My Templates

Bigragh

Bigraph Match

```
//
// Created by 24087 on 24-11-14.
//
#include <algorithm>
#include <climits>
#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>
```

```cpp
#include <limits>

class Dinic {
    struct Edge {
        int to;
        long long capacity, flow;
        Edge(int to, long long capacity) : to(to), capacity(capacity), flow(0) {}
    };

    int n, source, sink;
    std::vector<std::vector<int>> adj;    // 邻接表存储边的索引
    std::vector<Edge> edges;              // 存储边的信息
    std::vector<int> level;               // 分层图
    std::vector<int> ptr;                 // 当前弧优化的指针
    const long long INF = std::numeric_limits<long long>::max(); // 无限大表示

    bool bfs() {
        std::queue<int> q;
        level.assign(n + 1, -1);           // 初始化分层图，从1开始
        level[source] = 0;
        q.push(source);

        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int idx : adj[u]) {
                const Edge& e = edges[idx];
                if (e.flow < e.capacity && level[e.to] == -1) {
                    level[e.to] = level[u] + 1;
                    q.push(e.to);
                }
            }
        }

        return level[sink] != -1;          // 是否能够到达汇点
    }

    long long dfs(int u, long long pushed) {
        if (u == sink) return pushed;
        for (int& i = ptr[u]; i < adj[u].size(); ++i) {
            int idx = adj[u][i];
            Edge& e = edges[idx];
            if (level[u] + 1 != level[e.to] || e.flow == e.capacity) continue;
            long long flow = dfs(e.to, std::min(pushed, e.capacity - e.flow));
            if (flow > 0) {
                e.flow += flow;
                edges[idx ^ 1].flow -= flow;  // 更新反向边
                return flow;
            }
        }
        return 0;
    }

public:
    Dinic(int n, int source, int sink) : n(n), source(source), sink(sink) {
        adj.resize(n + 1);    // 从1开始，多分配一个位置
```

```
        level.resize(n + 1);
        ptr.resize(n + 1);
    }

    void add_edge(int u, int v, long long capacity) {
        edges.emplace_back(v, capacity);    // 正向边
        edges.emplace_back(u, 0);           // 反向边，容量为0
        adj[u].push_back(edges.size() - 2); // 正向边的索引
        adj[v].push_back(edges.size() - 1); // 反向边的索引
    }

    long long max_flow() {
        long long total_flow = 0;
        while (bfs()) {
            ptr.assign(n + 1, 0);           // 每次BFS后重置指针
            while (const long long flow = dfs(source, INF)) {
                total_flow += flow;
            }
        }
        return total_flow;
    }

    void debug_print_flow() {
        for (int i = 1; i <= n; i++) {
            for (int idx : adj[i]) {
                const Edge& e = edges[idx];
                std::cerr << i << "->" << e.to << " " << edges[idx].capacity <<
std::endl;
            }
        }
    }
};

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n, m, e;
    std::cin >> n >> m >> e;

    Dinic dinic(n + m + 2, n + m + 1, n + m + 2);

    for (int i = 1; i <= n; i++) {
        dinic.add_edge(n + m + 1, i, 1);
    }
    for(int i = 1; i <= m; i++) {
        dinic.add_edge(n + i, n + m + 2, 1);
    }
    for(int i = 1, u, v; i <= e; i++) {
        std::cin >> u >> v;
        dinic.add_edge(u, n + v, 1);
    }

    std::cout << dinic.max_flow() << std::endl;
```

```
}
```

