//Binary Indexed Tree

```
#define MAX 1010
//normally 1 based index
int tree[MAX];
int MaxVal;//always should be set(size of the set len)

//cumulitive sum
int query(int idx) {
    if(idx<=0) return 0;
    int sum = 0;
    idx =min(idx,MaxVal);
    while (idx > 0) {
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}

void update(int idx,int val) {
    if(idx<=0) return;
    while (idx <= MaxVal) {
        tree[idx] += val;
        idx += (idx & -idx);
    }
}

//single indexed value
int querySingle(int idx) {
    int sum = tree[idx];
    // sum will be decreased
    if (idx > 0) {
        // special case
        int z = idx - (idx & -idx);
        // make z first
        idx--;
        // idx is no important any more, so instead y,
you can use idx
        while (idx != z) {
            // at some iteration idx (y) will become z
            sum -= tree[idx];
            // substruct tree frequency which is between
y and "the same path"
            idx -= (idx & -idx);
        }
    }
    return sum;
}

void scale(int c) {
    for (int i = 1 ; i <= MaxVal ; i++)
        tree[i] = tree[i] / c;
}

// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// some of them (we do not know which one)

// bitMask - initially, it is the greatest bit of MaxVal
// bitMask store interval which should be searched
int find(int cumFre) {
    int idx = 0; // this var is result of function

    while ((bitMask != 0) && (idx < MaxVal)) {
        int tIdx = idx + bitMask;
        // we make midpoint of interval
        if (cumFre == tree[tIdx])
            // if it is equal, we just return idx
            return tIdx;
        else if (cumFre > tree[tIdx]) {
            // if tree frequency "can fit" into cumFre,
            // then include it
            idx = tIdx; // update index
            cumFre -= tree[tIdx];
// set frequency for next loop
        }
        bitMask >>= 1;
// half current interval
    }
    if (cumFre != 0)
// maybe given cumulative frequency doesn't exist
        return -1;
    else
        return idx;
}

// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// the greatest one
int findG(int cumFre) {
    int idx = 0;

    while ((bitMask != 0) && (idx < MaxVal)) {
        int tIdx = idx + bitMask;
        if (cumFre >= tree[tIdx]) {
// if current cumulative frequency is equal to cumFre,
// we are still looking for higher index (if exists)
            idx = tIdx;
            cumFre -= tree[tIdx];
        }
        bitMask >>= 1;
    }
    if (cumFre != 0)
        return -1;
    else
        return idx;
}


//for 2D BIT
int query(int x, int y) {
    int y1;
    int sum=0;
    while (x > 0) {
        y1 = y;
        while (y1 > 0) {
            sum += tree[x][y1];
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }
    return sum;
}

void update(int x, int y, int val) {
    int y1;
    while (x <= max_x) {
        y1 = y;
        while (y1 <= max_y) {
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}
//2D BIT Ended
```

//Articulation Point

```
// 1 based, nodes starts from 1
#define MAXN 10005
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];
//tin is here the array for discovery time, and fup for
the discovery time lowest

set<int> result_of_articaultion_points;
void IS_CUTPOINT(int u) {
    result_of_articaultion_points.insert(u);
    return;
}

void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    int children = 0;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (to == p) continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] >= tin[v] && p != -1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if (p == -1 && children > 1)
        IS_CUTPOINT(v);
}
int main() {
    int n=II,m=II;
//n=number of vertexes and m= number of the edges.
    for(int i=1; i<=m; i++) {
        int u=II, v=II;
        g[u].pb(v);
        g[v].pb(u);
    }
    timer = 0;
    for (int i=1; i<=n; ++i)
        used[i] = false;
    dfs (1);
    cout<<"Articulation_points:\n";
    for(set<int>::iterator
it=result_of_articaultion_points.begin();
it!=result_of_articaultion_points.end(); it++) {
        cout<<*it<<endl;
    }
    return 0;
}
```

