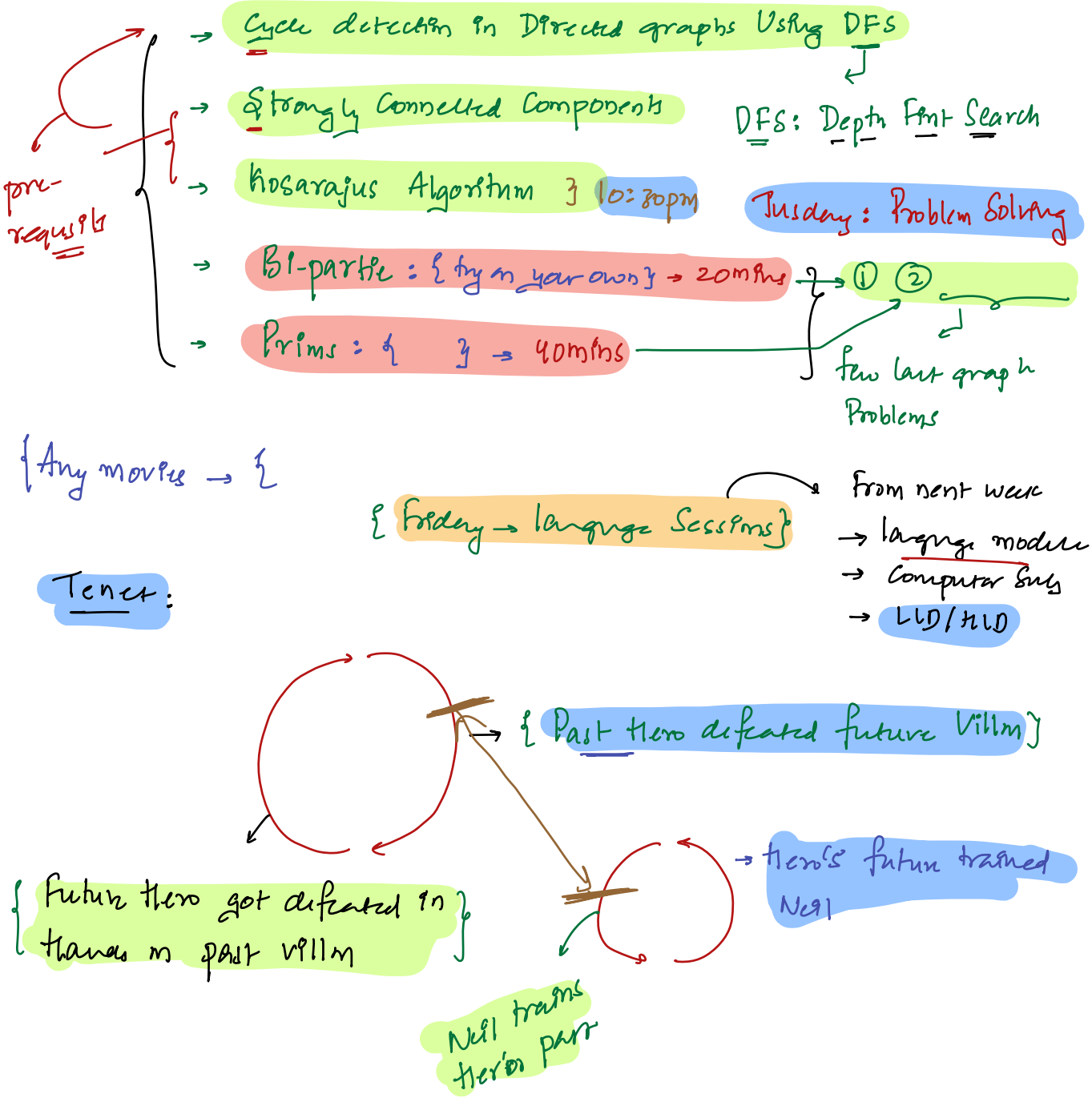
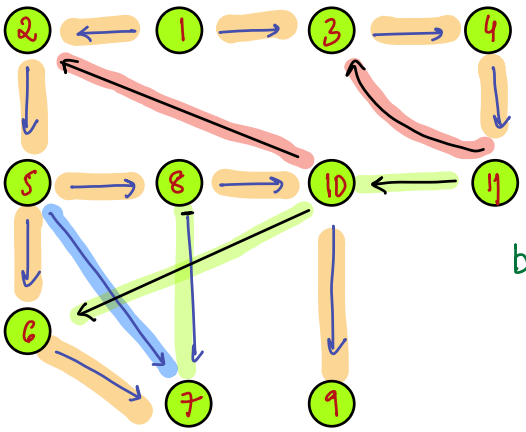
→ Today's Content:

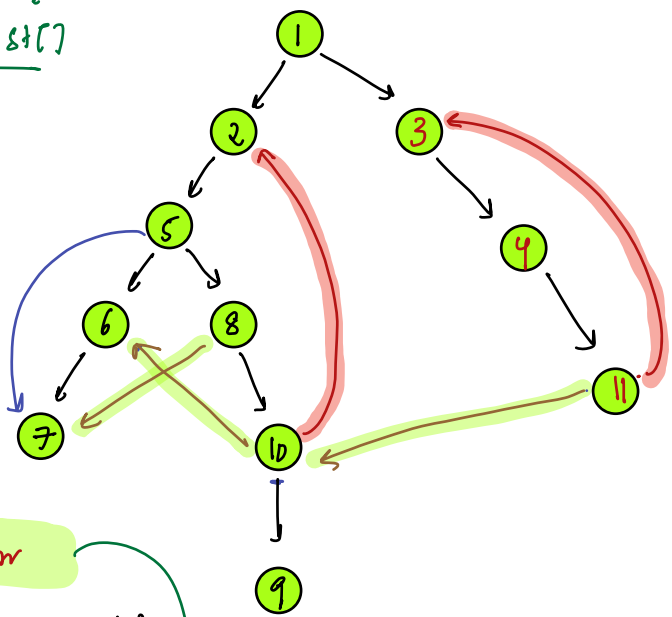




vis[12] = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| | T | T | T | T | T | T | T | T | T | T | T |

st: ~~| | | | | | | | | | | | |~~

bool st[]



→ : { Tree Edge } : DFS Tree

→ : { Forward Edge } : Ancestor → Child

→ : { Backward Edge } : node → Ancestor

→ : { Cross Edges } : { node not in same path }

{ If Backward Edge is present
cycle is present }

Cross Edge : { They will always go from Right → Left } → { TODO : for fun proof }

obs: 8 → 7 : Edge to Visited Node, Still No Cycle

10 → 2 : Node has a Edge to it's ancestor node it's has a Cycle

11 → 3 : 3 is ancestor to 11 hence cycle, 3 is inside start represent 3 to be ancestor of 11

Pseudocode:

Only directed graphs / DFS / BFS → Topological

```
bool isCycle(list<int> g[], int s, bool vis[], bool st[]){

    // Say s is already visited
    if(vis[s] == True){ return False }

    vis[s] = True; { iterate on it's adj nodes }
    for(i=0; i< g[s].size(); i++){
        d = g[s][i];
        // if d belongs in stack
        if(st[d] == True){ return True }
        if(vis[d] == True){ continue }
        else{ // visiting node d for first time
            st[d] = True;
            if(isCycle(g, d, vis, st)){return True}
        }
        st[d] = False
    }
    // If we are coming to end, & not returning True, what do we need to return
    return false
}
```

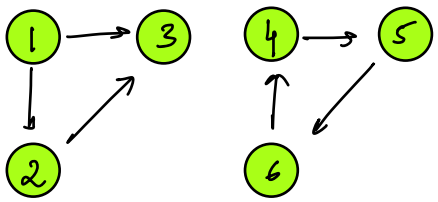
need one of these things

strictly optional

if function call entered d is in stack

if function call completed d is no longer in stack

Eg: N = 6 → nodes

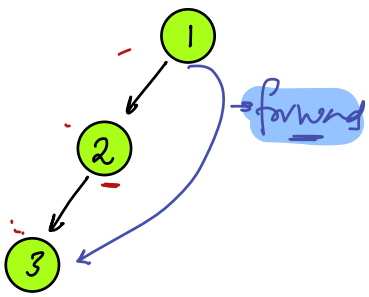


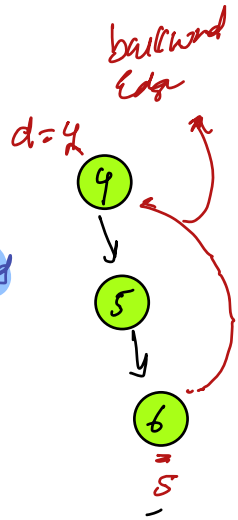
// vis[7] =

	1	2	3	4	5	6
	T	T	T	T	T	T

st[7] =

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





// Started dfs at 1, 2, 3

// Started dfs at 4, 5, 6 → {Cycle detected}

→ main() {

TC: → O(N + E)

