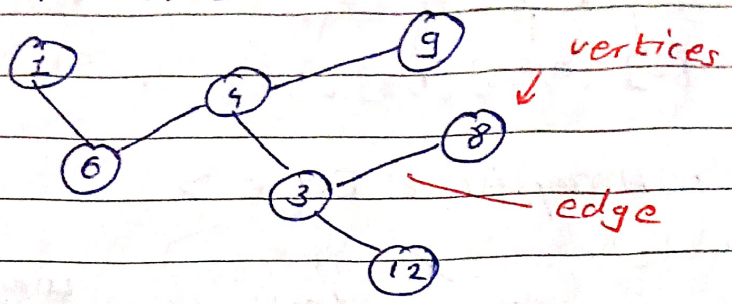
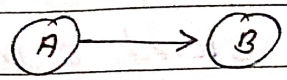
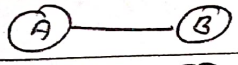
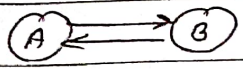
(*) Graphs :

DATE 19/09/2024

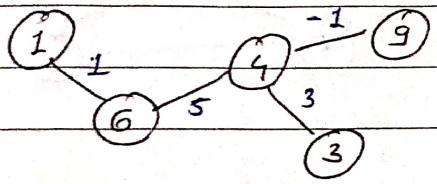
- networks of nodes



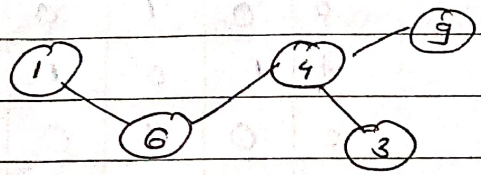
* Edges

1) Unidirectional → 
2) Undirected → 
3) Bidirected → 

* Types :



weighted graph

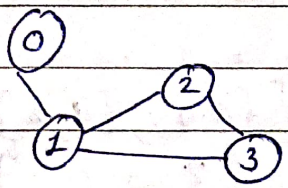


unweighted graph

* Storing a graph :

1) Adjacency List :

- List of lists



0 - (1)
1 - (0, 2, 3)
2 - (1, 3)
3 - (2, 1)

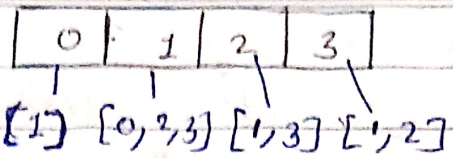
Create graph.

ArrayList < ArrayList >
Array < ArrayList > — mostly use
HashMap < Integer, ArrayList >

min edge in graph = $2V-1$

max edge in graph = ${}^{n}C_{2} = \frac{n(n-1)}{2}$

DATE



• vertex is the index

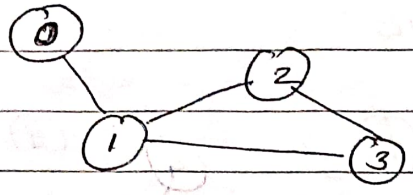
Array - ArrayList < Edge >

source destination weight.

- does not stored extra information
- optimized.

2) Adjacency matrix:

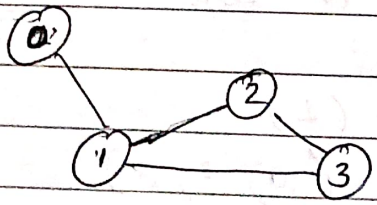
	0	1	2	3
0	0	1	0	0
1	1	0	1	1
2	0	1	0	1
3	0	1	1	0



when weight are present then stored wt instead of 1.

3) Edge list

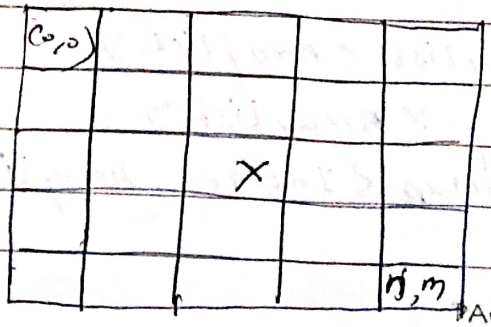
edges = { {0,1}, {1,2}, {1,3} {2,3} }



- this type structure mostly used when graph is sort.

4) implicit graph.

- mostaly use when traverse in 2D array.



(i,j-1) ← (i,j) → (i,j+1)

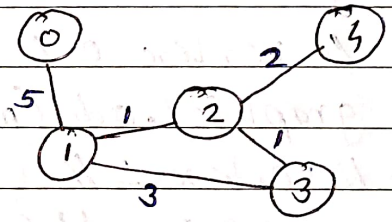
(i+1,j)

Sparse graph – no. of edge is minimum.
dense graph – no. of edge is maximum (almost cpt graph).

DATE

* Applications of graphs :
a) MAPS : (sortest path)
b) Social network : one person connect more people.
c) Delivery Network : sortest cyclic route
d) Physics & chemistry :
e) Routing algorithms.
f) Machine learning : (computation graphs, nueral network, for calculation).
g) Dependency graph :
h) computer vision : image segmentation, flood fill algorithm (fill the color in image).
i) Graph database : nebula, neo4g.
j) Research : (map humans brain (in brain present neurons).

* Create a graph :
1) Adjacency List :



ArrayList < Edge > graph;
Edge = (src, dest, wt)

0 → { 0, 1, 5 }
1 → { 1, 0, 5 }, { 1, 2, 1 }, { 1, 3, 3 }
2 → { 2, 1, 1 }, { 2, 3, 1 }, { 2, 4, 2 }
3 → { 3, 1, 3 }, { 3, 2, 1 }
4 → { 4, 2, 2 }

Note: ~~when~~ given

```
// code
static class Edges {
    int src;
    int dest;
    int wt;
    Edges(int s, int d, int w){
        src = s;
        dest = d;
        wt = w;
    }
}
public static void main(String args[]){
    int V = 5;
    ArrayList<Edges> [] graph = new ArrayList[V];
    for(int i=0; i<V; i++){
        graph[i] = new ArrayList();
    }
    // vertex 0 //
    graph[0].add(new Edges(0,1,5));
    // vertex 1 //
    graph[1].add(new Edges(1,0,5));
    graph[1].add(new Edges(1,2,1));
    graph[1].add(new Edges(1,3,3));
    // vertex 2 //
    graph[2].add(new Edges(2,1,1));
    graph[2].add(new Edges(2,3,1));
    graph[2].add(new Edges(2,4,2));
    // vertex 3 //
    graph[3].add(new Edges(3,1,3));
    graph[3].add(new Edges(3,2,1));
    // vertex 4 //
    graph[4].add(new Edges(4,2,2));
```

DATE

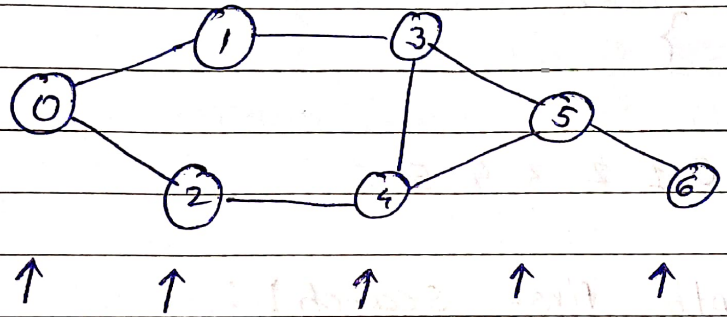
```
// print only 2th index neighbors //
for(int i=0; i< graph[2].size(); i++){
    Edges e = graph[2].get(i);
    System.out.println(e.dest);
}
}
```

O/P: 1
3
4

* Graph Traversals:

1) Breadth first search (BFS):
2) Depth first search (DFS):

1) Breadth first search:



- traverse by level wiese & used queue data structure.
- go to immediate neighbors first.

queue - | 0 | 1 | 2 | 3 | 0 | 4 | 0 | 1 | 5 | 4 | 3 | 6 | 5 | | |

visit [] = | T | T | T | T | T | T | T |
0 1 2 3 4 5 6

O/P: 0 1 2 3 4 5 6

- when remove vertex from queue then add neighbors of this vertex in queue store T or F if vertex is print or not / repeat or not

DATE

//code1
```
public static void BFS (ArrayList<Edge> graph[]){
    Queue <Integer> q = new LinkedList <>();
    boolean visit[] = new boolean[graph.length];
    q.add(0);
    while (!q.isEmpty()) {
        int curr = q.remove();
        if(!visit[curr]) {
            System.out.print(curr + " ");
            visit[curr] = true;
            for( int i=0; i< graph[curr].size(); i++){
                Edge e= graph[curr].get(i);
                q.add (e.dest);
            }
        }
    }
}
```
O(E+V)

o/p : 0 1 2 3 4 5 6

2) DFS (depth first search) :

//code1
```
public static void DFS (ArrayList<Edge> [] graph,
                        int curr, boolean[] visit) {
    System.out.print (curr+" ");
    visit[curr] = true;
    for ( int i=0; i< graph[curr].size(); i++){
        Edge e = graph[curr].get (i);
        if( !visit [e.dest]) {
            DFS ( graph, e.dest , visit );
        }
    }
}
```
O(E+V)

o/p : 0 1 3 4 2 5 6

DATE 20/9/2024

9. Has Path ?

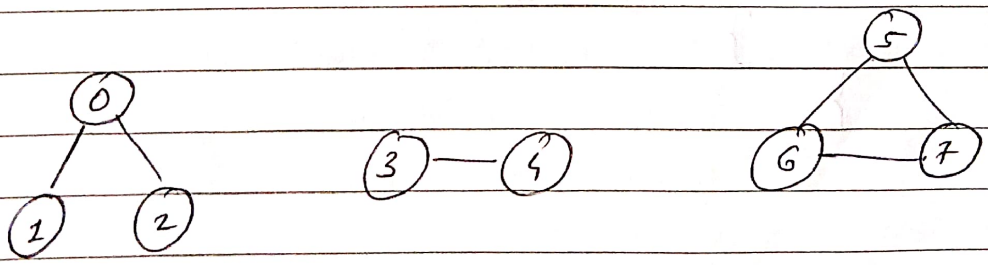
for given src & dest, tell if path is exists src to dest.

ex: src = 0 , dest = 5 ans = true.

```
//code// public static boolean hasPath(graph, int src, int dest,
                                    boolean visit[] ){
    if(src == dest) return true;
    visit[src] = true;
    for( int i=0; i< graph[src].size(); i++){
        Edge e = graph[src].get(i);
        // e.dest = neighbour //
        if (! visit[e.dest] && hasPath( graph,
                                    e.dest, dest, visit)){
            return true;
        }
    }
    return false;
}
```