//Bridge

```
//this is a 1 based code
const int MAXN = 10005;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];
set<pi> res_bridge;
void IS_BRIDGE(int v, int to) {
    res_bridge.insert(pi(v,to));
    return;
}
void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (to == p) continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] > tin[v])
                IS_BRIDGE(v,to);
        }
    }
}
void find_bridges(int n) {
    timer = 0;
    for (int i=1; i<=n; ++i)
        used[i] = false;
    for (int i=1; i<=n; ++i)
        if (!used[i])
            dfs (i);
}

int main() {
    int n=II,m=II;
//n=number of vertexes and m= number of the edges.
    for(int i=1; i<=m; i++) {
        int u=II, v=II;
        g[u].pb(v);
        g[v].pb(u);
    }
    timer = 0;
    find_bridges(n);
    cout<<"Bridges:\n";
    for(set<pi>::iterator it=res_bridge.begin();
it!=res_bridge.end(); it++) {
        cout<<it->xx <<" "<<it->yy <<endl;
    }
    return 0;
}
```

//Bellman-Ford Algorithm

```
//complexity VE
#define SIZE 1010
//#define INF 2000000000
vi adj[SIZE],cost[SIZE];

//0 based
bool BellmanFord(int source,int nodes) {
//returns true if it has negative cycle
    vector<ll>dist;
    int i,j,k,w,v;

    for(i=0; i<=nodes; i++) { //distance from source
        dist.push_back(INF);
    }
    dist[source]=0;

    for(i=1; i<=nodes-1; i++) {
//this iterator for the update
        for(j=1; j<=nodes; j++)
//current and the next iterator is for edges to go
            for(k=0; k<adj[j].size(); k++) {
                v=adj[j][k];
                w=cost[j][k];
                if(dist[j]!=INF)
                    dist[v]=min(dist[v],dist[j]+w);
            }
    }

    for(i=1; i<=nodes; i++)
        for(j=0; j<adj[i].size(); j++) {
            v=adj[i][j];
            w=cost[i][j];
```

```
            if(dist[v]>dist[i]+w)
                return true;
        }
    return false;
}
/// ANOTHER WAY OF BELLMAN-FORD. BECAREFULL DEFINING THE
INFINITY.
//complexity VE
#define SIZE 1010
//#define INF 2000000000
vi adjList[SIZE],RadjList[SIZE];
vector<ll>dist(SIZE);
bool vis[SIZE];
vi res;
//0 based
struct graph {
    int u, v, w;
    graph( int x, int y, int z) {
        u=x, v=y, w=z;
    }
};
vector<graph > gr;
void dfs(int u) {
    vis[u]=1;
    res.pb(u);
    for(int i=0; i<adjList[u].size(); i++) {
        int v=adjList[u][i];
        if(!vis[v])
            dfs(v);
    }
}
bool BellmanFord(int nodes) {
//returns true if it has negative cycle
    ll i,j,k,w,v,u;
    for(i=0; i<nodes+2; i++)
        dist[i]=INF;
    for(i=1; i<nodes; i++) {
//number of iteration to do. For bellman ford algorithm
it is 1 to n-1
        for(j=0; j<gr.size(); j++) {
//current and next loop the edges to visit
            u=gr[j].u;
            v=gr[j].v;
            w=gr[j].w;
            dist[v]=min(dist[v],dist[u]+w);
        }
    }

    bool foo=0;
    for(j=0; j<gr.size(); j++) {
        u=gr[j].u;
        v=gr[j].v;
        w=gr[j].w;
        if(dist[v]>dist[u]+w) {
            foo=1;
            dist[v]=dist[u]+w;
            if(!vis[u])
                dfs(u);
        }
    }
    return foo;
}
```

