Tree

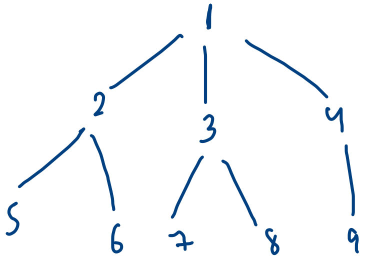
(i) no cycle

(ii) connected

traversal :

→ Pre, In, post

→ levelorder



levelorder :

1

2 3 4

5 6 7 8 9

Graph

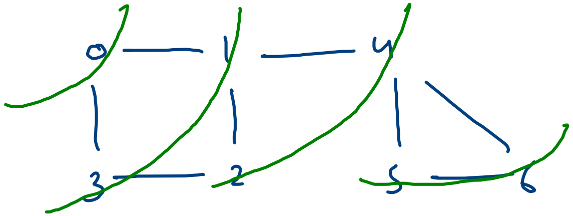
(i) cycle

(ii) it can be disconnected

traversal

→ DFT / DFS

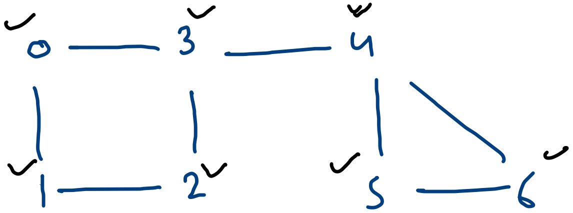
→ BFT / BFS



src = 0

levelorder :

0	→ 0 edges
1 3	→ 1 edges
2 4	→ 2 edges
5 6	→ 3 edges



Src = 2

2 @ 2

1 @ 21

3 @ 23

0 @ 210

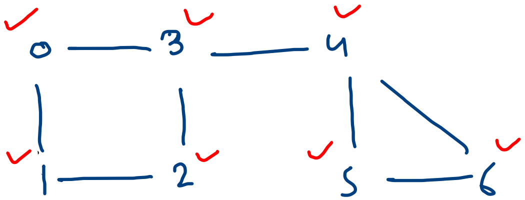
4 @ 234

5 @ 2345

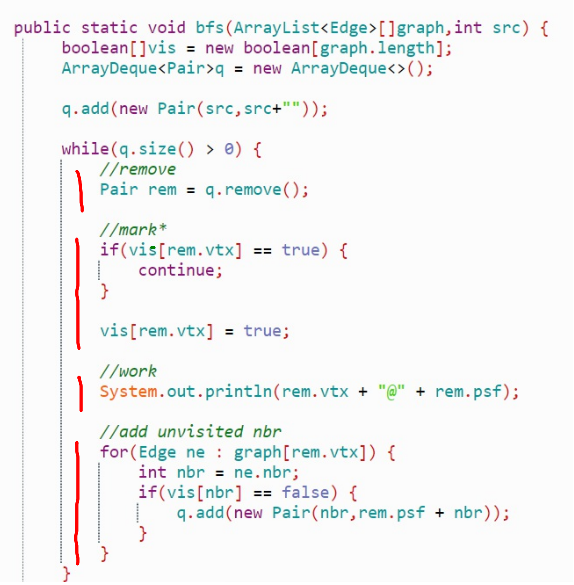
6 @ 2346

~~2,2~~	~~1,21~~	~~3,23~~	~~0,210~~	~~0,230~~	~~4,234~~	~~5,2345~~	~~6,2346~~	~~6,23456~~
0	1		2			3		