SC: → O(N + E)

// given N Node & E Edges

→ Create g[]

→ vis[N+1] = {F} st[N+1] = {F}

ans = False

For(int i=1; i<=N; i++) { → {N + E}

if(vis[i] == False){

ans = ans || isCycle(g, i, vis, st)

}

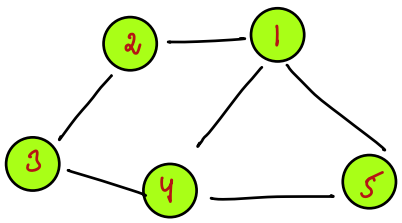
}

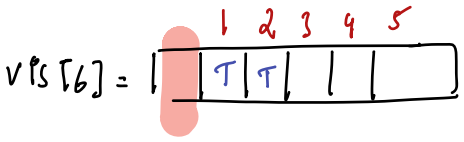
// If (ans == True) { Cycle present }

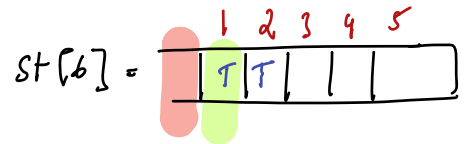
else { cycle not present }

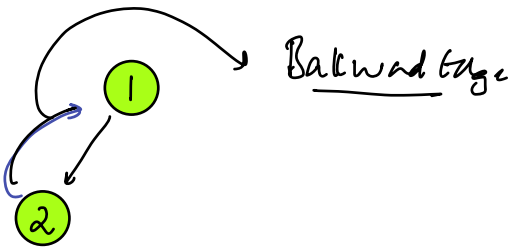
}

// Why above logic wont work for undercut graph?









Adj list 2 : 1 3

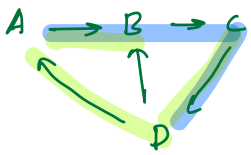
// Above exact code won't work for undirected graph

// Using DFS we can do it with small modification & Easily

// Strongly Connected Components → {Directed graphs}

⇒ Def: A Component is said to be Strongly Connected Component if from All nodes we can visit Each & every node

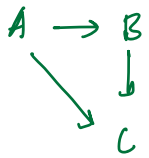
Ex1:



A : { B, C, D } C : { D, A, B }

D : { A, B, C } B : { C D A }

Ex2:



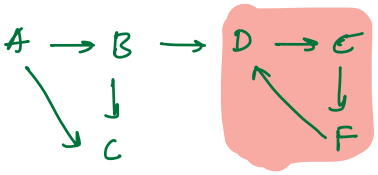
A : { B, C, D }

C : { }

B : { C }

⇒ Not a strongly connected Component

Ex3:



→ Component → {DEF}

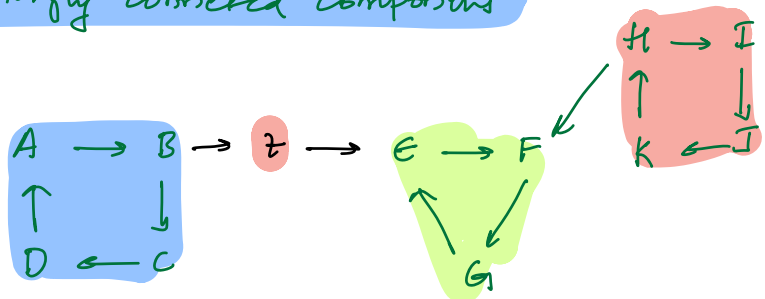
D : → { E, F }

E : → { F, D }

F : → { D, E }

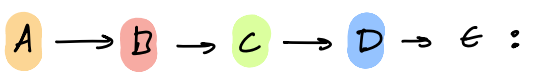
no: of Strongly Connected Components

Ex4:



⇒ # ans: 4

Ex5:



ans: 5

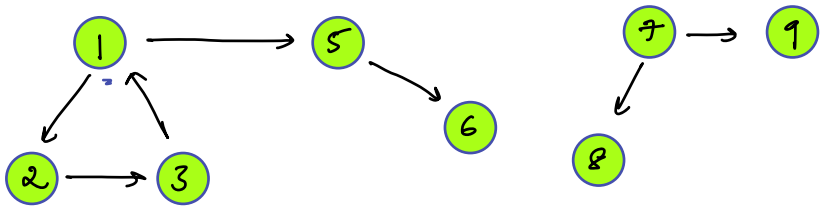
(A) (B) (C) (D) (E)

// → given a directed graph, Count no: of strongly connected Components

→ : Algo: { kosaraju's Algorithm

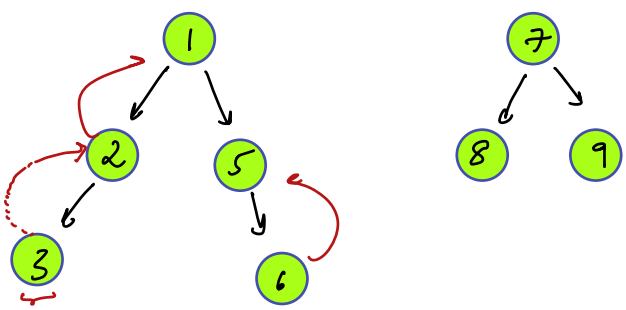
Step1: Apply dfs & store a node in list when we exit from that node

Eg1:



list: 3 2 6 5 1 8 9 7

→ { When we completly exit the node, add it in list }



{ keep repeating Step-1 till all node are inserted in list }

Step1: Given a graph, create a list

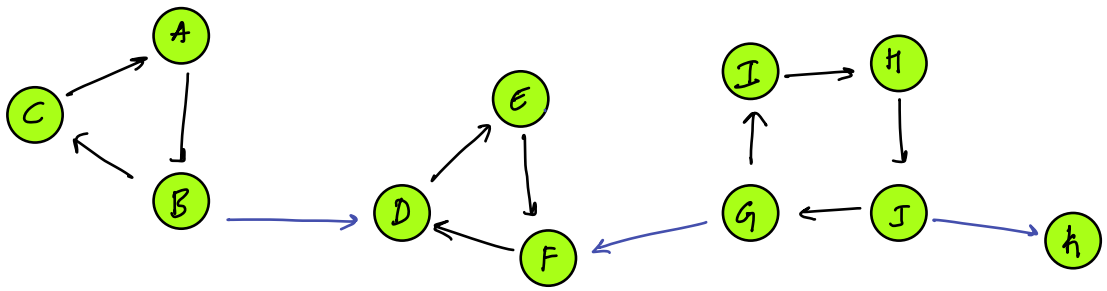
{ Insert a node when we completly come out of that node }

Step2: Reverse the list, & Reverse Every Edge in graph { change your outer adj list }

A → B, after reversing B → A

Step3:

Ex:



lpst:

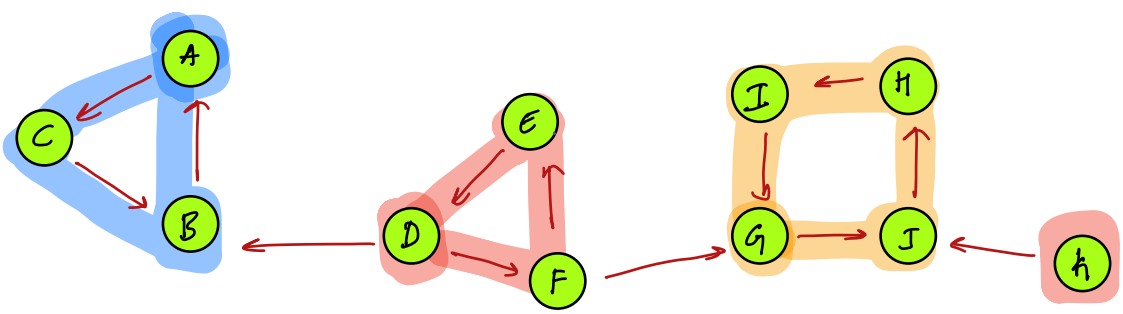
C F E D B A K J H I G

Step:2

1) revlpr:

G I H J K A B D E F C

2) revern every Edge



Step:3: { Take every node from rev-lpr & apply DFS, till all nodes visited }

→ G → H → I → J : {1}

→ K {1}

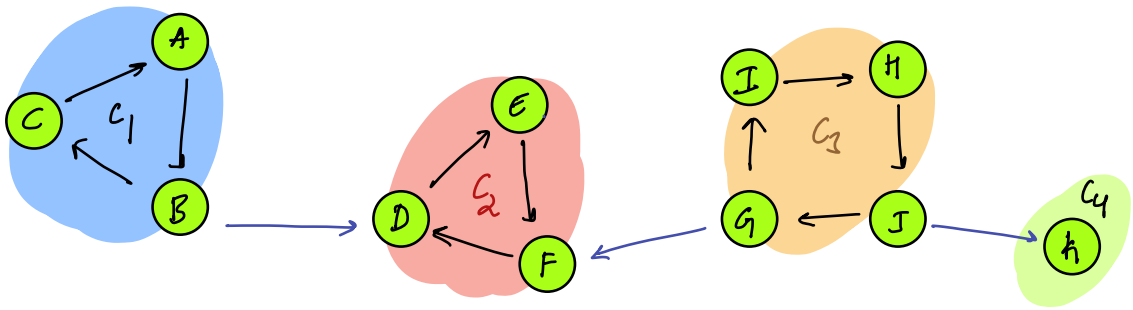
→ A → {A, B, C} : {1}

→ D → E → F : {1}

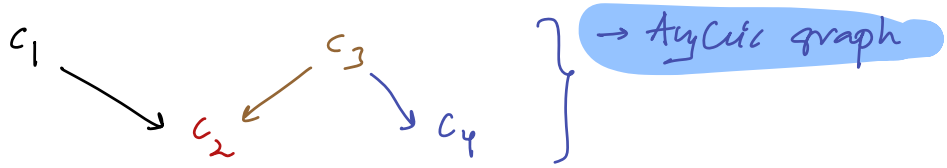
ans:4 :→ # how many time we applied DFS → count of strongly connected Components

TC: O(N+E) ↔ SC: O(N+E)

Why? : → Cp3/Cp4 : { Steven Halim Competitive Programming }



Proof

1) 

→ Acyclic graph

2) In your list: → a

→ There will be atleast one node of C_1 which will come after all node C_2

→ There will be atleast one node of C_3 which will come after all node C_2

→ There will be atleast one node of C_3 which will com after C_4

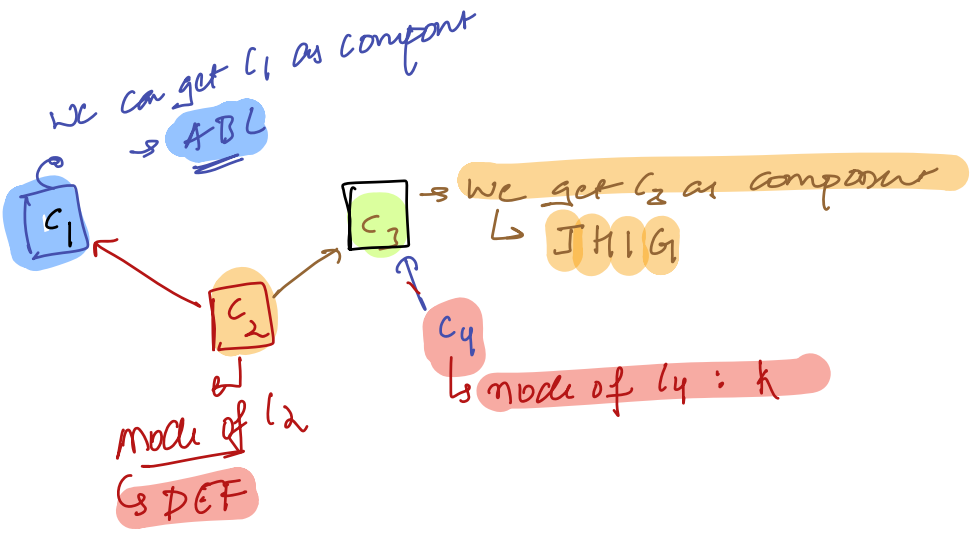
Eg: { list can come in any order, but it will satisfy condition }

{ All node of C_2 } → { One node of C_1 } → { All node C_4 } → { one node of C_3 }