//Finding Eular Path in O(Vertex)

```
int n;
vector < vector<int> > g (n, vector<int> (n));
///Here to Read the graph
vector<int> deg (n);
for (int i=0; i<n; ++i)
    for (int j=0; j<n; ++j)
        deg[i] += g[i][j];
int first = 0;
while (!deg[first])
    ++first;

int v1 = -1,  v2 = -1;
bool bad = false;
for (int i=0; i<n; ++i)
    if (deg[i] & 1)
        if (v1 == -1)
            v1 = i;
        else if (v2 == -1)
            v2 = i;
        else
            bad = true;

if (v1 != -1)
    ++g[v1][v2],  ++g[v2][v1];

stack<int> st;
st.push (first);
vector<int> res;
while (!st.empty()) {
    int v = st.top();
    int i;
    for (i=0; i<n; ++i)
        if (g[v][i])
            break;
    if (i == n) {
        res.push_back (v);
        st.pop();
    } else {
        --g[v][i];
        --g[i][v];
        st.push (i);
    }
}

if (v1 != -1)
    for (size_t i=0; i+1<res.size(); ++i)
        if (res[i] == v1 && res[i+1] == v2 || res[i] ==
v2 && res[i+1] == v1) {
            vector<int> res2;
            for (size_t j=i+1; j<res.size(); ++j)
                res2.push_back (res[j]);
            for (size_t j=1; j<=i; ++j)
                res2.push_back (res[j]);
            res = res2;
            break;
        }

for (int i=0; i<n; ++i)
    for (int j=0; j<n; ++j)
        if (g[i][j])
            bad = true;

if (bad)
    puts ("-1");
else
    for (size_t i=0; i<res.size(); ++i)
        printf ("%d ", res[i]+1);
```

//Floyd Worshell

```
int V, E, u, v, w, AdjMatrix[200][200];
/*
// Graph in Figure 4.30
5 9
0 1 2
0 2 1
0 4 3
1 3 4
2 1 1
2 4 1
```

```
3 0 1
3 2 3
3 4 5
*/
scanf("%d %d", &V, &E);
for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++)
        AdjMatrix[i][j] = INF;
    AdjMatrix[i][i] = 0;
}

for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjMatrix[u][v] = w; // directed graph
}

for (int k = 0; k < V; k++)
// common error: remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            AdjMatrix[i][j] = min(AdjMatrix[i][j],
AdjMatrix[i][k] + AdjMatrix[k][j]);

for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        printf("APSP(%d, %d) = %d\n", i, j,
AdjMatrix[i][j]);
```

//Dijkstra's Algorithm

```
int mx = 20005;
int costMatrix[502][502];
vi adjList[502];
bool vis[502];
int dist[502];
priority_queue<int, vector<pi>,greater<pi> >q;
void Dijkstra(int source) {
    dist[source]=0;
    q.push(pi(0,source));
    while(!q.empty()) {
        pi x=q.top();
        int u=x.yy;
        q.pop();
        if(!vis[u]) {
            for(int i=0; i<adjList[u].size(); i++) {
                int v=adjList[u][i];
                int
max_cost=max(dist[u],costMatrix[u][v]);
                if(max_cost<dist[v]) {
                    dist[v]=max_cost;
                    q.push(pi(dist[v],v));
                }
            }
            vis[u]=1;
        }
    }
}
int main() {
    int t=II;
    for(int cs=1; cs<=t; cs++) {
        for(int i=0; i<502; i++)
            adjList[i].clear();
        for(int i=0; i<502; i++)
            dist[i]=mx;
        clr(vis, 0);
        for(int i=0; i<502; i++)
            for(int j=0; j<502; j++)
                costMatrix[i][j]=mx;
        int nodes=II, edges=II;
        for(int i=1; i<=edges; i++) {
            int u=II, v=II, cost=II;
            if(costMatrix[u][v]==mx) {
                costMatrix[u][v]=cost;
                costMatrix[v][u]=cost;
                adjList[u].pb(v);
                adjList[v].pb(u);
            } else {
                int mini=min(cost, costMatrix[u][v]);
                costMatrix[u][v]=mini;
                costMatrix[v][u]=mini;
            }
        }
        int source=II;
        Dijkstra(source);
        pf("Case %d:\n",cs);
        for(int i=0; i<nodes; i++) {
            if(dist[i]==mx)
                pf("Impossible\n");
            else
                pf("%d\n",dist[i]);
        }
    }
    return 0;
}
```