* Connected components :

ex:



// These are one graph in diff, component //

0	1	2	3	4	5	6	7
1	0	0	4	3	6	5	5
2					7	7	6

DATE 22 09 2024

// This code is traverse of all component of graph //

//code//

```
public static void bst(Arraylist<Edge> graph[]){
        boolean [] visit = new boolean [graph.length];
        for(int i=0; i<graph.length; i++){
            if( !visit [i] ){
                bstUtil (graph, i, visit);
            }
        }
}
```

// This method travers each index of array //

```
public static void bstUtil (graph, cur, visit){
        queue <Integer> q = new Linkedlist<>();
        q.add (cur);
        while( !q.isEmpty() ){
            int temp = q.remove();
            if( !visit [temp] ){
                System.out.print (temp + " ");
                visit[temp] = true;
                for(int i=0; i<graph[temp].size(); i++){
                    Edge e=graph[temp].get(i);
                    q.add (e.dest);
                }
            }
        }
}
```

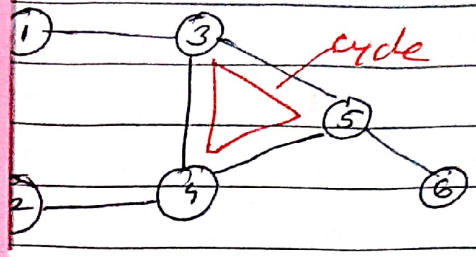
Note: detect cycle by BFS method. (191 page). undirected

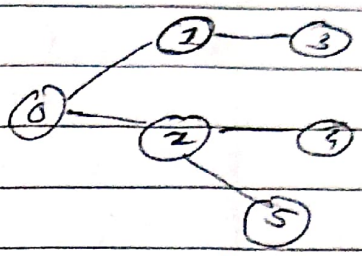
DATE 22/09/2024

* Cycle in graphs :

1) ... tion in undirected graph : (DFS) O(V+E)

Cycle detect :
1) Directed → BFS – kanns algo
→ DFS – visit & path boolean array
2) Undirected → BFS – create pair store parent node
→ DFS – par & curr in recursive fun.





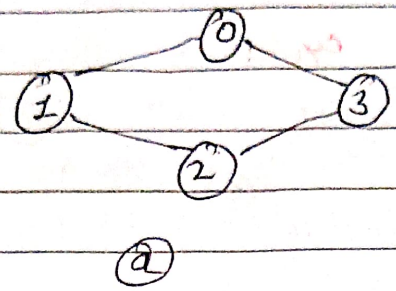
no cycle.

```
//code//
public static boolean detectCycle (graph) {
    boolean visit[] = new boolean[graph.length];
    for(int i=0; i< graph.length; i++){
        if( !visit[i] ){
            if (detectCycleUtil (graph, i, -1, visit)).
                return true;
        }
    }
    return false;
}
public static boolean detectCycleUtil (graph, curr, par, visit){
    visit[curr] = true;
    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if (visit[e.dest] && e.dest != par){
            return true;
        }else if(!vist[e.dest] &&
            detectCycleUtil (graph, e.dest, curr, visit)){
            return true;
        }
        // if par == e.dest ⇒ continue //
    }
    return false;
}
```

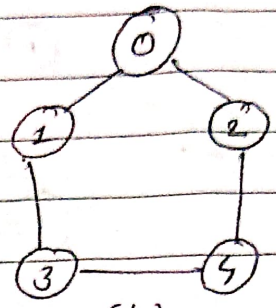
DATE

9. Bipartite :
check given graph is partite or not.

ex :

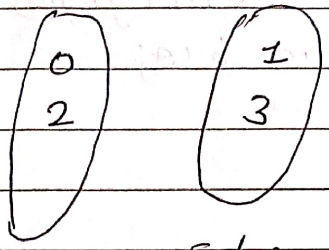


(a)



(b)

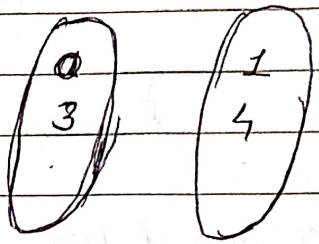
- creating two set & add element in different set when two vertex connected by edge.
- for dia.(a)



set A set B

bipartite

- for ex dia (b)



set A set B

not bipartite, because 2 not add any set

- explaination : dia (B).
- if 2 add in set A but 0 & 2 are connecte by edge and 2 add in set B but 4 & 2 are connected by edge. hence given graph is not bipartite.

approach :

fill the colour colors of node yellow & blue.
case 1 : neighbor colour — same ⇒ return false
case 2 : neighbor color - diff. ⇒ continue
case 3 : neighbor no colour → fill colour.

* dry run * code write *

DATE

```
//code
public static boolean isPartite(ArrayList<Edge> graph[]){
    int color[] = new int[graph.length];
    for(int i=0; i<color.length; i++){
        color[i] = -1;
    }
    Queue<Integer> q = new LinkedList<>();
    for(int i=0; i<graph.length; i++){
        if(color[i] == -1){
            q.add(i);
            color[i] = 0;
            while(!q.isEmpty()){
                int curr = q.remove();
                for(int j=0; j<graph[curr].size();){
                    Edge e = graph[curr].get(j);
                    if(color[e.dest] == -1){
                        if(color[curr] == 0){
                            color[e.dest] = 1;
                        }else if(color[curr] == 1){
                            color[e.dest] = 0;
                        }
                        q.add(e.dest);
                    }else if(color[curr] == color[e.dest]){
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

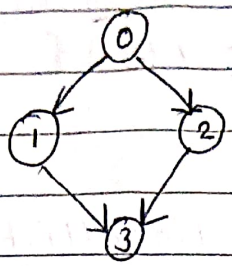
page 192 : cycle detect by BFS (Directed graph)

*dry run * write code *

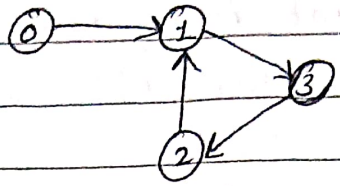
DATE

2) Cycle detection in directed graph :

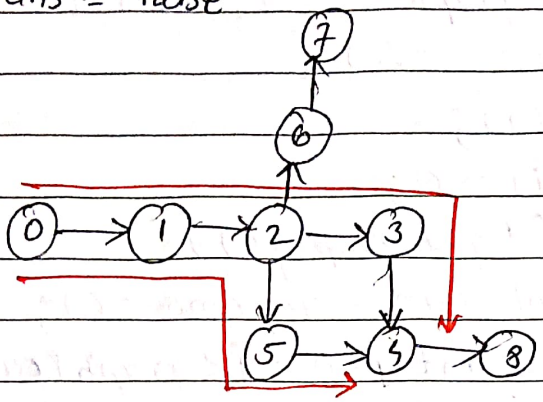
ex:



ans = false



ans = true.



track only current path.

0 1 2 3 4 — curr path

0 1 2 5 4 — curr path

```
// code:  public static boolean isCycle(graph) {
            boolean visit[] = new boolean[graph.length];
            boolean stack[] = new boolean[graph.length];

            for(int i=0; i<graph.length; i++){
                if(!visit[i]){
                    if(isCycleUtil(graph,i, visit, stack)){
                        return true;
                    }
                }
            }
            return false;
        }
```

DATE

```
public static boolean isCycleUtil(graph, cur, visit, stack){
    visit[cur] = true;
    stack[cur] = true;

    for(int i=0; i< graph[cur].size(); i++){
        Edge e = graph[cur].get(i);
        if(stack[e.dest]){
            return true;
        }
        if(!visit[e.dest] && isCycleUtil(g, e.dest, v, s))
            return true;
        }
    }
    Stack[cur]=false;
    return false;
}
```

remaining part:

DATE 17/10/2024

9) cycle detect using BFS: (undirected graph)

//code//

```
public class Pair {
    int first; int second;
    Pair(int f, int s) {
        first = f;
        second = s;
    }
}

public boolean BFS(adj, int curr, boolean visit[]){
    Queue<Pair> q = new LinkedList();
    q.add(new Pair(curr, -1));
    visit visit[curr] = true;

    while(!q.IsEmpty()) {
        int node = q.peek().first;
        int par = q.peek().second;
        q.remove();

        for(int i=0; i<adj.get(node).size(); i++){
            int neighbor = adj.get(node).get(i);
            if(par == neighbor) {
                continue;
            }
            if(visit[neighbor]) return true;

            q.add(new Pair(neighbor, node));
            visit[neighbor] = true;
        }
    } return false;
}
```

DATE

```
public boolean isCycle(ArrayList<ArrayList<Integer>> adj){
    boolean visit[] = new boolean[adj.size()];
    for(int i=0; i< adj.size(); i++){
        if(!visit[i]){
            if(BFS(adj, i, visit){
                return true;
            }
        }
    }
    return false;
}
```

Q. Cycle detect using BFS : (directed graph)

approach : (Kann's algorithm)

1) Calculate indegree of all vertices
2) Add 0 degree indese in queue
3) Perform BFS & count the visited ~~vertese~~ vertex.
4) ~~Check~~ if countvertex == V return false otherwise return true (cycle is exist).

//code//

```
public boolean isCyclic(int V, ArrayList<ArrayList<Integer>> adj)
    int [] indeg = new int[v];
    for(int i=0; i<V; i++) {
        for(int j : adj.get(i)){
            ~~indeg[adj.get(i)].~~
            indeg[j]++;
        }
    }

    Queue<Integer> q = new LinkedList();
```

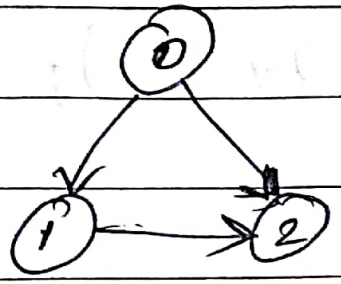
DATE