Bigragh Weight Match

```
//
// Created by 24087 on 24-11-14.
//
#include <climits>
#include <iostream>
#include <map>
#include <queue>
#include <vector>

#define int long long

class MinCostMaxFlow {
    const int INF = LLONG_MAX;

   public:
    explicit MinCostMaxFlow(int n)
        : n(n), adj(n + 1), dis(n + 1), vis(n + 1), cur(n + 1), ret(0) {}

    void add_edge(int u, int v, int w, int c) {
        adj[u].emplace_back(Edge{v, w, c, static_cast<int>(adj[v].size()),
true});
        adj[v].emplace_back(
            Edge{u, 0, -c, static_cast<int>(adj[u].size()) - 1, false});
    }

    int min_cost_max_flow(int s, int t) {
        int max_flow = 0;
        while (spfa(s, t)) {
            std::fill(vis.begin(), vis.end(), false);
            int flow;
            while ((flow = dfs(s, t, INF))) {
                max_flow += flow;
            }
        }
        return max_flow;
    }

    int get_cost() const { return ret; }

    void debug_print_flows() {
        for (int i = 1; i <= n; i++) {
            for (auto &edge : adj[i]) {
                std::cerr << i << "->" << edge.to << ": "
                          << adj[edge.to][edge.rev].cap << '\n';
            }
        }
    }

    void print_ans() {
        std::map<int, int> mp;
        int nn = (n - 2) / 2;
```

```
        // std::cerr << nn << std::endl;
        for (int i = 1; i <= nn; i++) {
            for (auto &edge : adj[i]) {
                if (edge.is_positive && edge.cap == 0) {
                    mp[edge.to - nn] = i;
                }
            }
        }
        for (int i = 1; i <= nn; i++) {
            // if (mp[i] == 0) {
            //     continue;
            // }
            std::cout << mp[i] << ' ';
        }
    }

private:
    struct Edge {
        int to, cap, cost, rev;
        bool is_positive;
    };

    int n, ret;
    std::vector<std::vector<Edge>> adj;
    std::vector<int> dis, cur;
    std::vector<bool> vis;

    bool spfa(int s, int t) {
        std::fill(dis.begin(), dis.end(), INF);
        dis[s] = 0;
        std::queue<int> q;
        q.push(s);
        vis[s] = true;

        while (!q.empty()) {
            int u = q.front();
            q.pop();
            vis[u] = false;

            for (const auto &e : adj[u]) {
                if (e.cap > 0 && dis[e.to] > dis[u] + e.cost) {
                    dis[e.to] = dis[u] + e.cost;
                    if (!vis[e.to]) {
                        q.push(e.to);
                        vis[e.to] = true;
                    }
                }
            }
        }
        return dis[t] != INF;
    }

    int dfs(int u, int t, int flow) {
        if (u == t) return flow;
        vis[u] = true;
        int total_flow = 0;
```

```
        for (auto &e : adj[u]) {
            if (!vis[e.to] && e.cap > 0 && dis[e.to] == dis[u] + e.cost) {
                int pushed = dfs(e.to, t, std::min(flow - total_flow, e.cap));
                if (pushed > 0) {
                    e.cap -= pushed;
                    adj[e.to][e.rev].cap += pushed;
                    ret += pushed * e.cost;
                    total_flow += pushed;
                    if (total_flow == flow) break;
                }
            }
        }
        vis[u] = false;
        return total_flow;
    }
};

signed main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n, m;
    std::cin >> n >> m;

    MinCostMaxFlow mcmf(n * 2 + 2);

    for (int i = 1; i <= n; i++) {
        mcmf.add_edge(2 * n + 1, i, 1, 0);
        //mcmf.add_edge(i, 2 * n + 2, 1, 0);
        mcmf.add_edge(i + n, 2 * n + 2, 1, 0);
    }

    for (int i = 1, u, v, w; i <= m; i++) {
        std::cin >> u >> v >> w;
        mcmf.add_edge(u, v + n, 1, -w);
    }

    auto flow = mcmf.min_cost_max_flow(2 * n + 1, 2 * n + 2);
    auto cost = -mcmf.get_cost();

    //std::cerr << flow << '\n';
    std::cout << cost << '\n';

    mcmf.print_ans();
}
```