//Minimum Spanning Tree

//Kruskal's Algorithm O(mlgn)

```
#define node_sz 100
#define edge_sz 1000
vector<pair<int, pair<int, int> > > AdjList; //
G(COST,(U, V))
vector<pair <int, int> >res;
int n,m,cost;
vector<int>p(node_sz);
int dsu_get(int v) {
    return (v == p[v])? v:(p[v]=dsu_get(p[v]));
}
void dsu_unite(int a, int b) {
    a = dsu_get(a);
    b = dsu_get(b);
    if (rand()&1)
        swap(a, b);
    if (a!=b)
        p[a]=b;
}

void KruskalsAlgorithm() {
    cost = 0;
    sort(all(AdjList));
    p.resize(node_sz);
    for (int i=0; i<node_sz; ++i)
        p[i]=i;
    for(int i=0; i<m; ++i) {
        int a = AdjList[i].yy.xx, b =AdjList[i].yy.yy, l
= AdjList[i].xx;
        if (dsu_get(a)!=dsu_get(b)) {
            cost += l;
            res.pb(AdjList[i].second);
            dsu_unite(a, b);
        }
    }
}

int main() {
    n=II, m=II;
    AdjList.clear();
    p.clear();
    for(int i=1; i<=m; i++) {
        int u,v,w;
        cin>>u>>v>>w;
        AdjList.pb(mp(w,mp(u, v)));
        AdjList.pb(mp(w,mp(v, u)));
    }
```

```
    KruskalsAlgorithm();
    cout<<cost<<endl;
    return 0;
}
```

//Kruskal's Algorithm O(m^2 + mlgn)

```
#define edge_sz 1000
#define node_sz 100
int m,n;
vector<pair<int,pair<int,int> > >AdjList;
// G(COST,(U,V))
vector<int>tree_id(node_sz);
int cost=0;
vector<pair<int, int> >res;
void Krushkals_Algorithm() {
    sort(all(AdjList));
    cost=0;
    for (int i=0; i<node_sz; ++i)
        tree_id[i]=i;
    for (int i=0; i<m; i++) {
        int a=AdjList[i].yy.xx, b=AdjList[i].yy.yy,
l=AdjList[i].xx;
        if (tree_id[a]!=tree_id[b]) {
            cost+=l;
            res.pb(mp(a, b));
            int old_id=tree_id[b], new_id=tree_id[a];
            for (int j=0; j <n; ++j)
                if (tree_id[j] == old_id)
                    tree_id[j] = new_id;
        }
    }
    return;
}

int main() {
    n=II, m=II;
    AdjList.clear();
    tree_id.clear();
    for(int i=1; i<=m; i++) {
        int u,v,w;
        cin>>u>>v>>w;
        AdjList.pb(mp(w,mp(u, v)));
        AdjList.pb(mp(w,mp(v, u)));
    }

    Krushkals_Algorithm();
    cout<<cost<<endl;
    for(int i=0; i<res.size(); i++)
        cout<<res[i].xx<<" "<<res[i].yy<<endl;
    return 0;
}
```

//Prim's Algorithm for Sparse Graph

```
int n;
int mst_cost;
vector< vector< pair <int, int> > > AdjList;
vector<bool> taken;
// global boolean flag to avoid cycle
priority_queue<pi> pq;
// priority queue to help choose shorter edges

void process(int vtx) {
// so, we use -ve sign to reverse the sort order
    taken[vtx] = 1;
    for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
        pi v = AdjList[vtx][j];
        if (!taken[v.xx])
            pq.push(mp(-v.yy, -v.xx));
    }
}
int main() {
    int m;
    n=II, m=II;
    AdjList.resize(n+3);
    for(int i=1; i<=m; i++) {
        int u=II, v=II, w=II;
        u--,v--; //making the zero based vertex
        AdjList[u].pb(mp(v,w));
        AdjList[v].pb(mp(u,w));
    }
    taken.assign(n+1, 0);
// no vertex is taken at the beginning
    process(0);
//take vertex 0 and process all edges incident to vertex
0
    mst_cost = 0;
    while (!pq.empty()) {
// repeat until V vertices (E=V-1 edges) are taken
        pi front = pq.top();
        pq.pop();
        int u = -front.second, w = -front.first;
// negate the id and weight again
        if (!taken[u])
// we have not connected this vertex yet
            mst_cost += w, process(u);
// take u, process all edges incident to u
    }
// each edge is in pq only once!
    printf("MST cost = %d (Prim's)\n", mst_cost);
    return 0;
}
```