```
for(int i=0; i<indeg.length; i++){
    if(indeg[i]==0){
        q.add(i);
    }
}
// BFS //
int count=0;
while(!q.isEmpty()){
    int node = q.remove();
    count++;
    // reduce indegree of neighbors //
    for(int neighbor : adj.get(node)){
        indeg[neighbor]--;
        if(indeg[neighbor]==0){
            q.add(neighbor);
        }
    }
}
return count!=V;
}
```

count == V : not cycle
count != V : present.
count < V : present cycle.

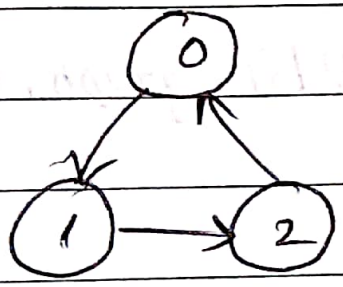
Q. Topological Sorting :

- Topological sorting only Directed Acyclic Graph (DAG) is directed graph with No cycle.
- It is linear order of vertices such that every directed edge $u \rightarrow v$, the vertex u comes before v in the order



→ 0 1 2

→ DAG

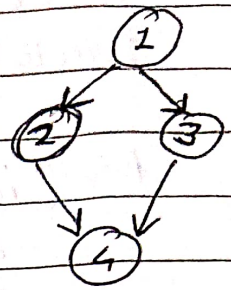


→ non - DAG

DATE 23/09/2025

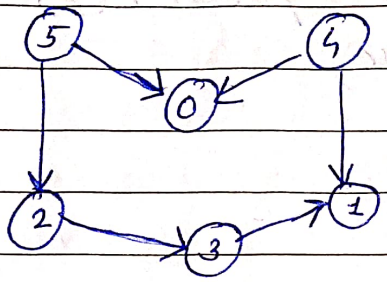
Topological sort use for checks dependency.

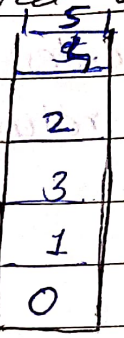
Ex: Action 1 - boil water
any work do first { Action 2 - add masala
Action 3 - add maggie
Action 4 - add serve maggie



o/p : 1 2 3 4 also 1 3 2 4

approach: used modified DFS.
- used stack for stored ~~curr~~ data



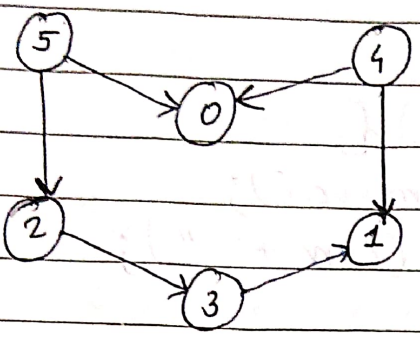


o/p: 5 4 2 3 1 0

```
//code// public static void topologicalSort(graph){
        boolean visit[] = new boolean[graph.length];
        Stack<Integer> s = new Stack();
        for(int i=0; i<graph.length ; i++){
            if(!visit[i]){
O(V+E)                  topologicalSortUtil(graph,i,visit,s);
            }
        }
        while(!s.isEmpty()){
            System.out.print(s.pop()+" ");
        }
        System.out.println();
    }
    public static void topologicalSortUtil(graph, curr, visit, s){
        visit[curr] = true;
```

```
for(int i=0; i<graph.[curr].size(); i++) {
    Edge e = graph[curr].get(i);
    if(!visit[e.dest]) {
        topoLogicalSortUtil(graph, e.dest, visit, s);
    }
} s.push(curr);
}
```

* Topological sort using BFS : (Kahn's algorithm)

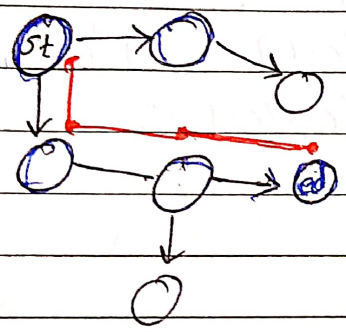


approach :

1. first calculate indegree & outdegree of every node/vertex.

ex: (0) → indegree = 2
outdegree = 0.

Fact : A DAG has atleast one vertex with in-degree 0 & one vertex with out-degree 0.



(st) - indegree - 0
(ed) - outdegree - 0.
- there is logest path hence starting pt indegree 0 & ending pt outdegree 0.

2. then add 0 indegree vertex in queue
3. apply BFS & remove queue element & subtract indegree of there neighbors.

	0	1	2	3	4	5
indegree =	2	2	1	1	0	0

DATE

```
// code //  public static void topSort(ArrayList<Edge> graph){
        int indeg[] = new int[graph.length];
        calDeg(graph, indeg);
        Queue<Integer> q = new LinkedList<>();
        for(int i=0; i<indeg.length; i++){
            if(indeg[i] == 0){
                q.add(i);
            }
        }
        // bfs //
        while(!q.isEmpty()){
            int curr = q.remove();
topological sort -> System.out.print(curr + " ");
print print
            for(int i=0; i<graph[curr].size(); i++){
                Edge e = graph[curr].get(i);
                indeg[e.dest]--;
                if(indeg[e.dest] == 0){
                    q.add(e.dest);
                }
            }
        }
    }
    public static void calDeg(graph, int indeg[]){
        for(int i=0; i<graph.length; i++){
            int v=i;
            for(int j=0; j<graph[v].size(); j++){
                Edge e = graph[v].get(j);
                indeg[e.dest]++;
            }
        }
    }
```

classmate

o/p : 5 0 2 3 1 :

DATE

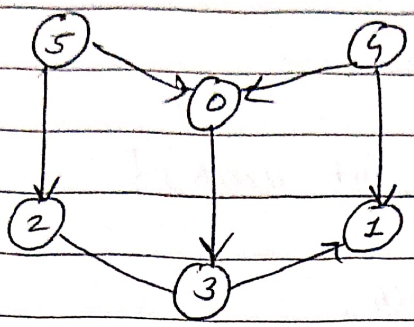
```
// code //  public static void topSort (ArrayList<Edge> graph){
        int indeg [] = new int[graph.length];
        calDeg (graph, indeg);
        Queue < Integer > q = new LinkedList<>();
        for( int i=0; i< indeg.length; i++){
            if ( indeg [i] == 0){
                q.add (i);
            }
        }
        // bFs //
        while (! q.isEmpty ()) {
            int curr = q.remove ();
            System.out.print (curr + " ");
            for( int i=0; i< graph[curr].size(); i++){
                Edge e= graph[curr].get(i);
                indeg [e.dest] --;
                if( indeg [e.dest] == 0) {
                    q.add (e.dest);
                }
            }
        }
    }
    public static void calDeg ( graph, int indeg []){
        for( int i=0; i< graph.length; i++){
            int v=i;
            for( int j=0; j< graph [v].size(); j++){
                Edge e= graph [v].get (j);
                indeg [e.dest] ++;
            }
        }
    }
```

topological sort → ~~pri~~ print

o/p : 4 5 0 2 3 1 ;

DATE

Q. All paths from source to target :



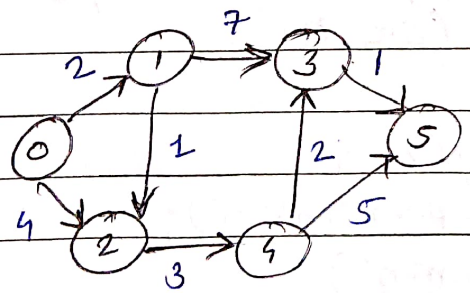
src = 5, dest = 1

o/p : 5 0 3 1
5 2 3 1 $O(v^v)$.

```
//code// public static void printAllPath(graph, src, dest, String path){
        if( src == dest) {
            System.out.println(path + dest);
            return;
        }
        for( int i=0; i< graph[src].size(); i++){
            Edge e = graph[src].get(i);
            printAllPath( graph, e.dest, dest, path+src);
        }
    }
```

Q Dijkstra's algorithm : (weighted graph).
- Sortest paths from the source to all vertices

Q ex :



src = 0

- This question base on gready algorithm.

ans :

0	2	3	8	6	9
0	1	2	3	4	5

This logic same for directed & undirected graph.
(~~dijekstra~~ (dijkstra algo).

DATE

```
// code
static class pair implements Comparable<pair> {
    int n;
    int path;

    public pair(int n, int path){
        this.n = n;
        this.path = path;
    }
    @Override
    public int compareTo(pair p2){
        return this.path - p2.path;
    }
}

public static void dijkstra(graph, int src) {
    int dist[] = new int[graph.length];
    for(int i=0; i<graph.length; i++){
        if(i!=src){
            dist[i] = Integer.MAX_VALUE;
        }
    }
    boolean visit[] = new boolean[graph.length];
    PriorityQueue<pair> pq = new PriorityQueue<>();
    pq.add(new pair(src, 0));

    while(!pq.isEmpty()) {
        pair cur = pq.remove();
        if(!visit[cur.n]){
            visit[cur.n] = true;
            for(int i=0; i<graph[cur.n].size(); i++){
                Edge e = graph[cur.n].get(i);
```

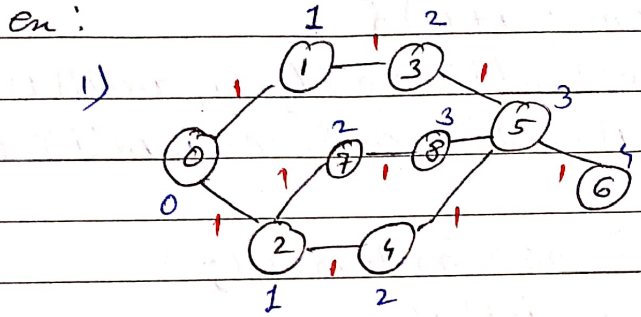
also use inbuilt function Arrays.fill()

DATE

```
int u = e.src;
int v = e.dest;
int wt = e.wt;

if(dist[u] + wt < dist[v]){
    dist[v] = dist[u] + wt;
    pq.add(new pair(v, dist[v]));}
}
}
}
}
```

Q. Shortest path in Undirected graph from source:

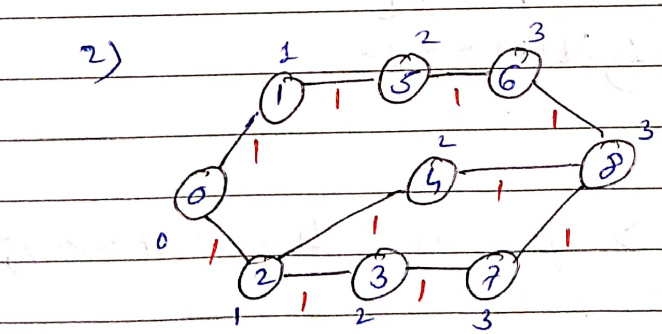
ex:



ex- source = 0.

o/p:

0	1	1	2	2	3	4	2	3



source = 0.

o/p:

0	1	1	2	2	2	3	3	3

traver levelwise (use BFS).

approach :

1) create distance arry array to stored distace sorce to vertex/node. & its initialise by -1 & apply BFS.

//code //

```
public static int[] ShortestPath(int[][] arr, int n,
                                 int m, int src) {
    Arraylist<Edge> graph = new Arraylist[n];
```

edge

m = length.
n = edge