Binary Index Tree

```
class BIT {
    using ll = long long;
    int n;
    std::vector<ll> tr;
    constexpr static int low_bit(const int x) {
```

```
        return x & -x;
    }

public:
    BIT(int n): n(n), tr(n + 1) {}
    void add(int idx, ll val) {
        for(int i = idx; i <= n; i += low_bit(i)) {
            tr[i] += val;
        }
    }
    ll query(int idx) {
        ll res = 0;
        for(int i = idx; i; i -= low_bit(i)) {
            res += tr[i];
        }
        return res;
    }
    ll query(int l, int r) {
        return query(r) - query(l - 1);
    }
};

class EBIT {
    class BIT {
        using ll = long long;
        int n;
        std::vector<ll> tr;
        constexpr static int low_bit(const int x) {
            return x & -x;
        }

    public:
        explicit BIT(int n): n(n), tr(n + 1) {}
        void add(int idx, ll val) {
            for(int i = idx; i <= n; i += low_bit(i)) {
                tr[i] += val;
            }
        }
        ll query(int idx) {
            ll res = 0;
            for(int i = idx; i; i -= low_bit(i)) {
                res += tr[i];
            }
            return res;
        }
        ll query(int l, int r) {
            return query(r) - query(l - 1);
        }
    };

    int n;
    BIT d, di;
public:
    explicit EBIT(int n): n(n), d(n), di(n) {}
    //sum(r) = d_sum * (r + 1) - di_sum
    using ll = long long;
```

```
    ll query(int idx) {
        return d.query(idx) * (idx + 1) - di.query(idx);
    }
    ll query(int l, int r) {
        return query(r) - query(l - 1);
    }

    void add(int l, int r, int val) {
        d.add(l, val);
        d.add(r + 1, -val);

        di.add(l, val * l);
        di.add(r + 1, -val * (r + 1));
    }
    void add(int idx, int val) {
        add(idx, idx, val);
    }
};
```

Binary Search

```
int binary_search(int destination, const std::vector<int> &vec) {
    int l = 0, r = static_cast<int>(vec.size()) - 1;
    // 全都不满足条件，归0，区间外
    //全都满足就是最后一个，区间内
    while(l < r) {
        int mid = (l + r + 1) / 2;//attention
        if(vec[mid] < destination) {
            l = mid;
        } else {
            r = mid - 1;
        }
    }
    return r;
}
//此方法是在讲，期望l，r都指向满足条件的最后一个

int binary_search(int destination, const std::vector<int> &vec) {
    int l = 1, r = static_cast<int>(vec.size());
    // 全都不满足条件，归1，区间内
    //全都满足就是最后一个 + 1，区间外
    while(l < r) {
        int mid = (l + r) / 2;//attention
        if(vec[mid] < destination) {
            l = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
//此方法是在讲，期望l，r都指向不满足条件的第一个

int binary_search(int destination, const std::vector<int> &vec) {
```

```
    int l = 0, r = static_cast<int>(vec.size());//n + 1
    while(l + 1 < r) {
        int mid = (l + r) / 2;
        if(vec[mid] < destination) {
            l = mid;
        } else {
            r = mid;
        }
    }
    return r;
}
//此方法是在讲期望l指向满足条件最后一个，r期望指向不满足条件第一个
```

Chtholly Tree

```
long long pow(long long a, long long b, int p) {
    a %= p;//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    long long res = 1;
    while(b) {
        if(b & 1) {
            res = res * a % p;
        }
        a = a * a % p;
        b >>= 1;
    }
    return res;
}

class ChthollyTree {
    struct Node {
        int l, r;
        mutable long long val;
        Node(int l, int r, long long val) : l(l), r(r), val(val) {}
        bool operator<(const Node &x) const { return l < x.l; }
    };
    int n;
    std::set<Node> s;
    auto split(int x) {
        auto it = s.lower_bound(Node(x, 0, 0));
        if (it != s.end() && it->l == x) {
            return it;
        }
        --it;
        int l = it->l, r = it->r;
        long long val = it->val;
        s.erase(it);
        s.emplace(l, x - 1, val);
        return s.emplace(x, r, val).first;
    }

   public:
    explicit ChthollyTree(const int n) : n(n) {
        s.emplace(1, n, 0);  // 不用多插入一位
    }
```

```
    void assign(int l, int r, long long v) {
        const auto itr = split(r + 1);
        const auto itl = split(l);  // 先右后左，防止左边界迭代器失效
        s.erase(itl, itr);
        s.emplace(l, r, v);
    }

    void add(int l, int r, long long v) {
        for (auto itr = split(r + 1), itl = split(l); itl != itr; ++itl) {
            itl -> val += v;
        }
    }
    long long query_power(int l, int r,int x, int p) {
        long long res = 0;
        for (auto itr = split(r + 1), itl = split(l); itl != itr; ++itl) {
            (res += pow(itl->val, x, p) * (itl->r - itl->l + 1)) %= p;
        }
        return res;
    }
    long long get_x_th(int l, int r, int x) {
        std::map<long long, int> mp;
        for (auto itr = split(r + 1), itl = split(l); itl != itr; ++itl) {
            mp[itl->val] += itl->r - itl->l + 1;
        }
        for (const auto & [fst, snd] : mp) {
            if (x <= snd) {
                return fst;
            }
            x -= snd;
        }
        return -1;
    }
};
```