//Strongly Connected Component

//Tarjan SCC

```
#define lim          1005 //number of nodes
//No problem in multiple edges and self loop
//0 based
VI adj[lim];//only adj should be cleared
int col[lim],low[lim],tim[lim],timer;
int group_id[lim],n,m,components;//components=number of
components group_id = which node belongs to which node
stack<int>S;

void scc(int u) {
    int i,v,tem;
    col[u]=1;
    low[u]=tim[u]=timer++;
    S.push(u);
    fr(i,0,SZ(adj[u])-1) {
        v=adj[u][i];
        if(col[v]==1)
            low[u]=min(low[u],tim[v]);
        else if(col[v]==0) {
            scc(v);
            low[u]=min(low[u],low[v]);
        }
    }

    //SCC checking...
    if(low[u]==tim[u]) {
        do {
            tem=S.top();
            S.pop();
            group_id[tem]=components;
            col[tem]=2; //Completed...
        } while(tem!=u);
        components++;
    }
}

int TarjanSCC(int n) { //some change may be required here
    int i;
    timer=components=0;
    mem(col,0);
```

```
    while(!S.empty()) S.pop();
    fr(i,0,n-1) if(col[i]==0) scc(i);
    return components;
}

VI nadj[lim];//new adjcency list after SCC(DAG)
void MakeNewDAG_Graph(int n) {
    int i,j,u,v;

    fr(i,0,components-1) nadj[i].clear();

    fr(i,0,n-1) {
        fr(j,0,SZ(adj[i])-1) {
            u=group_id[i];
            v=group_id[adj[i][j]];
            if(u!=v)
                nadj[u].pb(v);
        }
    }
}

void add(int ina,int inb) {
    adj[ina].pb(inb);
}

int main() {
    int i,j,t,cas=0,u,v,ans;

    while(scanf("%d %d",&n,&m)==2) {
        fr(i,0,n-1) adj[i].clear();
        fr(i,1,m) {
            scanf("%d %d",&u,&v);
            adj[u].pb(v);
        }
        TarjanSCC(n);
        printf("Total Groups: %d\n",components);
        MakeNewDAG_Graph(n);
        printf("NewGraphLinkUsingSCC: this graph is
directed acyclic graph:\n");
        //this link between groups no.....
        fr(i,0,components-1) {
            fr(j,0,SZ(nadj[i])-1) {
                u=i;
                v=nadj[i][j];
                print2(u,v);
            }
        }
    }
    return 0;
}
/*
Input:
8 14
0 1
1 2
1 5
1 4
2 6
2 3
3 2
3 7
4 5
5 6
7 6
7 3
6 5
4 0
Output:
Total Groups: 3
NewGraphLinkUsingSCC: this graph is directed acyclic
graph:
1 0
2 1
2 0
2 0


Another Input:
6 6
0 1
1 2
2 1
3 4
4 5
5 4
Total Groups: 4
NewGraphLinkUsingSCC: this graph is directed acyclic
graph:
1       0
3       2

*/
```

//Kosaraju's Algorithm for SCC by 2DFS

```
//Kosaraju's Algorithm Learned from http://emaxx.ru - Hard copy

#define sz 101
vi adjList[sz], RadjList[sz];
vector<bool> used;
vi order,component;
void dfs1(int u) {
    used[u]=1;
    for(int i=0; i<adjList[u].size(); i++) {
        int v=adjList[u][i];
        if(!used[v])
            dfs1(v);
    }
    order.pb(u);
}

void dfs2(int u) {
    used[u]=1;
    component.pb(u);
//here we can denote the graph node belongs to which
strongly connected components
    for(int i=0; i<RadjList[u].size(); i++) {
        int v=RadjList[u][i];
        if(!used[v])
            dfs2(v);
    }
}

int main() {
    int n=II,m=II; //number of nodes, edges
    for(int i=1; i<=m; i++) {
        int u=II, v=II;
        adjList[u].pb(v);
        RadjList[v].pb(u);
    }
    used.assign(n+2, false);
    for(int i=1; i<=n; i++) {
        if(!used[i])
            dfs1(i);
    }
    used.assign(n+2, false);
    reverse(all(order));
    for(int i=0; i<order.size(); i++) {
        int  v=order[i];
        if(!used[v]) {
            cout<<v<<"=";
            dfs2(v);
            for(int j=0; j<component.size(); j++)
                pf(" %d",component[j]);
```

```
            pf("\n");
            component.clear();
        }
    }
    return 0;
}
```