```
    for(int i=0; i<n ; i++){
        graph[i] = new ArrayList();
    }
    // create graph //  length of 2D matrix
    for(int i=0; i<m ; i++){
        graph[arr[i][0]].add(new Edge(i, arr[i][1]));
        graph[arr[i][1]].add(new Edge(i, arr[i][0]));
    }
    boolean visit[] = new boolean[n];
    int dist[] = new int[n];
    // initialise by -1 //
    for(int i=0; i<n ; i++){
        if(src != i)  dist[i] = -1;
    }

    Queue<Integer> q = new LinkedList<>();
    q.add(src);
```

~~dist sorce to sorce is 0~~ ⇒ ~~dist[src] = 0~~

```
    visit[src] = true;
```

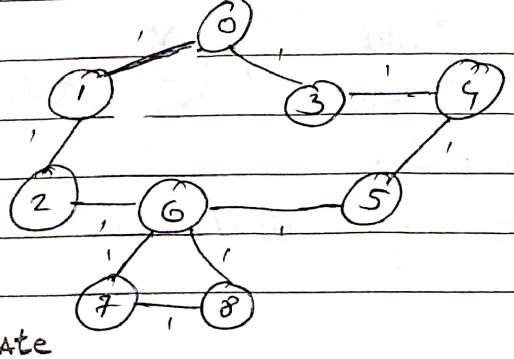
```
while( !q.isEmpty() ) {
    int curr = q.remove();
    for( int i=0; i< graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if( !visit[e.dest]){
            visit[e.dest] = true;
            q.add(e.dest);
            dist[e.dest] = dist[curr] + 1;
        }
    }
}
return dist;
}
```

Note: when input 2D array & you convert in graph / connect edge. (Undirected graph):

arr = [0,1], [0,3], [3,4], [4,5], [5,6], [1,2], [2,6], [6,7], [7,8], [6,8]].

- arr[i][0] → are the index/node of graph.
- arr[i][1] → are the data of arr[i][0] th index.
- size of graph/edge is max number in the arr[i][1] = 8

0 → 0, 1
1 → 0, 3
2 → 3, 4
3 → 4, 5
4 → 5, 6
5 → 1, 2
6 → 2, 6
7 → 6, 7
8 → 7, 8
9 → 6, 8



DATE 25/09/2025

Un-directed graph:

```
for(int i=0; i<arr.length; i++){
    graph[arr[i][0]].add(arr[i][1]);
    graph[arr[i][1]].add(arr[i][0]);
}
```

0	1	2	3	4	5	6	7	8
1	0	1	0	3	4	5	6	7
3	2	6	4	5	6	2	8	6
						7		
						8		

if Edge class present:

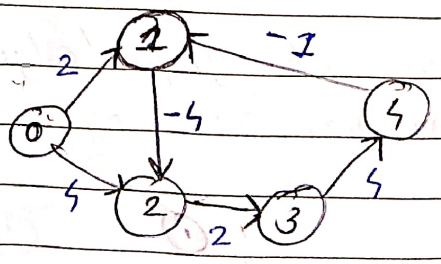
```
for(int i=0; i<arr.length; i++){
    graph[arr[i][0]].add(new Edge(arr[i][0], arr[i][1], 1));
    graph[arr[i][1]].add(new Edge(arr[i][1], arr[i][0], 1));
}
```

directed graph:

```
for(int i=0; i<arr.length; i++){
    graph[arr[i][0]].add(arr[i][1]);
}
```

Q. Bellman Ford algorithm:

- Shortest paths from the source to all vertices (negative edge).



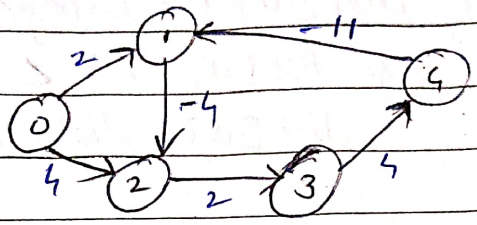
- Dijkstra's algorithm only positive wt graph ✓ but ~~this~~ ~~Ans~~ when wt is -ve Dijkstra's algorithm fail.
- This problem solve by Bellman ford algorithm.

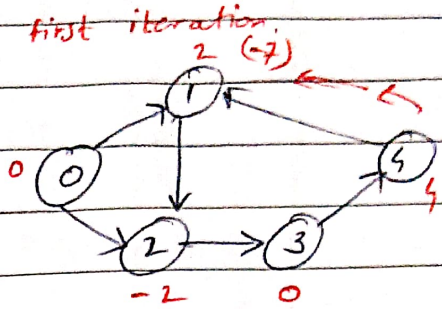
for all edge (u,v).
    if (dist[u] + wt(u,v) < dist[v]) {
        dist[v] = dist[u] + wt(u,v);

} perform this operation V-1 times.

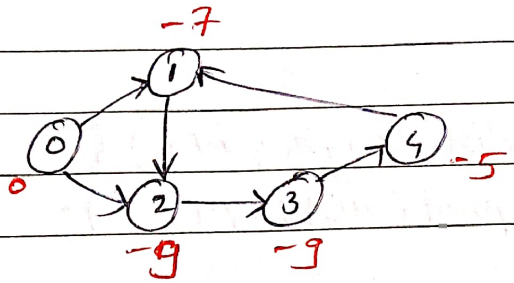
Note: this algorithm doesn't work for -ve wt. graph cyclic graph.

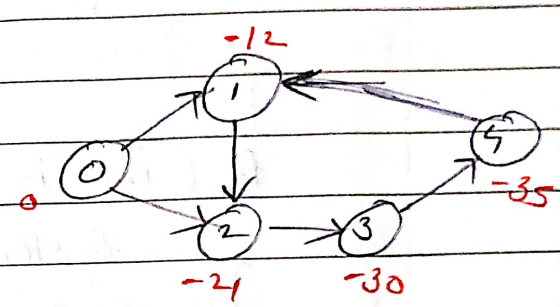
exc:





second iteration.





- this loop is continuous run.

//code// public static void bellmanford (graph, int src) {
    int dist[] = new int[graph.length];
    for(int i=0; i< dist.length; i++) {
        if (i != src) {
            dist[i] = Integer.MAX_VALUE;
        }
    }
    int v = graph.length;
    for (int i=0; i< v-1; i++){ - // O(v)
        //edge - O(E) //
        for(int j=0; j< graph.length; j++){
            for(int k=0; k< graph[j].size(); k++

```
Edge e = graph[i].get(k);

int u = e.src;
int v = e.dest;
int wt = e.wt;
// relaxation //
if ( dist[u] != Integer.MAX_VALUE
    && dist[u] + wt < dist[v] ) {
        dist[v] = dist[u] + wt;
    }
  }
 }
}

for( int i=0; i< dist.length; i++) {
    System.out.print( dist[i]+" " );
}
System.out.println( );
}
```

O(V*E)

o/p : 0 2 -2 0 5

// another method //

```
public static void bellmanford ( graph, int src, int V){
    int dist [] = new int[v];
    for ( int i=0; i< dist.length; i++) {
        if( i != src ){
            dist[i] = Integer.MAX_VALUE;
        }
    }
```

DATE

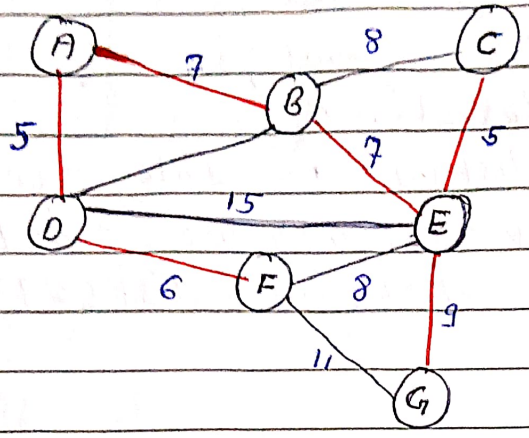
```
for( int i=0; i< V-1; i++){
    // edge //  O(E)
    for( int j=0; j< graph.size(); j++){
        Edge e = graph.get(j);

        int u = e.src;
        int v = e.dest;
        int wt = e.wt;
        // relaxation //
        if( dist[u] != Integer.MAX_VALUE
            && dist[u]+ wt < dist [v] ){
                dist[v]= dist[u]+wt;
        }
    }
    // print //
    for ( int i=0 ; i< dist.length; i++){
        System.out.print (dist[i] +" ");
    }
    System.out.println();
}
public static void main (String args[]){
    int V= 5;
    ArrayList <Edge> edges = new ArrayList<>();
    // create edge graph // call BellmanFord //
}
public static void createGraph(ArrayList<Edge> graph){
    graph.add(new Edge(0,1,2));
    graph.add( new Edge(0,2,4));
    "                (1,2,-4));
    "                (2,3,2));
    "                (3,4,4));
    "                (4,1,-1));
}
```

DATE 27/09/2025

* Minimum Spanning tree (MST):

Def: It is the subset of edges - weighted undirected graph that connects all the vertices together without any cycles and with the minimum posible total edge weight.

ex:



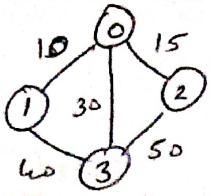
cond^n:
1) no cycle perform
2) all vertices are connected
3) min wt of connected vertices.

q. Prim's algorithm
- find the min cost of MST

approach: create pair & store it vertices & cost

//code// Static class Pair implements Comparable <Pair> {

```
    int vertice;
    int cost;

    public Pair(int v, int c){
        vertice = v;
        cost = c;
    }
    @Override
    public int compareTo(Pair p2){
        return this.cost - p2.cost;
    }
}
```



DATE

```
public static void mstCost(ArrayList<Edge> graph[]){
    boolean visit[] = new boolean[graph.length];
    PriorityQueue<Pair> pq = new PriorityQueue<>();
    pq.add(new Pair(0,0));
    int totalCost = 0;

    while(!pq.isEmpty()){
        Pair curr = pq.remove();
        if(!visit[curr.vertice]){
            visit[curr.vertice] = true;
            totalCost += curr.cost;

            for(int i=0; i<graph[curr.vertice].size(); i++){
                Edge e = graph[curr.vertice].get(i);
                pq.add(new Pair(e.dest, e.wt));
            }
        }
    }
    System.out.println("final cost of MST = " + totalCost);
}
```

O/p : final cost of MST = 55.

Q Corona spread

- find the time of hospital when corona is virus is spread in all patient.

ex:

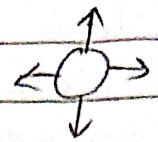
1	2	1	1	0	1
1	1	0	1	1	2
0	1	0	2	1	1
1	1	0	1	0	1
1	0	1	2	0	1

0 - Empty room
1 - normal patient
2 - corona patient

DATE

Corona spread in 4 direction

- in 1 unit - each covid patient spread corona near by there.



2	2	2	1	0	2
1	2	0	2	2	2
0	1	0	2	2	2
1	1	0	2	0	1
1	0	2	2	0	1

Unit - 1

2	2	2	2	0	2
2	2	0	2	2	2
0	2	0	2	2	2
1	1	0	2	0	2
1	0	2	2	0	1

Unit - 2

-- continue.

ans = unit 5

← This question solve by BFS.