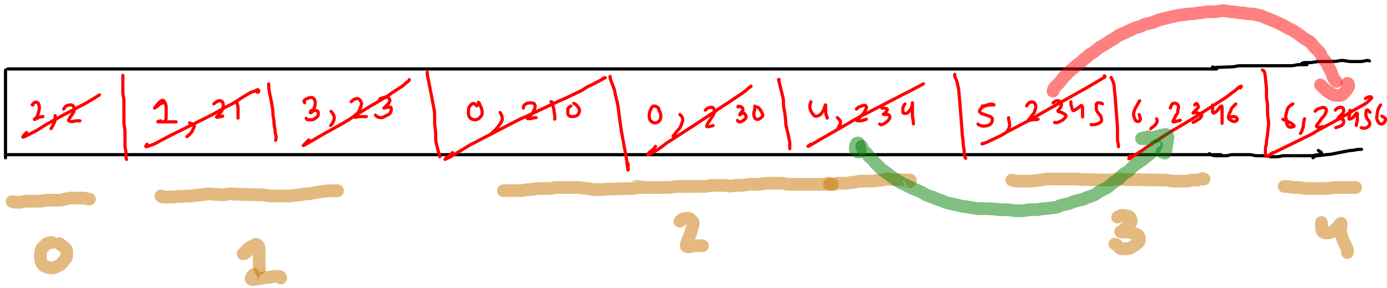
BFS
- remove
- mark *
- work
- add unvisited nbr



Src = 2



```
public static void bfs(ArrayList<Edge>[]graph,int src) {
    boolean[]vis = new boolean[graph.length];
    ArrayDeque<Pair>q = new ArrayDeque<>();

    q.add(new Pair(src,src+""));

    while(q.size() > 0) {
        //remove
        Pair rem = q.remove();

        //mark*
        if(vis[rem.vtx] == true) {
            continue;
        }

        vis[rem.vtx] = true;

        //work
        System.out.println(rem.vtx + "@" + rem.psf);

        //add unvisited nbr
        for(Edge ne : graph[rem.vtx]) {
            int nbr = ne.nbr;
            if(vis[nbr] == false) {
                q.add(new Pair(nbr,rem.psf + nbr));
            }
        }
    }
}
```



2 @ 2

1 @ 21

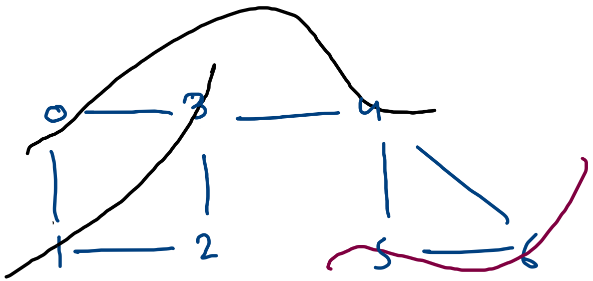
3 @ 23

0 @ 210

4 @ 234

5 @ 2345

6 @ 2346



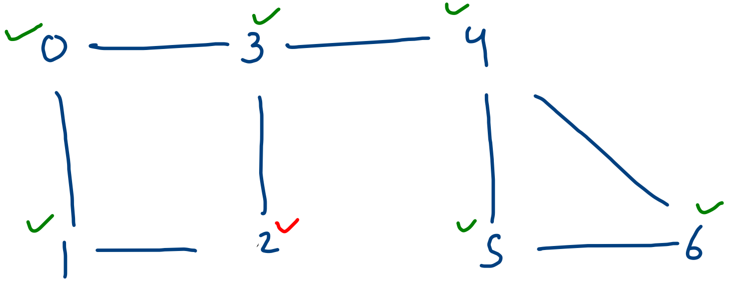
src = 2

0 ⟶ 2 (0 edges away from 2)

1 ⟶ 1, 3 (1 edges away)

2 ⟶ 0, 4 (2 edges away)

3 ⟶ 5, 6 (3 edges away)