**Flow Related

//BPM by Ford-Fulkersion

```
// A C++ program to find maximal Bipartite matching.
#include <iostream>
#include <string.h>
using namespace std;

// M is number of applicants and N is number of jobs
#define M 6
#define N 6

// A DFS based recursive function that returns true if a
// matching for vertex u is possible
bool bpm(bool bpGraph[M][N], int u, bool seen[], int
matchR[]) {
    // Try every job one by one
    for (int v = 0; v < N; v++) {
        // If applicant u is interested in job v and v is
        // not visited
        if (bpGraph[u][v] && !seen[v]) {
            seen[v] = true; // Mark v as visited

         // If job 'v' is not assigned to an applicant OR
         // previously assigned applicant for job v
(which is matchR[v])
            // has an alternate job available.
            // Since v is marked as visited in the above
line, matchR[v]
            // in the following recursive call will not
get job 'v' again
            if (matchR[v] < 0 || bpm(bpGraph, matchR[v],
seen, matchR)) {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}

// Returns maximum number of matching from M to N
int maxBPM(bool bpGraph[M][N]) {
    // An array to keep track of the applicants assigned
to
    // jobs. The value of matchR[i] is the applicant
number
    // assigned to job i, the value -1 indicates nobody
is
    // assigned.
    int matchR[N];

    // Initially all jobs are available
    memset(matchR, -1, sizeof(matchR));

    int result = 0;
// Count of jobs assigned to applicants
    for (int u = 0; u < M; u++) {
        // Mark all jobs as not seen for next applicant.
        bool seen[N];
        memset(seen, 0, sizeof(seen));

        // Find if the applicant 'u' can get a job
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
    return result;
}

// Driver program to test above functions
int main() {
    // Let us create a bpGraph shown in the above example
    bool bpGraph[M][N] = {  {0, 1, 1, 0, 0, 0},
        {1, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0},
        {0, 0, 1, 1, 0, 0},
        {0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 1}
    };

    cout << "Maximum number of applicants that can get
job is "
         << maxBPM(bpGraph);
    return 0;
}
```

//BPM by Blossom Algo, BPM by HopCraft

See in CookBook Palindrome, SUST

//Max Flow Min Cut

//Edmond's Karp

```
//This Algorithm takes O(VE^2) (actually O(V^3E)
#define MAX_V 40

int res[MAX_V][MAX_V], mf, f, s, t;
// global variables
vi p;
vector<vi> AdjList;

void augment(int v, int minEdge) {
// traverse BFS spanning tree from s to t
    if (v == s) {
        f = minEdge;
// record minEdge in a global variable f
        return;
    } else if (p[v] != -1) {
        augment(p[v], min(minEdge, res[p[v]][v]));
// recursive
        res[p[v]][v] -= f;
        res[v][p[v]] += f;
    }       // update
}

int main() {
    int V, k, vertex, weight;
    /*
    // Graph in Figure 4.24
    4 0 1
    2 2 70 3 30
    2 2 25 3 70
    3 0 70 3 5 1 25
    3 0 30 2 5 1 70

    // Graph in Figure 4.25
    4 0 3
    2 1 100 3 100
    2 2 1 3 100
    1 3 100
    0

    // Graph in Figure 4.26.A
    5 1 0
    0
    2 2 100 3 50
    3 3 50 4 50 0 50
    1 4 100
    1 0 125
```

```
    // Graph in Figure 4.26.B
    5 1 0
    0
    2 2 100 3 50
    3 3 50 4 50 0 50
    1 4 100
    1 0 75

    // Graph in Figure 4.26.C
    5 1 0
    0
    2 2 100 3 50
    2 4 5 0 5
    1 4 100
    1 0 125
    */
    scanf("%d %d %d", &V, &s, &t);

    memset(res, 0, sizeof res);
    AdjList.assign(V, vi());
    for (int i = 0; i < V; i++) {
        scanf("%d", &k);
        for (int j = 0; j < k; j++) {
            scanf("%d %d", &vertex, &weight);
            res[i][vertex] = weight;
            AdjList[i].push_back(vertex);
        }
    }

    mf = 0;
    while (1) {
// now a true O(VE^2) Edmonds Karp's algorithm
        f = 0;
        bitset<MAX_V> vis;
        vis[s] = true;
// we change vi dist to bitset!
        queue<int> q;
        q.push(s);
        p.assign(MAX_V, -1);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            if (u == t) break;
            for (int j = 0; j < (int)AdjList[u].size();
j++) {  // we use AdjList here!
                int v = AdjList[u][j];
                if (res[u][v] > 0 && !vis[v])
                    vis[v] = true, q.push(v), p[v] = u;
            }
        }
        augment(t, INF);
        if (f == 0) break;
        mf += f;
    }

    printf("%d\n", mf);
// this is the max flow value

    return 0;
}
```