```
// code
static int r;
static int c;
static class Pair{
        int st;
        int ed;
        Pair (int r, int c){
            st = r;
            ed = c;
        }
}
public static boolean valid (int i, int j){
        return i<r && i>=0 && j<c && j>=0;
}
public static int covidSpread ( int [][] hosp){
        r= hosp.length;
        c= hosp[0].length;
        Queue <Pair> q = new LinkedList<>();
```

```
for (int i=0; i<r; i++){
    for (int j=0; j<c; j++){
        if ( hosp[i][j] == 2){
            q.add(new Pair(i,j));
        }
    }
}
if(q.isEmpty()) return 0;

int time=0;
while(!q.isEmpty()){
    time++;
    int currPatient = q.size();

    while(currPatient>0){
        Pair curr = q.remove();
        int i= curr.st;
        int j= curr.ed;

        int row[] = {-1,1,0,0};
        int col[] = {0,0,-1,1};
        for(int k=0; k<4; i++){
            int newR = i+row[k];
            int newC= j+col[k];
            if( valid[newR,newC) &&
                hosp[newR][newC]==1){
                hosp[newR][newC] = 2;
                q.add(new Pair(newR,newC));
            }

        }
        currPatient--;
    }
```

```
for(int i=0; i<r; i++){
    for(int j=0; j<c; j++){
        if(hasp[i][j]==1){
            return -1;
        }
    }
}
return time-1;
}
```

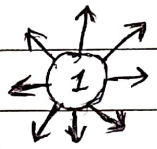
Q. Find the number of Island :

ex:

1	1	1	0	1	1
0	1	1	0	0	1
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	0	1	1

total island = 3.

cond^n :
- if 1-1 near each other then there are one island



Note: In this question check 8 cond^n for checking near 1-1

// code //

```
static int r,c;
static class Pair {
    int row;
    int col;
    Pair(int r, int c){
        row=r;
        col=c;
    }
}
public static boolean valid(int rn, int cn){
    return rn<r && rn>=0 && cn<c && cn>=0;
}
```

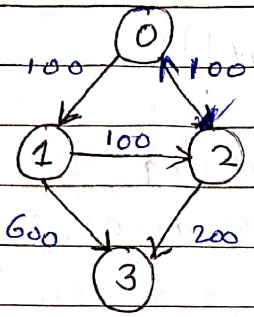
```
public static int FindIsland ( int [][] land ){
    r = land.length;
    c = land[0].length;
    boolean [][] visit = new boolean[r][c];
    Queue < Pair > q = new boolean[r][c];
    int lNum = 0;

    for(int i=0; i<r; i++ ){
        for( int j=0; j<c; j++){
            if( ! visit[i][j] && land[i][j] == 1) {
                visit[i][j] = true;
                q.add (new pair(i,j));
                lNum++;
                while( ! q.isEmpty()) {
                    Pair curr = q.remove();

                    int n = curr.row;
                    int m = curr.col;
                    int newR[] = { 0,0,1,1,1,-1,-1,-1};
                    int newC[] = {1,-1,1,-1,0,1,0,-1};
                    // Check only cond" near 1-1 //
                    for( int k=0; k<8; k++){
                        if( valid(n+newR[k], m+newC[k]) &&
                            !visit[n+newR[k]][m+newC[k]]){

                            visit[n+newR[k]][m+newC[k]] = true;
                            q.add (new Pair(n+newR[k], m+newC[k]);
                        }
                    }
                }
            }
        }
    }
    return lNum;
}
```

DATE 28/09/2024

Q. Cheapest flights within k stops

- There are n cities connected by some number of flights. you are given an array flights where flights[i] = [from, to, price] indicates that there is a flight.
- you are also given three integer src, dest & k return the cheapest price from src to dest with at most k stops. If there is no such route, return -1;
(all value are +ve).

ex:



src = 0, dest = 3, k = 1

cost = 700

//code//

```
public static int cheapestFlight(int n, int flight[][],
                                 int src, int dest, int k){
    ArrayList<Edge> graph[] = new ArrayList<>();
    CreateGraph(flight, graph);

    int dist[] = new int[n];
    for(int i=0; i<n; i++){
        if(i!=src){
            dist[i] = Integer.MAX_VALUE;
        }
    }

    Queue<Info> q = new LinkedList<>();
    q.add(new Info(src,0,0));
```

```
while (!q.isEmpty()){
    Info curr = q.remove();
    if(curr.stops > k){
        break;
    }
    for(int i=0; i<graph[curr.v].size(); i++){
        Edge e = graph[curr.v].get(i);
        int u = e.src;
        int v = e.dest;
        int wt = e.wt;

        if(curr.cost+wt < dist[v] && curr.stops <= k){
            dist[v] = curr.cost + wt;
            q.add(new Info(v, dist[v], curr.stops+1));
        }
    }
}
if(dist[dest] == Integer.MAX_VALUE){
    return -1;
}else{
    return dist[dest];
}
}
Static class Info{
    int v,cost,stops;

    public Info(int v, int c, int s){
        this.v = v;
        cost = c;
        stops = s;
    }
}
```

classmate

o/p : 700.

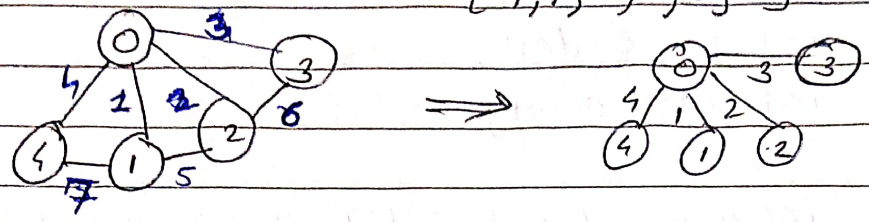
DATE

9. Connecting Cities :

- find the minimum cost of connecting all cities on the map (MST).

ex: cities [][] = {{0, 1, 2, 3, 4},
{1, 0, 5, 0, 7},
{2, 5, 0, 6, 0},
{3, 0, 6, 0, 0},
{4, 7, 0, 0, 0}}

ans = 10



- for this question use queue
- only store edge cost & dest.

//code//

```
static class Edge implements Comparable<Edge>{
    int dest;
    int cost;
    public Edge(int d, int c){
        this.dest = d;
        this.cost = c;
    }
    @Override
    public int compareTo(Edge e2){
        return this.cost - e2.cost;
    }
}
public static int connectingCities(int cities[][]){
    PriorityQueue<Edge> pq = new PriorityQueue();
    boolean[] visit = new boolean[cities.length];
    pq.add(new Edge(0,0));
    int finalCost = 0;
```

DATE

```
while (!pq.isEmpty()) {
    Edge curr = pq.remove();
    if (!visit[curr.dest]) {
        visit[curr.dest] = true;
        finalCost += curr.cost;

        for (int i=0; i< cities[curr.dest].length; i++){
            if (cities[curr.dest][i] != 0) {
                pq.add(new Edge(i, cities[curr.dest][i]));
            }
        }
    }
} return finalCost;
}
```

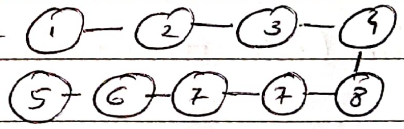
* ~~Obj~~ Disjoint set DS : (Union).

- In this two operation find & Union.

Ex: set(1) = 1 — 2 — 3 — 4

set(2) = 5 — 6 — 7 — 8

Union of 4 & 8 : Union(4, 8) =



} one set.

operation :

find (2) - ans: set(1) } before Union(4, 8)

find (6) - ans: set2 }

find (2) - ans: set1 } after union(4, 8).

find (6) - ans: set1 }

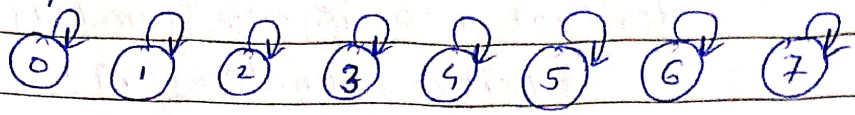
DATE 30/09/2024

- Implementation (optimized):
  Parent + union by rank

ex: n=8.

	0	1	2	3	4	5	6	7
par =	0	1	2	3	4	5	6	7

initially par node is parent itself.



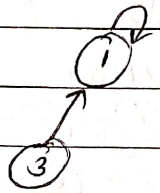
	0	1	2	3	4	5	6	7
rank =	0	0	0	0	0	0	0	0

initially rank is zero of all node.

- find() - this method is use to find parent of a given node.
- Union() - it is use to join two groups

for example some operation:

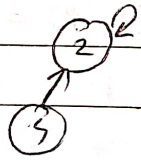
1) Union (1,3):



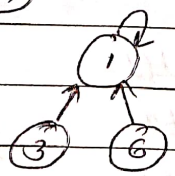
- same degree any node make parent/leader.

O(4k)2) find(3) - o/p: 1 - o/p is leader.

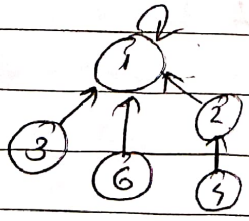
O(4k)3) Union (2,4) -



3) Union(3,6) -



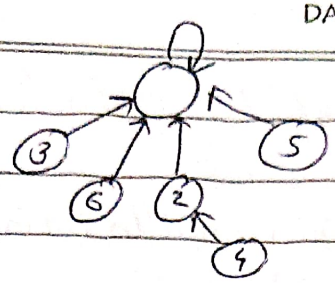
5) Union (1,4) -



6) find(4) - 1

DATE

7) Union (1,5) –



	0	1	2	3	4	5	6	7
Par =	0	1	1	1	1	1	1	7
rank =	0	2	1	0	0	0	0	0

//code//

```
static int n=7;
static int par[] = new int[n];
static int rank[] = new int[n];
public vo static void init() {
    for(int i=0; i<n; i++){
        par[i] = i;
    }
}
public static int find(int x) {
    if( x = par[x] ) {
        return x;
    }
    return par[x] = find[par[x]];
}
public static void union(int a, int b){
    int parA = find(a);
    int parB = find(b);
    if( rank[parA] == rank[parB]){
        par[parB] = parA; rank[parA]++;
    } else if( rank[parA] < rank[parB] ){
        par[parA] = parB;
    }else {
        par[parB] = parA;
    }
}
```

```
public static void main (String [] args){
    init ();
    System.out.println (find (3));
    Union (1, 3);
    System.out.println ( find (3));
    Union (2,4);
    Union (3,6);
    Union (1,4);
    System.out.println( find(3));
    System.out.println ( find (4));
    Union (1,5);
}
```

o/p: 3 1 1 1 – (in next line).

Q. Kruskal algorithm : (MST Greedy) O(V+E log E)

– find min cost of src to dest.