Step:2 → {rev of list} order to apply DFS

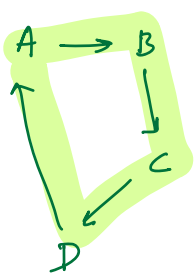
{one node C_3} → {All node of C_4} → {one node of C_1} → {All node of C_2}

Step:3



→ {CLSR → Algorithm of strongly connected Components}

→ {of strongly Connected} {Instagram}



{They do recomendation to all 4 of it seems}

→ probably of them buying together

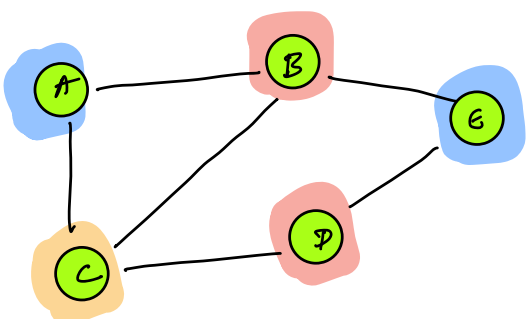
//

→ bi-partic graphs :

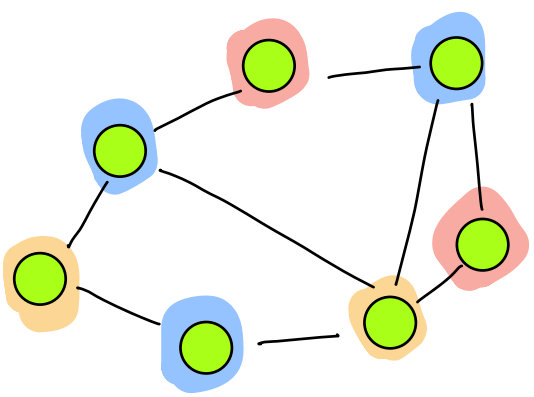
→ Prims

→ graph Colouring : → NP Hard / NP Complete

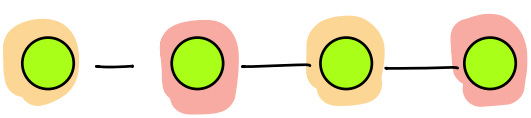
↳ given unadirected graph, min no: of colours required to colour entire graph such that no 2 adjacent nodes have same colour : Chromatic Number



→ 3 colours



→ 3 colours



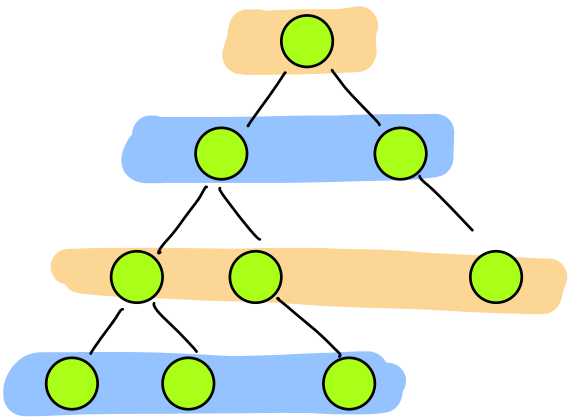
→ 2 colours

→ If Chromatic number of a graph = 2

: graph is called Bi-partite graph

Chromatic Number

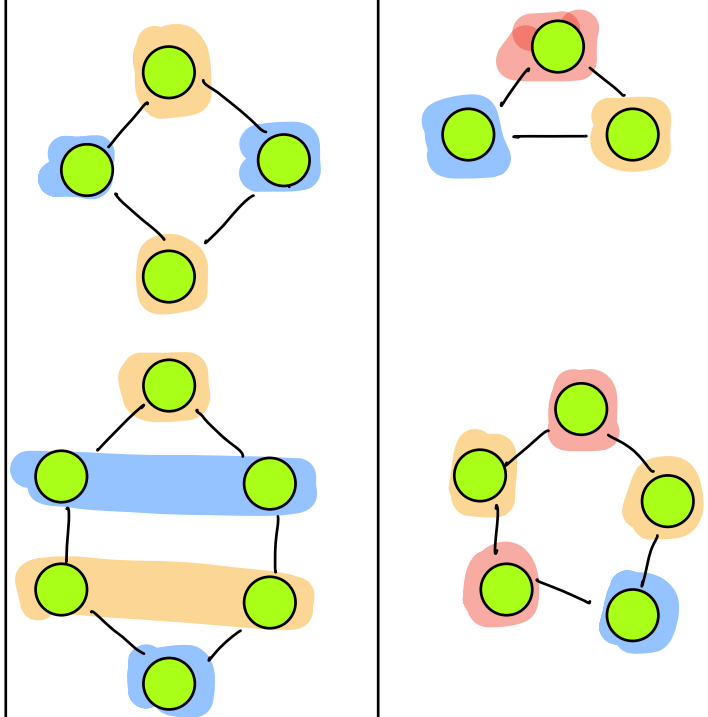
Observation1 : If graph has no cycle : 2