//Dinic's Max Flow

See at COOKBOOK Palindrome, SUST

//Longest Common Subsequence

```
string a,b,ans_str;
ll len1,len2;
bool visit[mod][mod];
ll dp[mod][mod];
ll calLCS(ll i, ll j) {
    if(a[i]=='\0'||b[j]=='\0')
        return 0;
    if(visit[i][j]==true)
        return dp[i][j];
    ll ans=0;
    if(a[i]==b[j])
        ans = 1 + calLCS(i+1,j+1);
    else {
        ll v1 = calLCS(i+1,j);
        ll v2 = calLCS(i,j+1);
        ans =max (v1,v2);
    }
    dp[i][j]=ans;
    visit[i][j]=true;
    return dp[i][j];
}
void printLCS(ll i, ll j) {
    if(a[i]=='\0'||b[j]=='\0') {
        cout<<ans_str<<endl;
        return;
    }
    if(a[i]==b[j]) {
        ans_str+=a[i];
        printLCS(i+1,j+1);
        ans_str.erase(ans_str.end()-1);
    } else {
        if(dp[i+1][j]>dp[i][j+1])
            printLCS(i+1,j);
        else if(dp[i+1][j]<dp[i][j+1])
            printLCS(i,j+1);
        else {
            printLCS(i,j+1);
            printLCS(i+1,j);
        }
    }
}
int main() {
    cin>>a>>b;
    len1 = a.length();
    len2 = b.length();
    cout<<calLCS(0,0)<<"\n";
    printLCS(0,0);
    return 0;
}
```

//Longest Increasing Subsequence

//n^2

```
int n; // the number of items in the sequence
int Sequence[32]; // the sequence of integers
int L[32]; // L[] as described in the algorithm

void takeInput() {
    scanf("%d", &n); // how many numbers in the sequence
?

    // take the sequence
    for( int i = 0; i < n; i++ )
        scanf("%d", &Sequence[i]);
}

int LIS() {
// which runs the LIS algorithm and returns the result
    int i, j; // auxilary variables for iteration

    // initialize L[] with 1
    for( i = 0; i < n; i++ )
        L[i] = 1;

    // start from the left most item and itetare right
    for( i = 0; i < n; i++ ) {
// for the ith item item find all items that are in right
        for( j = i + 1; j < n; j++ ) {

            if( Sequence[j] > Sequence[i] ) {
// the item is greater than the ith item
// so, L[j] = L[i] + 1, since jth item can be added after
ith
// item. if L[j] is already greater than or equal to L[i]
+ 1
                // then ignore it
                if( L[j] < L[i] + 1 )
                    L[j] = L[i] + 1;
            }
        }
    }
    // now find the item whose L[] value is maximum
    int maxLength = 0;
    for( i = 0; i < n; i++ ) {
        if( maxLength < L[i]  )
            maxLength = L[i];
    }
    // return the result
    return maxLength;
}
int LisSequence[32]; // for storing the sequence

void findSequence( int maxLength ) {
// finds a valid sequence
    int i, j; // variable used for iteration

    // at first find the position of the item whose L[]
is maximum
    i = 0;
    for( j = 1; j < n; j++ ) {
        if( L[j] > L[i] )
            i = j;
    }

    // initialize the position in LisSequence where the
items can be added.
    // observe that the data are saving from right to
left!
    int top = L[i] - 1;

    // insert the item in i-th position to LisSequence
    LisSequence[top] = Sequence[i];
    top--;
// is decreasing such that a new item can be added in a
new place

    // now find the other valid numbers to form the
sequence
    for( j = i - 1; j >= 0; j-- ) {
        if( Sequence[j] < Sequence[i] && L[j] == L[i] - 1
) {
        // we have found a valid item so, we will save it
            i = j; // as in our algorithm
            LisSequence[top] = Sequence[i]; // stored
            top--; // decreased for new items
        }
    }

 // so, we have got the sequence, now we want to print it
    printf("LIS is");
    for( i = 0; i < maxLength; i++ ) {
        printf(" %d", LisSequence[i]);
    }
```

```
    puts("");
}
int main() {
    takeInput();
    int result = LIS();
    printf("The LIS length is %d\n", result);
    findSequence( result );
    return 0;
}
```