src = 2

count = 1

lev = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

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

0 → 2

1 → 1 3

2 → 0 4

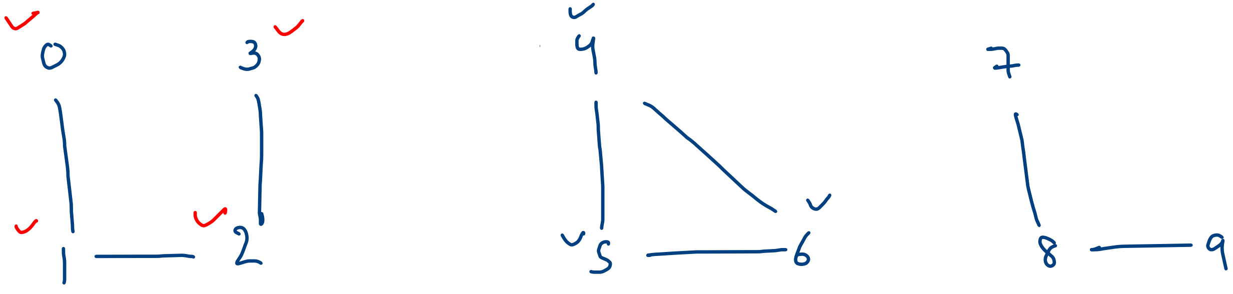
3 → 5 6

4 →

count times

- remove
- mark*
- work
- add unvis. nbr

cyclic

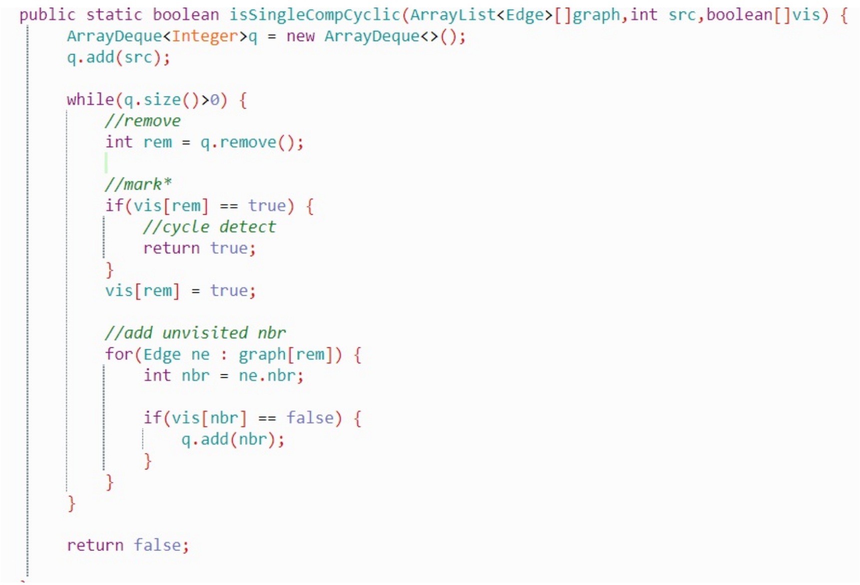


Acyclic : every comp is acyclic

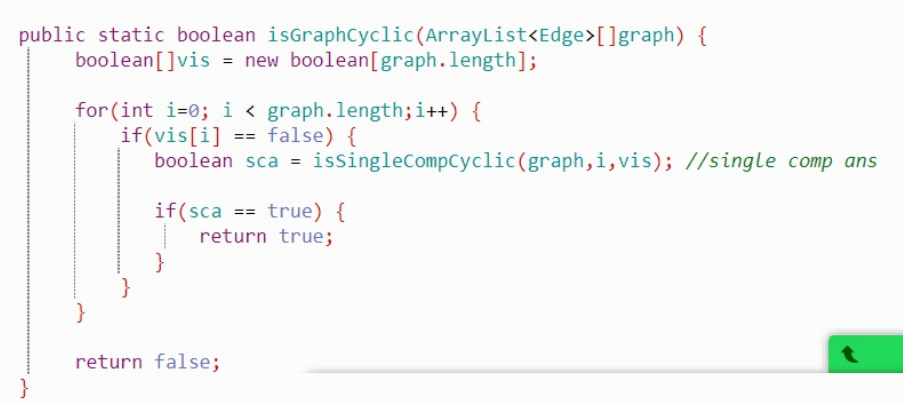
~~0~~	~~1~~	~~2~~	~~3~~

			*
~~4~~	~~5~~	~~6~~	~~6~~

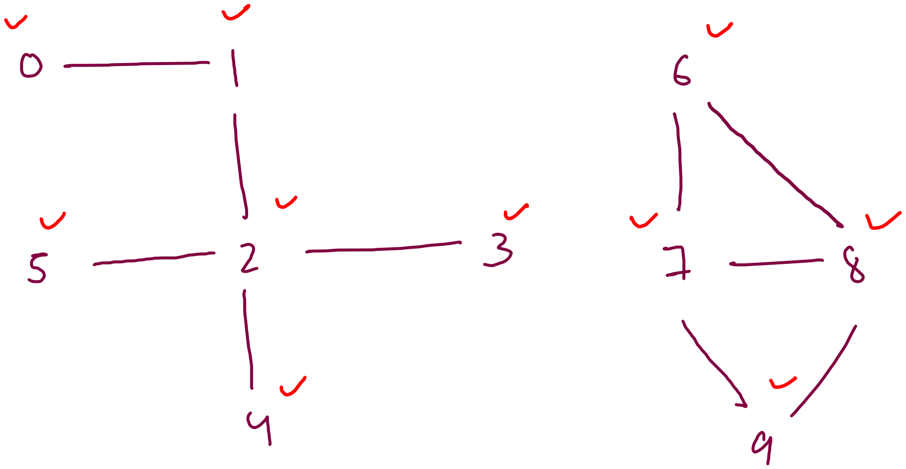
return true

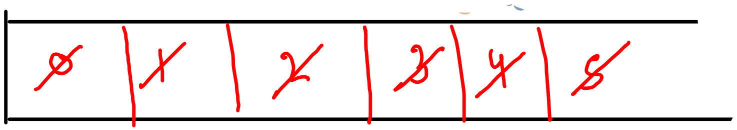


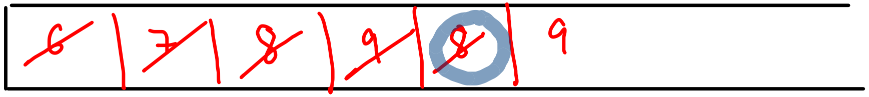
```
public static boolean isSingleCompCyclic(ArrayList<Edge>[]graph,int src,boolean[]vis) {
    ArrayDeque<Integer>q = new ArrayDeque<>();
    q.add(src);

    while(q.size()>0) {
        //remove
        int rem = q.remove();

        //mark*
        if(vis[rem] == true) {
            //cycle detect
            return true;
        }
        vis[rem] = true;

        //add unvisited nbr
        for(Edge ne : graph[rem]) {
            int nbr = ne.nbr;

            if(vis[nbr] == false) {
                q.add(nbr);
            }
        }
    }

    return false;
```

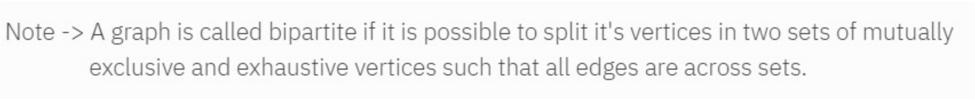


```
public static boolean isGraphCyclic(ArrayList<Edge>[]graph) {
    boolean[]vis = new boolean[graph.length];

    for(int i=0; i < graph.length;i++) {
        if(vis[i] == false) {
            boolean sca = isSingleCompCyclic(graph,i,vis); //single comp ans

            if(sca == true) {
                return true;
            }
        }
    }

    return false;
}
```