```
// code
static class Edge implements Comparable<Edge>{
    int src;
    int dest;
    int wt;
    public Edge (int s, int d, int w){
        src = s;
        dest = d;
        wt = w;
    }
    @Override
    public int compareTo( Edge e2){
        return this.wt - e2.wt;
    }
}
```

DATE

```
public static void createGraph(ArrayList<Edge> edge)
        edge.add(new Edge(0, 1, 10));
        edge.add(new Edge(0, 2, 15));
        edge.add(new Edge(0, 3, 30));
        edge.add(new Edge(1, 3, 40));
        edge.add(new Edge(2, 3, 50));
}
Static int n = 4;
static int par[] = new int[n];
static int rank[] = new int[n];

public static void kruskalAlg(ArrayList<Edge> edge
                              int V) {
    collection.sort(edge);  // O(ElogE)
    int cost = 0;

    for(i=0; count < V-1; i++) {    // O(V)
        Edge e = edge.get(i);
        // src, dest, wt //
        int parA = find(e.src);
        int parB = find(e.dest);
        if( parA != parB){
            union(e.src, e.dest);
            cost += e.wt;
            count++;
        }
    }
    System.out.println("min cost : " + cost);
}
```

DATE

Imp ° Create graph from 2D matrix.

ex:
```
matrix [][] = { { 0, 1, 1, 0 },
                { 1, 0, 0, 1 },
                { 1, 0, 0, 1 },
                { 0, 1, 1, 0 } };
```

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

- i & j are vertices & matrix[i][j] = value hence value is wt of edge bet i & j vertices.

edge graph:

```
ArrayList<Edge> graph = new ArrayList<>();
for(int i=0; i<matrix.length; i++){
    for(int j=0; j<matrix[i].length; i++){
        if(matrix[i][j]!=0){
            graph.add(new Edge(i, j, matrix[i][j]));
        }
    }
}
```

```
ArrayList<Edge> graph[] = new ArrayList[3];
for(int i=0; i<matrix.length; i++){
    for(int j=0; j<matrix[i].length; i++){
        if(matrix[i][j]!=0){
            graph[i].add(new Edge(i, j, matrix[i][j]));
        }
    }
}
```

DATE 01 10 2025

* Flood fill Algorithm :

- fill the color in the image. image is given in the form of grid [][].
  ∴ 1 - pixel present
  ∴ 0 - pixel not present.

ex: image = [[1,1,1], sr = 1
[1,1,0], sc = 1
[1,0,1]] color = 2,

o/p : image = [[2,2,2]
[2,2,0]
[2,0,1]]

```
//code// public static void helper(int[][] img, int sr, int sc,
                                    int color, visit, int orgCol){
    if( sr<0 || sc<0 || sr>=img.length ||
        sc>= img[0].length || visit[sr][sc] || img[sr][sc]!=orgCol
            return ;
    }
    img[sr][sc] = color;
    // left //
    helper(img, sr, sc-1, color, visit, orgCol);
    // right //
    helper(img, sr, sc+1, color, visit, orgCol);
    // up
    helper(img, sr+1, sc, color, visit, orgCol);
    //
    helper(img, sr-1, sc, color, visit, orgCol);
}
```

DATE

```
public static int[][] floodFill(int[][] img, int sr,
                                int sc, int color){
    boolean visit[][] = new boolean[img.length]
                                   [img[0].length];
    helper(img, sr, sc, color, visit, img[sr][sc]);
    return img;
}
```

Q - Floyd warshall Algorithm :

```
// code //
public void shortestPath(int[][] matrix){
    int n = matrix.length;
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            if(matrix[i][j]==-1){
                matrix[i][j] = Integer.MAX_VALUE;
            }
        }
    }
    for(int k=0; k<n; k++){
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                if(matrix[i][k]==Integer.MAX_VALUE ||
                   matrix[k][j]==Integer.MAX_VALUE){
                    continue;
                }
                matrix[i][j]= Math.min(matrix[i][j],
                              matrix[i][k]+matrix[k][j]);
            }
        }
    }
```

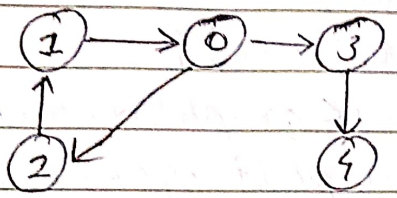
DATE

```
for(int i=0; i<n; i++){
    for( int j=0; j<n; j++ ){
        if( matrix[i][j] == Integer.MAX_VALUE){
            matrix[i][j] = -1;
        }
    }
}
return matrix;
}
```

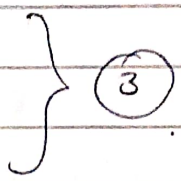
Q Strongly connected component : (SCC) (Kosaraju).

- SCC is a component in which we can reach every vertex of the component from every other vertex in that component.

ex: 1)

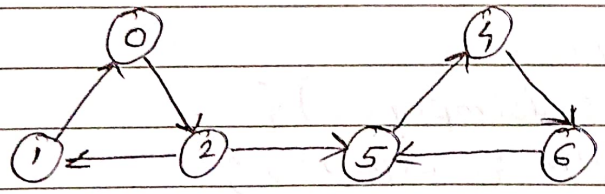


o/p: 0 1 2 3 4



- which vertex is maximum reach other vertex but maximum number of SCC

ex: 2)



o/p: 0 1 2 4 5 6

SCC = 2

approch : Use Kosaraju Algorithm :

1) get nodes in stack (topological sort).
2) Transpose the graph.
3) do the DFS according to stack nodes on the transpose graph

DATE

// code //
```
public static void KosarajuAlg(graph, int v){
    // step 1 //
    boolean visit[] = new boolean[v];
    Stack<Integer> s = new Stack<>();
    for(int i=0; i<v; i++){
        if(!visit[i]){
            topSort(graph, i, visit, s);
        }
    }
    // step 2 //
    ArrayList<Edge> transpose[] = new ArrayList[v];
    for(int i=0; i<graph.length; i++){
        transpose[i] = new ArrayList();
        visit[i] = false;
    }
    for(int i=0; i<v; i++){
        for(int j=0; j<graph[i].length; i++){
            Edge e = graph[i].get(j);
            transpose[e.dest].add(new Edge(e.dest,
                                           e.src));
        }
    }
    // step 3 //
    while(!s.isEmpty()){
        int curr = s.pop();
        if(!visit[curr]){
            System.out.print("SCC: ");
            dfs(transpose, curr, visit);
            System.out.println();
        }
    }
}
```

(topSort util function.) →

initialise →

function to print element →

DATE

Q. Bridge in graph : (Tarjan Algorithm)

- Bridge is an edge whose deletion increase the graph number of connected components.

```
//code
public static void dfs(graph, int cur, int par, int dt[],
                       int low[], boolean visit[], int time)

    vis[cur] = true;
    dt[cur] = low[cur] = ++time;

    for(int i=0; i<graph[cur].size(); i++) {
        Edge e = graph[cur].get(i);
        int nei = e.dest;
        if(nei == par){
            continue;
        } else if(!visit[nei]) {
            dfs(graph, nei, cur, dt, low, visit, time);
            low[cur] = Math.min(low[cur], low[nei]);
            if(dt[cur] < low[nei]) {
                System.out.println("Bridge : "+cur+
                                   " ---- " + nei);
            }
        } else {
            low[cur] = Math.min(low[cur], dt[nei]);
        }
    }
}

public static void tarjanBridge(graph, int v) {
    int dt[] = new int[v]; int low[] = new int[v];
    int time = 0; boolean visit[] = new boolean[v];
    for(int i=0; i<v; i++){
        if(!visit[i]) {
            dfs(graph, i, -1, dt, low, visit, time);
        }
    }
}
```

DATE 04/10/24

9. Articulation Point : (Tarjan's Algorithm)

//code//

```
public static void dfs(graph, int curr, int par, int dt[],
        int low[], int time, boolean visit[], boolean ap[]){

    visit[curr] = true;
    dt[curr] = low[curr] = ++time;
    int children = 0;                    = 1

    for(int i=0; i< graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        int nei = e.dest;
        if(par == nei){
            continue;
        }else if(visit[nei]){
            low[curr] = Math.min(low[curr], dt[nei]);
        }else{
            dfs(graph, nei, curr, dt, low, time, visit, ap);
            low[curr] = Math.min(low[curr], low[nei]);
            if(par != -1 && dt[curr] <= low[nei]){
                ap[curr] = true;
            }
            children++;
        }
    }
    if(par == -1 && children > 1){
        ap[curr] = true;
    }
}
```

DATE

```
public static void getAp (graph, int v){
    int dt[] = new int[v];
    int low [] = new int[v];
    int time =0;
    boolean visit[] = new boolean [v];
    boolean ap[] = new boolean [v];

    for( int i=0; i<v; i++ ){
        if ( !visit[i] ){
            dfs (graph, i, -1, dt, low, time, visit, ap);
        }
    }
    // print all APs. //
    for( int i=0; i<v; i++ ){
        if( ap[i] ){
            System. out. println ("AP : " + i);
        }
    }
}
```

Q Preq Pre-requisite Tasks

ex ⇒ graph = [[1,0], [2,1], [3,2]].

∴ 1 ← 0
2 ← 1
3 ← 2

} 0 → 1 → 2 → 3.

- approach :
  - create adjecency list (graph).
  - apply topological sort by kann's algorithm
  - ~~to~~ add count vertex, if count is equal to no. of task then return true otherwise false.

//code//
```
public static boolean isPossible(int N, int P,
                                 int[][] prerequisites){
    // N- no. of tasks, P- pairs //
    // create adjecency list //
    List<Integer> graph[] = new ArrayList[N];
    for(int i=0; i<N; i++){
        graph[i] = new Arraylist();
    }
    int indeg[] = new int[N];
    for(int i=0; i<P; i++){
        graph[prerequisites[i][1]].add(prerequisites[i][0]);
        indeg[prequisites[i][0]]++;
    }
    // kann's algorithm //
    queue<Integer> q = new LinkedList();
    for(int i=0; i<N; i++){
        if(indeg[i]==0){
            q.add(i);
        }
    }
    int count=0;
```

DATE

