Graph Theory

// A-star Algorithm

struct node

{

    int d, u;

    bool operator<(const node x) const { return d+rd[u]>x.d+rd[x.u]; }

};

void dijkstra()

{

    bool vis[N]={0};

    for (int i=0; i<n; i++) rd[i]=1e18;

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

    q.push({0, 0});

    rd[0]=0;

    while (q.size())

    {

        int u=q.top().sec; q.pop();

        if (vis[u]) continue;

        vis[u]=1;

        for (auto [v, w]:rs[u])

        {

            if (rd[v]>rd[u]+w)

            {

                rd[v]=rd[u]+w;

                q.push({rd[v], v});

            }

        }

    }

}

void astar()

{

    priority\_queue<node> q;

    q.push({0, n-1});

    while (q.size())

    {

        auto [d, u]=q.top(); q.pop();

        if (u==0)

        {

            cnt++; ans[cnt]=d;

            if (cnt==k) return;

            continue;

        }

        for (auto [v, w]:s[u]) q.push({d+w, v});

    }

}

// BFS (0-1)

void bfs()

{

    fill(d, d+n, 1e18);

    deque<int> q;

    q.push\_back(0);

    d[0]=0;

    while (q.size())

    {

        int u=q.front(); q.pop\_front();

        for (auto [v, w]:s[u])

        {

            if (d[v]>d[u]+w)

            {

                d[v]=d[u]+w;

                if (w==0) q.push\_front(v);

                else q.push\_back(v);

            }

        }

    }

}

// Condensation Graph

int n, m;

int x[M], y[M];

vector<int> s[N], rs[N];

int a[N], val[N]={0};

bool vis[N]={0};

stack<int> st;

int g[N];

int dp[N], ans;

void dfs\_topo(int u)

{

    vis[u]=1;

    for (auto v:s[u]) if (!vis[v]) dfs\_topo(v);

    st.push(u);

}

void dfs\_scc(int u, int r)

{

    vis[u]=1; g[u]=r;

    val[r]+=a[u];

    for (auto v:rs[u]) if (!vis[v]) dfs\_scc(v, r);

}

void dfs\_dp(int u)

{

    vis[u]=1;

    for (auto v:s[u])

    {

        if (!vis[v]) dfs\_dp(v);

        dp[u]=max(dp[u], dp[v]);

    }

    dp[u]+=val[u];

}

    for (int i=0; i<n; i++) if (!vis[i]) dfs\_topo(i);

    fill(vis, vis+n, 0);

    while (st.size())

    {

        int u=st.top(); st.pop();

        if (!vis[u]) dfs\_scc(u, u);

    }

    for (int i=0; i<n; i++) s[i].clear();

    for (int i=0; i<m; i++)

    {

        if (g[x[i]]!=g[y[i]])

        {

            x[i]=g[x[i]]; y[i]=g[y[i]];

            s[x[i]].push\_back(y[i]);

        }

    }

// Dial's Algorithm

void dial()

{

    bool vis[N]={0};

    fill(d, d+n, 1e18);

    queue<int> q[W];

    q[0].push(0);

    d[0]=0;

    int cur=0, sz=1;

    while (sz>0)

    {

        while (!q[cur%W].size()) cur=(cur+1)%W;

        int u=q[cur].front(); q[cur].pop(); sz--;

        if (vis[u]) continue;

        vis[u]=1;

        for (auto [v, w]:s[u])

        {

            if (d[v]>d[u]+w)

            {

                d[v]=d[u]+w;

                q[d[v]%W].push(v); sz++;

            }

        }

    }

}

// Dijkstra's Algorithm

void dijkstra()

{

    bool vis[N]={0};

    fill(d, d+n, 1e18);

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

    q.push({0, 0});

    d[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 (d[v]>d[u]+w)

            {

                d[v]=d[u]+w;

                q.push({d[v], v});

            }

        }

    }

}

// Floyd-Warshall Algorithm

void floyd()

{

    for (int i=0; i<n; i++)

    {

        for (int u=0; u<n; u++)

        {

            for (int v=0; v<n; v++)

            {

                d[u][v]=min(d[u][v], d[u][i]+d[i][v]);

            }

        }

    }

}

// Johnson's Algorithm

bool spfa()

{

    bool inq[N]={0}; int cnt[N]={0};

    fill(rd, rd+n, 1e18);

    queue<int> q;

    q.push(n);

    rd[n]=0;

    inq[n]=1;

    cnt[n]=0;

    while (q.size())

    {

        int u=q.front(); q.pop();

        inq[u]=0;

        for (auto [v, w]:s[u])

        {

            if (rd[v]>rd[u]+w)

            {

                rd[v]=rd[u]+w;

                cnt[v]=cnt[u]+1;

                if (cnt[v]>n) return 0;

                if (!inq[v])

                {

                    q.push(v);

                    inq[v]=1;

                }

            }

        }

    }

    return 1;

}

void dijkstra(int src)

{

    bool vis[N]={0};

    fill(d, d+n, 1e18);

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

    q.push({0, src});

    d[src]=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 (d[v]>d[u]+w)

            {

                d[v]=d[u]+w;

                q.push({d[v], v});

            }

        }

    }

}

    if (!spfa())

    {

        cout<<-1<<endl;

        return 0;

    }

    for (int u=0; u<n; u++)

    {

        for (int i=0; i<s[u].size(); i++)

        {

            s[u][i].sec+=rd[u]-rd[s[u][i].fir];

        }

    }

    for (int u=0; u<n; u++)

    {

        dijkstra(u);

        for (int v=0; v<n; v++)

        {

            if (d[v]==1e18) cout<<1e18<<' ';

            else cout<<d[v]+rd[v]-rd[u]<<' ';

        }

        cout<<endl;

    }

// SPFA Algorithm

void spfa()

{

    bool inq[N]={0};

    fill(d, d+n, 1e18);

    queue<int> q;

    q.push(0);

    d[0]=0;

    inq[0]=1;

    while (q.size())

    {

        int u=q.front(); q.pop();

        inq[u]=0;

        for (auto [v, w]:s[u])

        {

            if (d[v]>d[u]+w)

            {

                d[v]=d[u]+w;

                if (!inq[v])

                {

                    q.push(v);

                    inq[v]=1;

                }

            }

        }

    }

}