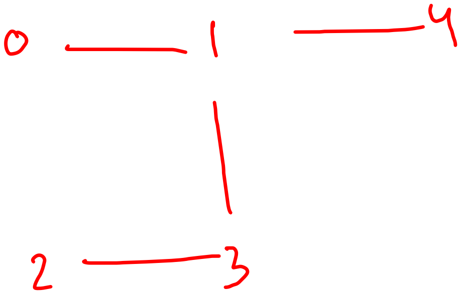


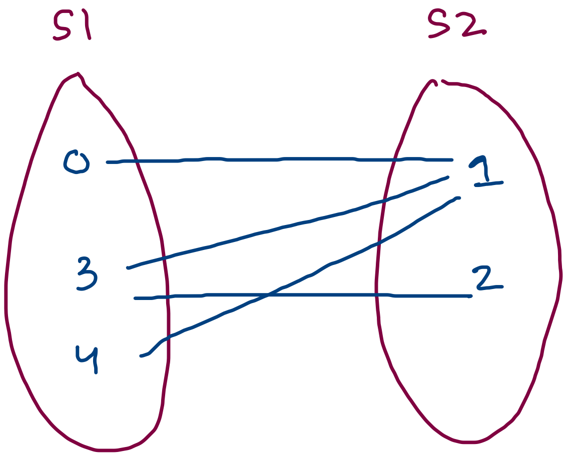






Note -> A graph is called bipartite if it is possible to split it's vertices in two sets of mutually exclusive and exhaustive vertices such that all edges are across sets.





(i) $S1 \cap S2 = \phi$

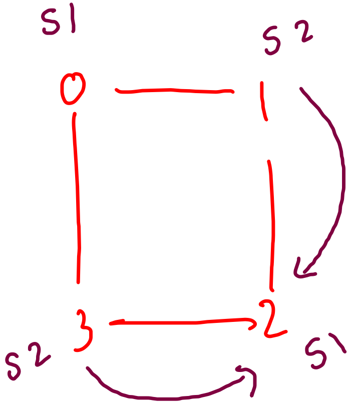
(ii) $S1 \cup S2$ = all vertices

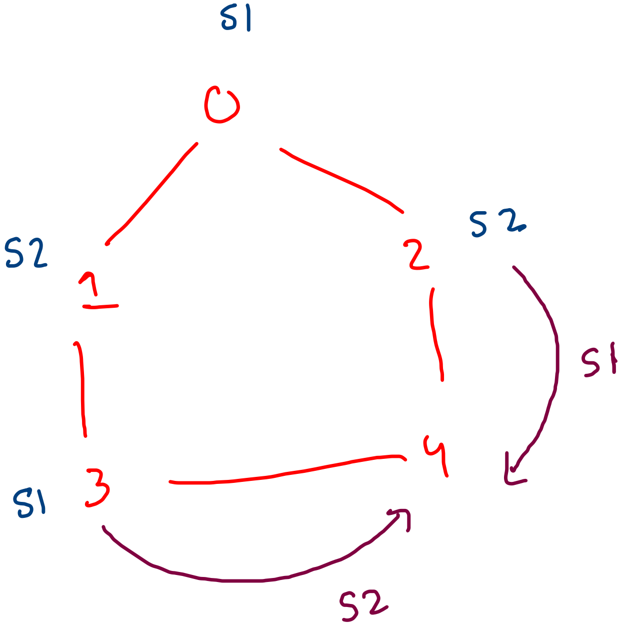
(iii) edges should be across the set

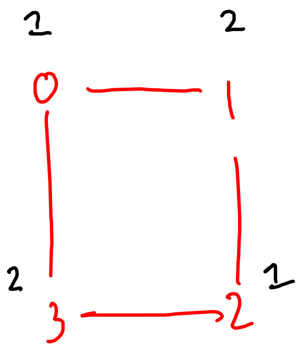
for a graph to be bipartite : every comp should be bipartite

(i) An acyclic is always bipartite

(ii) cycle — odd (non-bipartite)
— even (bipartite)