//nLgK

```
const int inf = 2000000000; // a large value as infinity
int n; // the number of items in the sequence
int Sequence[32]; // the sequence of integers
int L[32]; // L[] as described in the algorithm
int I[32]; // I[] as described in the algorithm

void takeInput() {
    scanf("%d", &n); // how many numbers in the sequence
?
    for( int i = 0; i < n; i++ ) // take the sequence
        scanf("%d", &Sequence[i]);
}

int LisNlogK() { // which runs the NlogK LIS algorithm
    int i; // auxilary variable for iteration

    I[0] = -inf; // I[0] = -infinite
    for( i = 1; i <= n; i++ ) // observe that i <= n are
given
        I[i] = inf; // I[1 to n] = infinite

    int LisLength = 0; // keeps the maximum position
where a data is inserted

    for( i = 0; i < n; i++ ) { // iterate left to right
        int low, high, mid; // variables to perform
binary search
        low = 0; // minimum position where we to put data
in I[]
        high = LisLength; // the maximum position

        while( low <= high ) {
// binary search to find the correct position
            mid = ( low + high ) / 2;
            if( I[mid] < Sequence[i] )
                low = mid + 1;
            else
                high = mid - 1;
        }
        // observe the binary search carefully, when the
binary search ends
        // low > high and we put our item in I[low]
        I[low] = Sequence[i];
        L[i]=low;
        if( LisLength < low )
// LisLength contains the maximum position
            LisLength = low;
    }
    return LisLength; // return the result
}

int LisSequence[32]; // for storing the sequence
void findSequence( int maxLength ) {
// finds a valid sequence
    int i, j; // variable used for iteration
    // at first find the position of the item whose L[]
is maximum
    i = 0;
    for( j = 1; j < n; j++ ) {
        if( L[j] > L[i] )
            i = j;
    }

    // initialize the position in LisSequence where the
items can be added.
    // observe that the data are saving from right to
left!
    int top = L[i] - 1;
    // insert the item in i-th position to LisSequence
    LisSequence[top] = Sequence[i];
    top--; // is decreasing such that a new item can be
added in a new place

    // now find the other valid numbers to form the
sequence
    for( j = i - 1; j >= 0; j-- ) {
        if( Sequence[j] < Sequence[i] && L[j] == L[i] - 1
) {
        // we have found a valid item so, we will save it
            i = j; // as in our algorithm
            LisSequence[top] = Sequence[i]; // stored
            top--; // decreased for new items
        }
    }
    // so, we have got the sequence, now we want to print
it
    printf("LIS is");
    for( i = 0; i < maxLength; i++ ) {
        printf(" %d", LisSequence[i]);
    }
    puts("");
}
int main() {
    takeInput();
    int result = LisNlogK();
    printf("The LIS length is %d\n", result);
    findSequence(result);
    return 0;
}
```