```
while( !q.isEmpty()){
    int curr = q.remove();
    count++;
    for( int i=0; i<graph[curr].size(); i++){
        int ele = graph[curr].get(ele i);
        indeg[ele] - -;
        if(indeg[ele]==0){
            q.add(ele);
        }
    }
}
return count==N;
}
```

Q. ~~find~~ Alien Dictionary : // * dry run *//

- find the order of ~~char~~ character given String.

ex: human dictory order

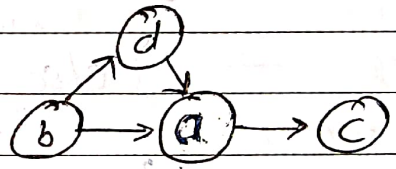
- a, b, c, d ....

ex: dict[] = {"baa", "abcd", "abca", "cab", "cad"};

"abca" "cad"

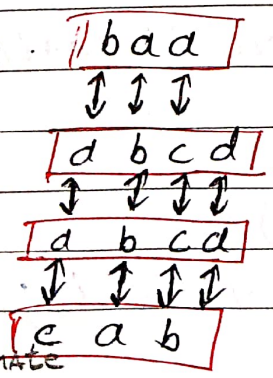
∴ baa
abcd
abca
cab
cad

b → a
d → a
a → c
b → d



order ⇒ b d a c

compare




```
for(int i=0; i< dict.length; i++){
    String st1 = dict[i];
    String st2 = dict[i+1];
    int j=0; k=0;
    while ( j<st1.len && k<st2.len
        st1==st2 PACE
        st1[j] == st2[k]) j++, k++
```

// code //

```
public String findOrder(String[] dict, int n, int k){
                                  ↑ words   ↑ alphabet
    // create adjecency list //
    Arraylist<Integer> adj[] = new Arraylist[k];
    for(int i=0; i<k; i++){
        adj[i] = new Arraylist();
    }
    int indeg[] = new int[k];
    // stored "a" = 0, 'b' = 1, 'c' = 2, ---- so on //

    for(int i=0; i<n-1; i++){
        String st1 = dict[i];
        String st2 = dict[i+1];
        int p=0, q=0;
        while(p<st1.length() && q<st2.length() &&
                st1[p] == st2[q]){
            p++; q++;
        }
        if(p<st1.length() && q<st2.length()){
                                                  -'a'
            adj[st1.charAt(p)].add(st2.charAt(q)-'a');
            indeg[st2.charAt(q)-'a']++;
        } else if(p<st1.length()){
            return "";
        }
    }
    // Kann's algorithm //
    Queue<Integer> q = new LinkedList<>();
    for(int ele : indeg){
        if(ele==0) q.add
    for(int i=0; i<k; i++){
        if(indeg[i]==0){ q.add(i);
    }
}
```

meaning. ⟶ k >= st2.length.

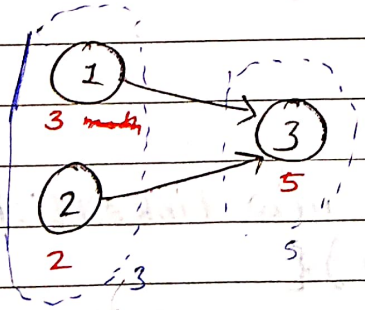
ex: abcd
    abc

```
StringBuilder st = new StringBuilder("");
while( !q.isEmpty()){
    int curr= q.remove();
    char ch=(char) ('a'+curr);
    st.append(st);

    for( int i=0; i< adj[curr].size() ; i++) {
        int ele = adj[curr].get(i);
        indeg[ele]--;
        if( indeg[ele]==0){
            q.add(ele);
        }
    }
}
if ( st.length() < k){          → cycle cond^n.
    return "";
}
return st.toString();
}
```

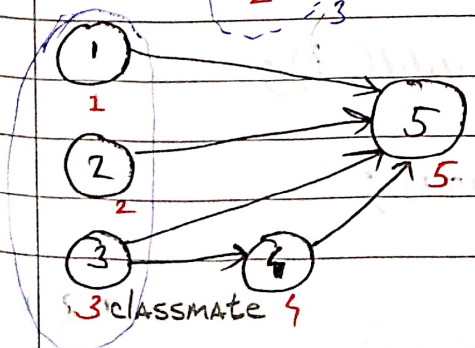
Q. Parallel course III : (Leetcode easy ~~Hard~~)

ex:

1)



→ ans = 8



→ 3+4+5 ⇒ ans = 12

cond^n:

approach: create adj. list & apply topological sort (kann's algo).

DATE 13/10/2024

// code //

```
class Edge {
    int src, dest, swt;
    Edge (int s, int d, int w){
        src = s; dest = d; swt = w;
    }
}
public int minimumTime (int n, int [][] relations, int[] time
    // create adjency list //
    int newTime [] = new int [n+1];
    for( int i=0; i<n; i++) {
        newTime [i+1] = time[i];
    }
    ArrayList <Edge> graph [] = new Arraylist<>();
    for( int i=0; i<=n; i++) {
        graph [i] = new Arraylist();
    }
    int indeg [] = new int [n+1];
    for( int i=0; i< relations.length; i++) {
        int src = relations [i][0];
        int dest = relations [i][1];
        int swt = newTime [src];
        graph[src].add ( new Edge(src, dest, swt));
        indeg[dest]++;
    }
    // kann's algorithm //
    Queue <Integer> q = new linkedlist<>();
    for( int i=1; i<=n; i++){
        if( indeg [i] == 0) q.add( i);
    }
    int period [] = new int[n+1];
```

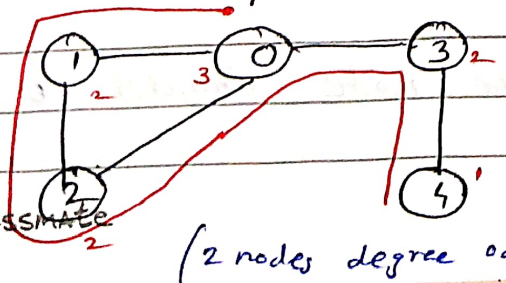
DATE

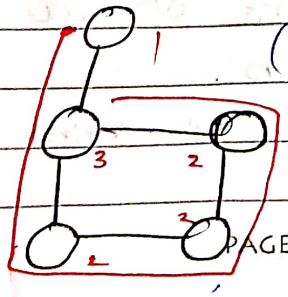
```
while(!q.isEmply()){
    int curr = q.remove();
    for(int i=0; graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        int src = e.src;
        int dest = e.dest;
        int swt = e.swt;
        period[dest] = Math.max(period[src]+swt,
                                period[dest]);
        indeg[dest]--;
        if(indeg[dest]==0){
            q.add(dest);
        }
    }
}
int max=0;
for(int i=1; i< period.length; i++){
    period[i] += newTime[i];
    if(max < period[i]){
        max = period[i];
    }
}
return max;
}
```

14-10-2024.

Q. Euler Path & Euler Circuit

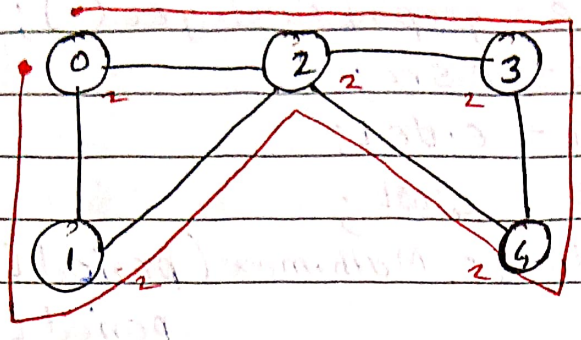
- It is the path in a graph that visit every edge exactly once





DATE

but in euler circuit starting & ending point same & vertex does not visit more than one time.

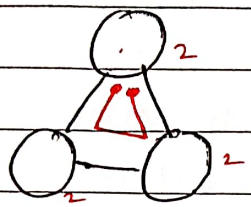


Starting pt = ending pt.

all degrees are even

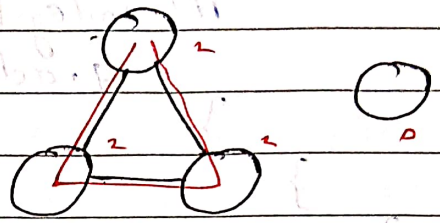
Some example :

1]



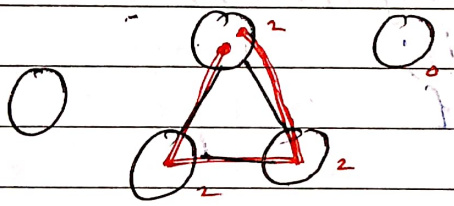
EP ✗
EC ✓

2]



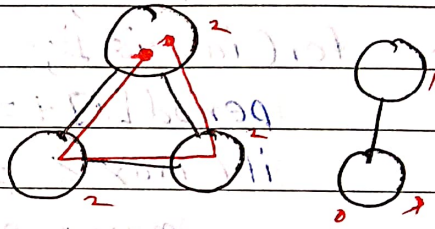
EP ✓
EC ✓

3]



EP ✓
EC ✓

4]



EP ✗
EC ✓

Euler path : all edges are visited.

Euler ckt : Starting pt == ending point. & all edges are visited.

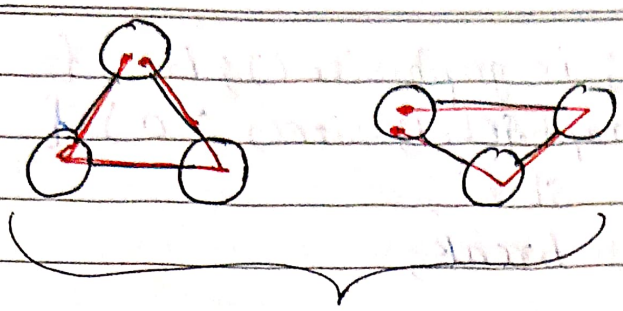
Condⁿ :

- degree of all nodes / vertex must be even.
- all non-zero degree node should be connected

// dry run remaining //

DATE

ex:



one graph.

⇒ not connected

∴ EC - ✗

∴ This is not EC.

approach : (DFS).(EC).

1) ~~the~~ find degree of every node
2) if degree of any node is odd return false.
3) hence also check non visited vertex (connected graph) degree is zero.

approach for Euler path:

- Same as EC approach but extra condn is 0 or 2 nodes have odd degree.

// code of Euler ckt //

```
private ~~boolean~~ void DFS (int curr, graph , boolean visit[]){
        visit[curr] = true;
        ~~for( int i=0; i< graph.get(curr).size();~~
        for( int ele : graph.get(curr) ){
            if( ! visit[~~curr~~ ele] ) {
                DFS ( ele, graph , visit);
            }
        }
}
public ~~static~~ ~~void~~ boolean isEulerCkt ( int v, graph ){
        boolean visit[] = new boolean[v];
        int start = -1;
```

this loop finding the vertex whose degree is greater than 0 for start traverse.