Obs2: Even length cycle

Chromatic number # 2

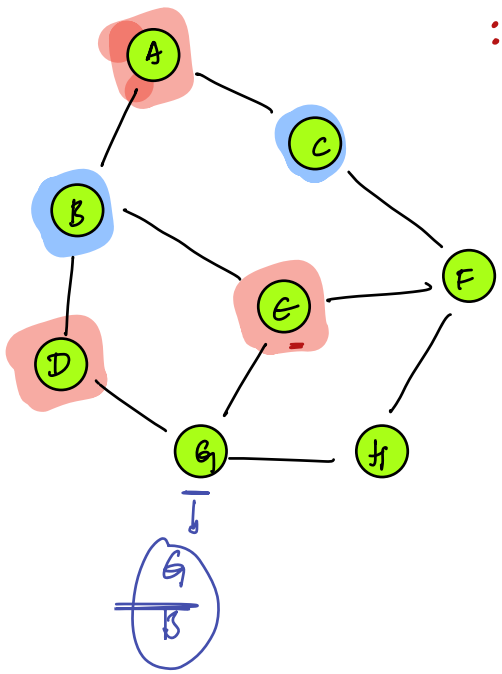
observation2 : Graph has cycle



// given a graph with cycle, check if has even length cycle or Not?

→ If we can fill cycle 2 colour : Chromic number : 2 : Even length

1) given a graph, → Chromatic = 2 → Bipartite or not ?

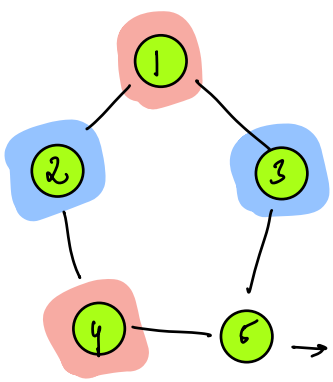


: | A | B | C | D | E | F | G |
R B B R R R B

F → for f
: already C : Red
: already E : Blue
} filling the above using 2 colors not possible
→ not bipartite

col: | 0 | 1 | 2 | 3 | 4 | 5 |
| ∅ | ∅ | ∅ | ∅ | ∅ |
1 2 2 1 1

0 → Predicate no color
1 → Color Red
2 → Color Blue



| 1 | 2 | 3 | 4 | 5 |
C: R B B R R

5 → { According to 4 : Blue : 2
We already allocate Red to 5 : 1 }

→ Contradiction, ∴ Not Bi-partite

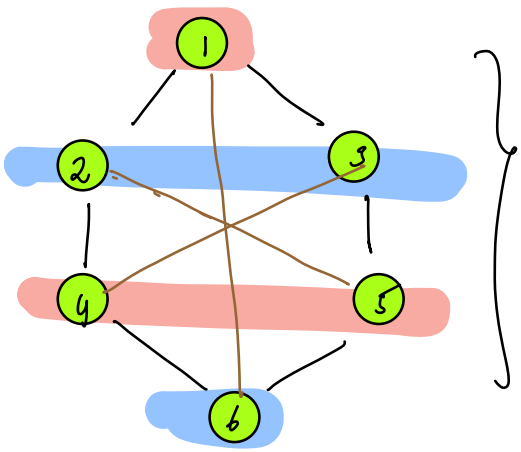
// Pseudocode:

```
bool checkBipartite ( list<int> g[], int N) {
    int c[N+1] = 0
    int c[1] = 1;
    queue<int> q;
    q.insert(1);
    while( q.size()>0) {
        int u = q.front();
        q.dequeue();
        // Assign colour to it's adjacent Nodes
        for( int i=0; i< g[u].size(); i++) {
            v = g[u][i]
            if( c[v] == 0) {// no colour assigned }
                c[v] = 3 - c[u]
                q.insert(v)
            }
            else if( c[v] == c[u]) {
                return false
            }
        }
    }
    return True
}
```