Disjoint Set Union

```
class DisjointSetUnion {
    int n;
    std::vector<int> fa, size;

public:
    explicit DisjointSetUnion(const int _n): n(_n), fa(_n + 1), size(_n + 1, 1) {
        std::iota(fa.begin(), fa.end(), 0);
    }
    int find(const int x) {
        if(fa[x] == x)
            return x;
        return fa[x] = find(fa[x]);
    }
    void unite(int x, int y) {
        x = find(x);
        y = find(y);
        if(x == y)
            return;
        if(size[x] < size[y])
```

```
            std::swap(x, y);
        fa[y] = x;
        size[x] += size[y];
    }
};
```

Tree Decomposition

```
class TreeDecomposition {
    struct Node {
        int fa = 0, siz = 1, dep = 0, son = 0, top = 0, dfn = 0;
    };

    int n, s, mod;
    std::vector<std::vector<int>> tr;
    std::vector<Node> nodes;
    Segment_tree<ll> bit;

    void dfs1(int now, int father, int depth) {
        nodes[now].fa = father;
        nodes[now].dep = depth;
        for (int nxt : tr[now]) {
            if (nxt == father) {
                continue;
            }
            dfs1(nxt, now, depth + 1);
            nodes[now].siz += nodes[nxt].siz;
            if (nodes[nxt].siz >
                nodes[nodes[now].son].siz) {  // 要保证nodes[0].siz = 0
                nodes[now].son = nxt;
            }
        }
    }

    void dfs2(int now, int tp) {
        static int tot = 0;
        nodes[now].top = tp;
        nodes[now].dfn = ++tot;
        if (nodes[now].son == 0) return;
        dfs2(nodes[now].son, tp);
        for (int nxt : tr[now]) {
            if (nxt == nodes[now].son || nxt == nodes[now].fa) continue;
            dfs2(nxt, nxt);
        }
    }

   public:
    TreeDecomposition(int n, int s, int mod, std::vector<std::vector<int>> &&tr,
                      std::vector<int> &&val)
        : n(n), s(s), mod(mod), tr(std::move(tr)), nodes(n + 1), bit(n, mod) {
        nodes[0].siz = 0;  //!!!!!!!
        dfs1(s, 0, 1);
        dfs2(s, s);
        for (int i = 1; i <= n; i++) {
            bit.add(nodes[i].dfn, nodes[i].dfn, val[i]);
```

```
        }
    }

    void modify_lca(int u, int v, int val) {
        while (nodes[u].top != nodes[v].top) {
            if (nodes[nodes[u].top].dep > nodes[nodes[v].top].dep) {
                bit.add(nodes[nodes[u].top].dfn, nodes[u].dfn, val);
                u = nodes[nodes[u].top].fa;
            } else {
                bit.add(nodes[nodes[v].top].dfn, nodes[v].dfn, val);
                v = nodes[nodes[v].top].fa;
            }
        }
        bit.add(nodes[u].dfn, nodes[v].dfn, val);
    }

    void modify_subtree(int u, int val) {
        bit.add(nodes[u].dfn, nodes[u].dfn + nodes[u].siz - 1, val);
    }

    ll query_lca(int u, int v) {
        ll res = 0;
        while (nodes[u].top != nodes[v].top) {
            if (nodes[nodes[u].top].dep > nodes[nodes[v].top].dep) {
                res += bit.ask(nodes[nodes[u].top].dfn, nodes[u].dfn);
                res %= mod;
                u = nodes[nodes[u].top].fa;
            } else {
                res += bit.ask(nodes[nodes[v].top].dfn, nodes[v].dfn);
                res %= mod;
                v = nodes[nodes[v].top].fa;
            }
        }
        res += bit.ask(nodes[u].dfn, nodes[v].dfn);
        res %= mod;
        return res;
    }
    ll query_subtree(int u) {
        return bit.ask(nodes[u].dfn, nodes[u].dfn + nodes[u].siz - 1);
    }
};
```

