Trees

// Centroid Decomposition

struct node

{

    int sz[N];

    node()

    {

        for (int i=0; i<n; i++) fill(c[i], c[i]+L, -1);

        fill(lvl, lvl+n, -1);

        decomp(0, 0, 0);

    }

    void decomp(int u, int p, int dep)

    {

        dfs\_sz(u, u);

        int cent=get(u, u, sz[u]);

        lvl[cent]=dep;

        dfs\_c(cent, cent, dep, cent, 0);

        for (auto v:s[cent]) if (lvl[v]==-1) decomp(v, cent, dep+1);

    }

    int get(int u, int p, const int tsz)

    {

        for (auto v:s[u]) if (v!=p)

        {

            if (lvl[v]!=-1) continue;

            if (tsz<sz[v]\*2) return get(v, u, tsz);

        }

        return u;

    }

    void dfs\_sz(int u, int p)

    {

        sz[u]=1;

        for (auto v:s[u]) if (v!=p)

        {

            if (lvl[v]!=-1) continue;

            dfs\_sz(v, u);

            sz[u]+=sz[v];

        }

    }

    void dfs\_c(int u, int p, const int level, const int cent, int dep)

    {

        c[u][level]=cent; d[u][level]=dep;

        for (auto v:s[u]) if (v!=p)

        {

            if (lvl[v]!=-1) continue;

            dfs\_c(v, u, level, cent, dep+1);

        }

    }

} \*cd;

// Heavy-Light

struct HLD

{

    int dep[N], p[N], sz[N];

    int heavy[N], head[N];

    HLD()

    {

        fill(heavy, heavy+n, -1);

        dfs\_sz(0);

        dfs\_hl(0, 0);

    }

    void dfs\_sz(int u)

    {

        sz[u]=1;

        for (auto v:s[u]) if (v!=p[u])

        {

            dep[v]=dep[u]+1; p[v]=u;

            dfs\_sz(v);

            if (heavy[u]==-1 || sz[heavy[u]]<sz[v]) heavy[u]=v;

            sz[u]+=sz[v];

        }

    }

    void dfs\_hl(int u, int H)

    {

        seq[idx]=u; pre[u]=idx++;

        head[u]=H;

        if (heavy[u]!=-1) dfs\_hl(heavy[u], H);

        for (auto v:s[u]) if (v!=p[u] && v!=heavy[u]) dfs\_hl(v, v);

        post[u]=idx-1;

    }

    void update(int x, int y, int val)

    {

        int fx=head[x], fy=head[y];

        while (fx!=fy)

        {

            if (dep[fx]<dep[fy])

            {

                swap(x, y);

                swap(fx, fy);

            }

            seg->update(pre[fx], pre[x], val);

            x=p[fx]; fx=head[x];

        }

        if (dep[x]>dep[y]) swap(x, y);

        seg->update(pre[x], pre[y], val);

    }

    int query(int x, int y)

    {

        int fx=head[x], fy=head[y]; int ans=0;

        while (fx!=fy)

        {

            if (dep[fx]<dep[fy])

            {

                swap(x, y);

                swap(fx, fy);

            }

            ans += seg->query(pre[fx], pre[x]);

            x=p[fx]; fx=head[x];

        }

        if (dep[x]>dep[y]) swap(x, y);

        return ans + seg->query(pre[x], pre[y]);

    }

} \*hld;

// LCA (binary lifting)

struct LCA

{

    int pre[N], post[N], idx=0;

    int anc[N][L], dep[N];

    LCA()

    {

        for (int i=0; i<n; i++) fill(anc[i], anc[i]+L, -1);

        dep[0]=0; dfs(0, 0);

    }

    void dfs(int u, int p)

    {

        pre[u]=idx++;

        anc[u][0]=p;

        for (int i=1; i<L; i++)

        {

            anc[u][i]=anc[anc[u][i-1]][i-1];

            if (anc[u][i]==-1) break;

        }

        for (auto v:s[u]) if (v!=p)

        {

            dep[v]=dep[u]+1;

            dfs(v, u);

        }

        post[u]=idx-1;

    }

    bool check(int u, int v) { return pre[u]<=pre[v] && post[u]>=post[v]; }

    int query(int u, int v)

    {

        if (check(u, v)) return u;

        if (check(v, u)) return v;

        if (dep[u]>dep[v]) swap(u, v);

        for (int i=L-1; i>=0; i--) if (anc[u][i]!=-1 && !check(anc[u][i], v)) u=anc[u][i];

        return anc[u][0];

    }

} \*lca;

// LCA (FCB Algorithm)

struct LCA

{

    int st[N][L], lg[N];

    int pre[N], dep[N], idx=0;

    LCA()

    {

        dep[0]=0; dfs(0, 0);

        for (int i=2; i<=idx; i++) lg[i]=lg[i>>1]+1;

        for (int j=1; (1<<j)<=idx; j++)

        {

            for (int i=0; i+(1<<j)-1<idx; i++)

            {

                st[i][j]=f(st[i][j-1], st[i+(1<<(j-1))][j-1]);

            }

        }

    }

    void dfs(int u, int p)

    {

        st[idx][0]=u; pre[u]=idx++;

        for (auto v:s[u]) if (v!=p)

        {

            dep[v]=dep[u]+1;

            dfs(v, u);

            st[idx][0]=u; idx++;

        }

    }

    int query(int u, int v)

    {

        u=pre[u]; v=pre[v];

        if (u>v) swap(u, v);

        int i=lg[v-u+1];

        return f(st[u][i], st[v-(1<<i)+1][i]);

    }

    int f(int u, int v) { return dep[u]<dep[v]?u:v; }

} \*lca;

// Prim's Algorithm

int prim()

{

    int key[N]; bool vis[N]={0};

    fill(key, key+n, 1e18);

    priority\_queue<PII, vector<PII>, greater<PII>> q;

    q.push({0, 0});

    key[0]=0;

    while (q.size())

    {

        int u=q.top().sec; q.pop();

        if (vis[u]) continue;

        vis[u]=1;

        for (auto [v, w]:s[u])

        {

            if (key[v]>w)

            {

                key[v]=w;

                q.push({key[v], v});

            }

        }

    }

    int ans=0;

    for (int u=1; u<n; u++) ans+=key[u];

    return ans;

}

// Tarjan Algorithm & DFS Tree

void tarjan(int u, int p)

{

    vis[u]=1;

    pre[u]=low[u]=i++;

    int cnt=0;

    for (auto v:s[u])

    {

        if (v==p) continue;

        if (!vis[v])

        {

            tarjan(v, u);

            low[u]=min(low[u], low[v]);

            if (low[v]>=pre[u] && u!=p) cut[u]=1;

            cnt++;

        }

        else low[u]=min(low[u], pre[v]);

    }

    if (cnt>=2 && u==p) cut[u]=1;

}