```
for(int i=0; i< graph.size(); i++){
    if(graph.get(i).size() > 0){
        st = i;
        break;
    }
}
if( start == -1){
    return true;
}

DFS (start, graph, visit);

for(int i=0; i< v; i++){
    if(!visit[i] && graph.get(i).size() >0){
        return false;
    }
}

~~int odd~~ // odd degree //
for(int i=0; i< graph.size(); i++){
    if( graph.get(i).size() % 2 != 0){
        return false;
    }
}
return true;
}
```

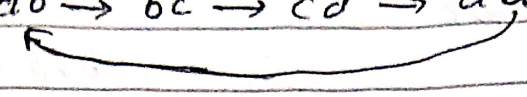
not equal to.

DATE 15/10/2024

Q. Circle of String : (easy).

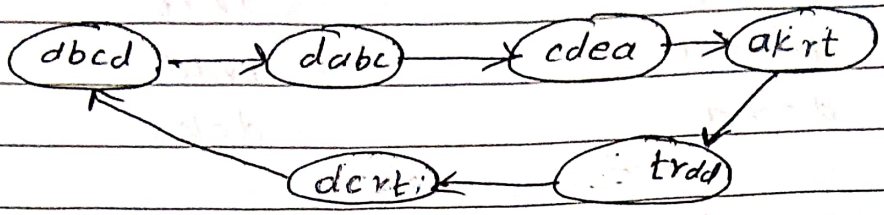
ex: 1) "ab", "bc", "cd", "da".

→ ab → bc → cd → da



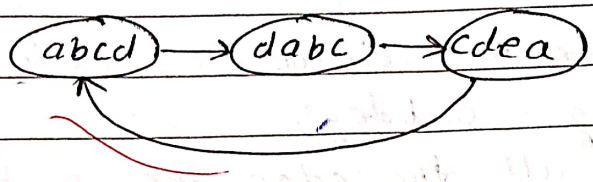
2) "abcd", "dabc", "cdea", "akrt", "trdd", "dcka".

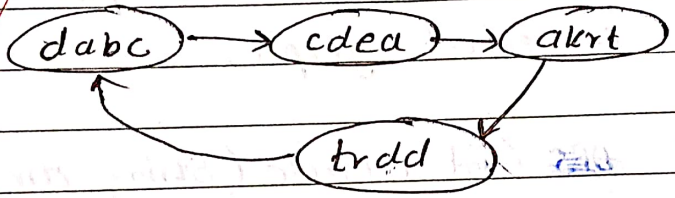
⇒



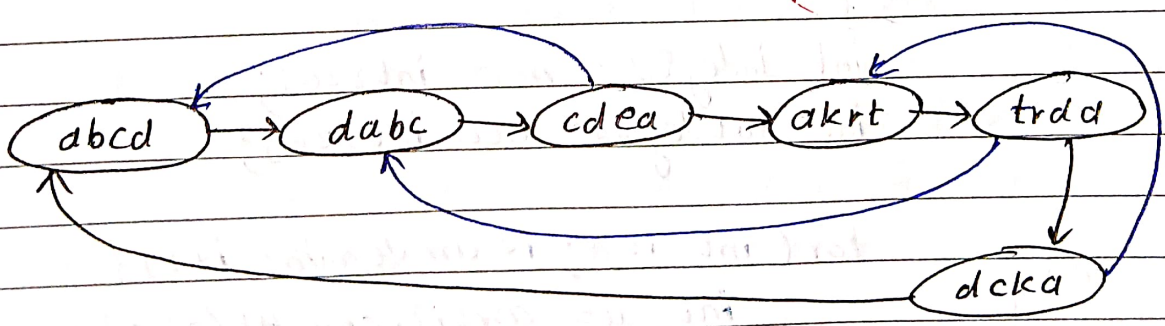
correct.

but. cycle also create by ~~given~~ like given.





hence,
when circle present in string means all string are visited & first & last string are connected each other, called as Hemiltonin cycle.



Thats, the permutation generate because every time check all vertex. Time complexity O(N!).

DATE

approach :

1) In string important character is first & last, hence create edge bet" them.
i.e "~~ad~~", "~~dc~~"

a → d	..... a bad
d → c	---- dabc
c → a	--- cdea
a → t	..... akrt
t → d	-- trdd
d → a	----- dcka

∴ a → d → c → a → t → d → a

2) Check ~~gfg~~ created circuit is euler or not.
↳ Indeg = outdeg
↳ DFS : all the edge are part of single connected graph.

```
// code // public void ~~DFS~~ (~~int~~ IsCircle (String arr[]) {
    // create adjecency list //
    ArrayList<Integer> adj[] = new ArrayList[26];
    for( int i=0; i<26; i++ ){
        adj[i] = new ArrayList();
    }
    int indeg[] = new int[26];
    int outdeg[] = new int[26];

    for( int i=0; i< arr.length; i++) {
convert char into integer {
        int u = arr[i].charAt(0) - 'a';
        int v = arr[i].charAt( arr[i].length()-1 ) - 'a';
```

```
        adj[u].add(v);
        indeg[v]++;
        outdeg[u]++;
    }

    // check indeg == outdeg. //
    for(int i=0; i<26; i++){
        if(indeg[i] != outdeg[i]){
            return 0;
        }
    }
    // Euler circuit //
    DFS(arr[0].charAt(0)-'a', adj, visit);

    for(int i=0; i<26; i++){
        if(indeg[i]!=0 && !visit[i]){
            return 0;
        }
    }
    return 1;
}
```

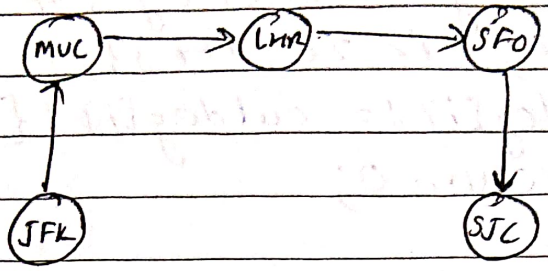
Q. Reconstruct ~~Itte~~ Itinerary :

- given the tickets like [from, toplace] of, reconstruct the itinerary in order and return it
- All the tickets belong ~~the~~ to man who departs from JFK, thus the itinery must begin with JFK. If there are multiple valid ~~ite~~ itineraries you should return the itinery that has the smallest lexical order when read as a single string.

DATE

ex 1) tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]].

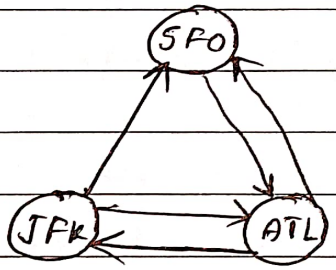
create graph:



return the path in the form of path.

itinery : ["JFK", "MUC", "LHR", "SFO", "SJC"].

ex 2) tickets = [["JFK", "SFO"], ["JFK", ATL"], ["SFO", "ATL"], ["ATL", "JFK"], ["ATL", "SFO"]].



a) itinery : ["JKF", "SFO", "ATL", "JFK", "ATL", "SFO"]

b) itinery : ["JKF", "ATL", "JFK", "SFO", "ATL", "SFO"].

but (b) ans is larger in lexical order. (ATL > SFO)

q.poll() ↔ q.remove()

DATE

// code //

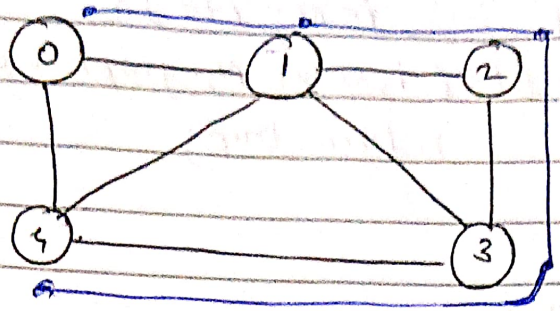
```
private void DFS(String curr, Map<String, PriorityQueue<String
        >> graph, List<String> itinerary) {

    while (map.containsKey(curr) && !map.get(curr).

private static void DFS(String s, Map<String, <PriorityQueue<String>> map,
                        List<String> list) {

    while(map.containkey(s) && !map.get(s).isEmpty()) {
        String d = map.get(s).poll();
        DFS(d, map, list);
    }
    list.add(0,s);
}

public List<String> findItinerary(List<List<String>> tickets) {

    Map<String, PriorityQueue<String>> map = new HashMap();

    for( List<String> lk : tickets ) {
        String s = lk.get(0);
        String d = lk.get(1);
        PriorityQueue<String> pq = map.getOrDefault(s, new PriorityQueue<>
                                                    ());
        pq.add(d);
        map.put(s, pq);
    }

    List<String> list = new ArrayList<>();
    DFS("JFK", map, list);
    return list;
}
```

DATE 31/10/2024

Q Hamiltonian Path :

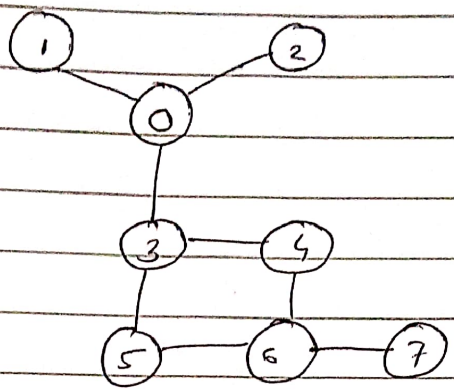
- Visit all the vertex exactly ones once time and does not visit again.

ex : 1)



- Hamiltonian Path
  0 - 1 - 2 - 3 - 4
- Hamiltonian circuit
  0 - 1 - 2 - 3 - 4 - 0

ex: 2)



It does not hamiltonian path

//code//

```
boolean DFS(int cur, int count, int N, graph, visit){
    visit[cur] = true;
    count++;
    if(count == N) return true;
    for(int i=0; i< graph[cur].size(); i++){
        int ele = graph[cur].get(i);
        if(!visit[ele] && DFS(ele, count, N, graph, visit)){
            return true;
        }
    }
    visit[cur] = false;
    count--;
    return false;
}
```

DATE

```
~~publi~~ boolean hamiltonian (int N, ArrayList<Integer> graph[]){
        boolean visit[] = new boolean[N];
        int count = 0;
        for(int i=0; i<N; i++){
            if(DFS(i, count, N, graph, visit)){
                return true;
            }
        }
        return false;
}
```