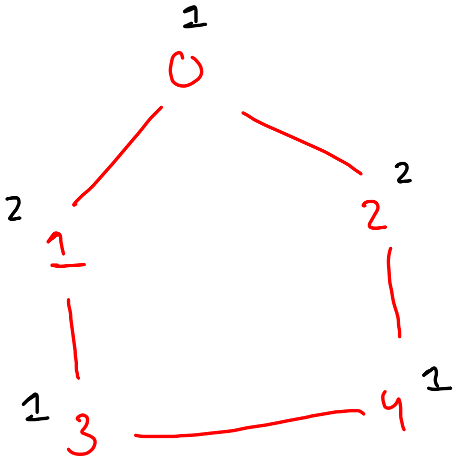






1 -> S1

2 -> S2

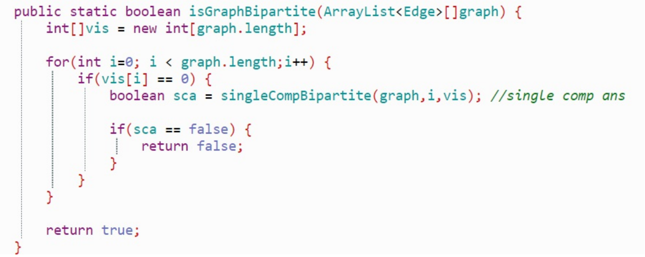


~~0,1~~	~~1,2~~	~~3,2~~	~~2,1~~	~~2,1~~

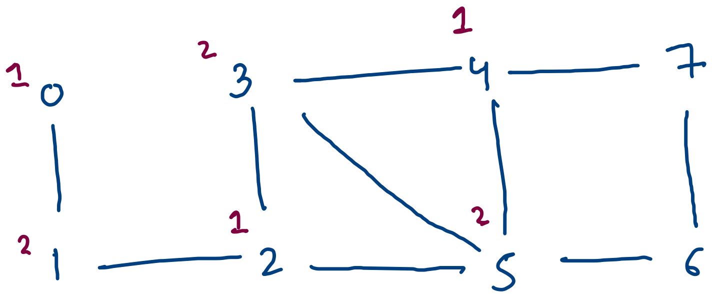
no contradiction

~~0,1~~	~~1,2~~	~~2,2~~	~~3,1~~	~~4,1~~	~~4,2~~

contraction

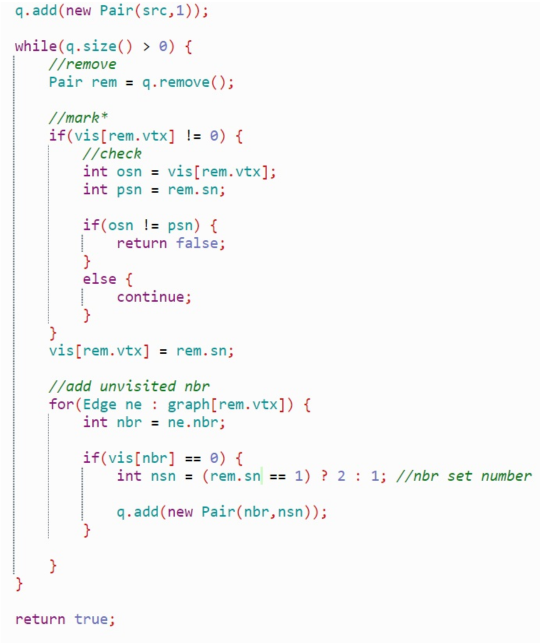


```
public static boolean isGraphBipartite(ArrayList<Edge>[]graph) {
    int[]vis = new int[graph.length];

    for(int i=0; i < graph.length;i++) {
        if(vis[i] == 0) {
            boolean sca = singleCompBipartite(graph,i,vis); //single comp ans

            if(sca == false) {
                return false;
            }
        }
    }

    return true;
}
```