{ Iterate & get no: of node in Red Colour & green colour }

TC: O(N+E) SC: O(N+E)

// 28: given a bi-partite, max edges can be added between 2 different nodes so that entire graph is still bi-partite



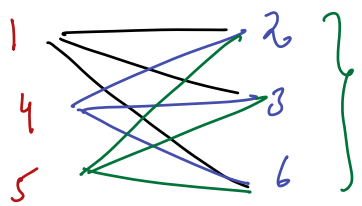
max no: of Edge we can add to make

→ ans: 3

→ Ideas: →

Red Colour

Blue Colour



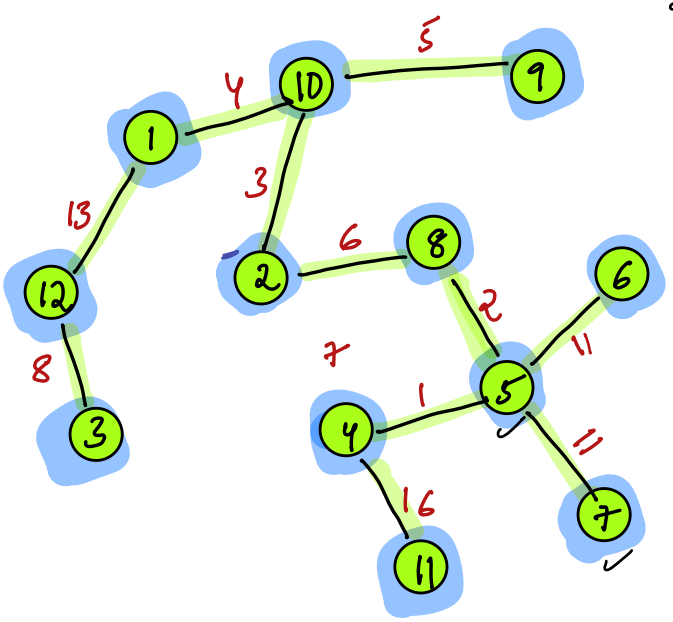
// Edges are between Red & Blue Colour

// Max Edges we can get

= #(No: of Node in Red Colour) * #(No. of Node in Blue Colour) - Edges

↳ max Edges we can add

308) {Prims} ——————→ Minimum Spanning Tree



Start: 2

u	v	w	
2	10	3	* since both visited cycle there
2	8	6	*
2	4	7	cannot consider ?
2	3	15	* cannot consider
10	9	5	*
10	1	4	*
10	8	9	* cannot consider
1	12	13	
9	8	10	* cannot consider
8	5	2	
5	6	11	
5	7	11	
5	4	1	
12	3	8	
4	3	14	* cannot consider
4	11	16	

Idea: We are always adding Adjacent Nodes, everything become 1 connected component.

In a single connected component, if we have connect 2 nodes cycle will form.

1) min heap

2) Take the Edge from min weight.
   - → if Both are visited → skip
   - → from Unvisited node, all the nodes to unvisited nodes

// Pseudocode:

```
int Prims ( list<pair<int, int>> g[], int N) {
    vis[N+1] = {F}   vis[1] = True
    minheap <pair<int, int>> mh; => // w v
    i=0; i< g[1].size(); i++){
        v= g[1][i].first
        w= g[1][i].second
        mh.insert( { w, v } )
    }
    while ( mh.size()>0) {
        pair<int, int> data = mh.top();
        mh.delete();
        int v = data.second
        if ( vis[v] == false ){
            vis[v] = True
            ans = ans + data.first
            i=0; i< g[v].size(); i++) {
                int x = g[v][i].first
                int y = g[v][i].second
                if(vis[x] == false) { mh.insert ({y,x})
            }
        }
    }
    return ans;
}
```

Edge weight u → v

insert all unvisited adjacent nodes inside heap v y x

TC: O(N + ElogE) SC: O(N+E)

0 Issue →

↳ Dynamic Programming → 8

Oct:
Nov:
Dec

9573546915 → Any weekday { 11am - 5pm }

↳ LinkedIn / Whatsapp / Email

Jan
Feb
Mar
April
May
Jun

→ 5 months → Jokes → Sports spam
→ { offended }

→ Diamonds :

→ Life liquor for Everyone