osn = 2

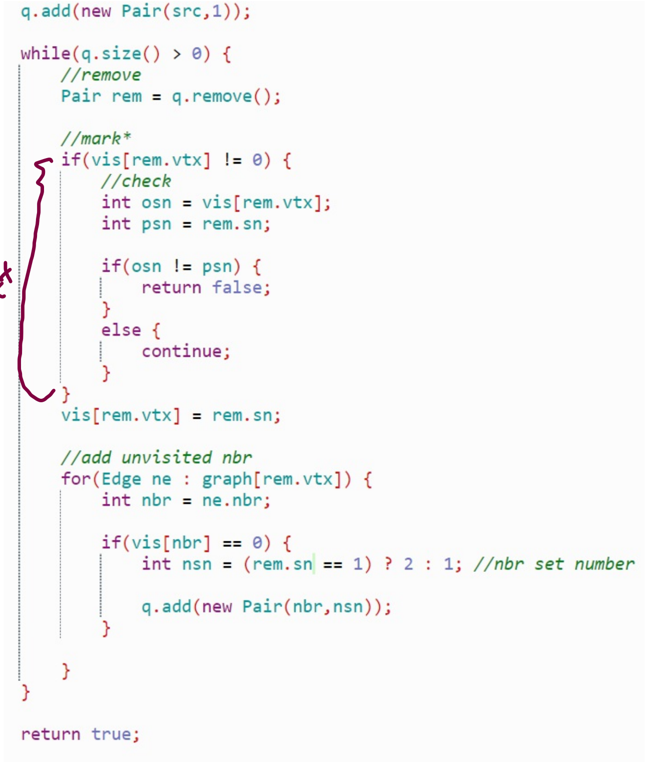
psn = 1



```
q.add(new Pair(src,1));

while(q.size() > 0) {
    //remove
    Pair rem = q.remove();

    //mark*
    if(vis[rem.vtx] != 0) {
        //check
        int osn = vis[rem.vtx];
        int psn = rem.sn;

        if(osn != psn) {
            return false;
        }
        else {
            continue;
        }
    }
    vis[rem.vtx] = rem.sn;

    //add unvisited nbr
    for(Edge ne : graph[rem.vtx]) {
        int nbr = ne.nbr;

        if(vis[nbr] == 0) {
            int nsn = (rem.sn == 1) ? 2 : 1; //nbr set number

            q.add(new Pair(nbr,nsn));
        }

    }
}

return true;
```

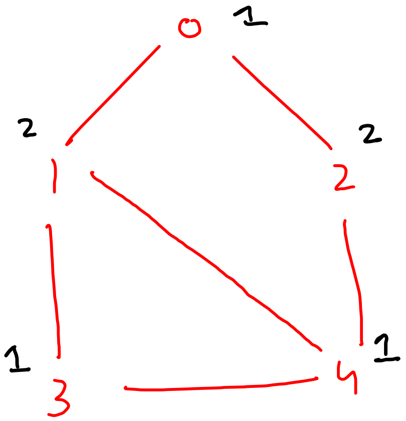
| ~~0,1~~ | ~~1,2~~ | ~~2,1~~ | ~~3,2~~ | ~~5,2~~ | ~~4,1~~ | ~~5,1~~ | 4,1 | 6,1 | 7,2 |

check



```
q.add(new Pair(src,1));

while(q.size() > 0) {
    //remove
    Pair rem = q.remove();

    //mark*
    if(vis[rem.vtx] != 0) {
        //check
        int osn = vis[rem.vtx];
        int psn = rem.sn;

        if(osn != psn) {
            return false;
        }
        else {
            continue;
        }
    }
    vis[rem.vtx] = rem.sn;

    //add unvisited nbr
    for(Edge ne : graph[rem.vtx]) {
        int nbr = ne.nbr;

        if(vis[nbr] == 0) {
            int nsn = (rem.sn == 1) ? 2 : 1; //nbr set number

            q.add(new Pair(nbr,nsn));
        }

    }
}

return true;
```

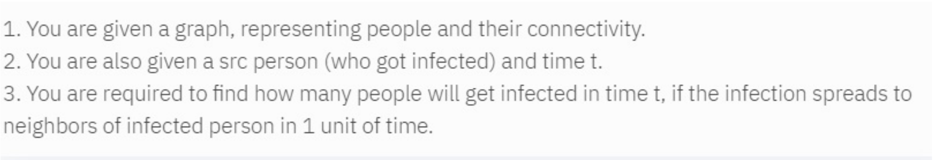
cycle detect



~~0,1~~	~~1,2~~	~~2,2~~	~~3,1~~	~~4,1~~	~~4,1~~	~~4,2~~

check

Spread of infection



1. You are given a graph, representing people and their connectivity.
2. You are also given a src person (who got infected) and time t.
3. You are required to find how many people will get infected in time t, if the infection spreads to neighbors of infected person in 1 unit of time.

src = 4

t = 2