Monotone Queue

```
#include <iostream>
#include <deque>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n, k;
    std::cin >> n >> k;
```

```
    std::vector<int> vec(n + 1);
    std::deque<int> dq_max, dq_min;

    for(int i = 1; i <= n; i++) {
        std::cin >> vec[i];
    }
    for(int i = 1; i <= n; i++) {
        if(!dq_min.empty() && dq_min.front() + k <= i) {
            dq_min.pop_front();
        }
        while(!dq_min.empty() && vec[dq_min.back()] >= vec[i]) {
            dq_min.pop_back();
        }
        dq_min.push_back(i);
        if(i >= k) {
            std::cout << vec[dq_min.front()] << ' ';
        }
    }
    std::cout << '\n';
    for(int i = 1; i <= n; i++) {
        if(!dq_max.empty() && dq_max.front() + k <= i) {
            dq_max.pop_front();
        }
        while(!dq_max.empty() && vec[dq_max.back()] <= vec[i]) {
            dq_max.pop_back();
        }
        dq_max.push_back(i);
        if(i >= k) {
            std::cout << vec[dq_max.front()] << ' ';
        }
    }
}
```

Monotone Stack

```
//
// Created by 24087 on 24-10-21.
//
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n;
    std::cin >> n;
    std::vector<int> stk, vec(n + 1);
    for(int i = 1; i <= n; ++i) {
        std::cin >> vec[i];
    }

    long long ans = 0;
```

```
    for(int i = n; i > 0; i--) {
        while(!stk.empty() && vec[stk.back()] < vec[i]) {//牛只能看到比自己矮的，所以要找第一个>=自己的
            stk.pop_back();
        }
        ans += stk.empty() ? n - i : stk.back() - i - 1;//居然看不到最高的那个牛
        stk.push_back(i);
    }
    std::cout << ans;
}
```

Max Flow

Dinic

```
class Dinic {
    struct Edge {
        int to;
        long long capacity, flow;
        Edge(int to, long long capacity): to(to), capacity(capacity), flow(0) {}
    };

    int n, source, sink;
    std::vector<std::vector<int>> adj;     // 邻接表存储边的索引
    std::vector<Edge> edges;               // 存储边的信息
    std::vector<int> level;                // 分层图
    std::vector<int> ptr;                  // 当前弧优化的指针
    const long long INF = std::numeric_limits<long long>::max(); // 无限大表示

    bool bfs() {
        std::queue<int> q;
        level.assign(n + 1, -1);           // 初始化分层图，从1开始
        level[source] = 0;
        q.push(source);

        while(!q.empty()) {
            int u = q.front();
            q.pop();
            for(int idx : adj[u]) {
                const Edge &e = edges[idx];
                if(e.flow < e.capacity && level[e.to] == -1) {
                    level[e.to] = level[u] + 1;
                    q.push(e.to);
                }
            }
        }

        return level[sink] != -1;          // 是否能够到达汇点
    }

    long long dfs(int u, long long pushed) {
        if(u == sink) return pushed;
        for(int &i = ptr[u]; i < adj[u].size(); ++i) {
            int idx = adj[u][i];
```

```
            Edge &e = edges[idx];
            if(level[u] + 1 != level[e.to] || e.flow == e.capacity) continue;
            long long flow = dfs(e.to, std::min(pushed, e.capacity - e.flow));
            if(flow > 0) {
                e.flow += flow;
                edges[idx ^ 1].flow -= flow;  // 更新反向边
                return flow;
            }
        }
        return 0;
    }

public:
    Dinic(int n, int source, int sink): n(n), source(source), sink(sink) {
        adj.resize(n + 1);    // 从1开始，多分配一个位置
        level.resize(n + 1);
        ptr.resize(n + 1);
    }

    void add_edge(int u, int v, long long capacity) {
        edges.emplace_back(v, capacity);    // 正向边
        edges.emplace_back(u, 0);           // 反向边，容量为0
        adj[u].push_back(edges.size() - 2); // 正向边的索引
        adj[v].push_back(edges.size() - 1); // 反向边的索引
    }

    long long max_flow() {
        long long total_flow = 0;
        while(bfs()) {
            ptr.assign(n + 1, 0);           // 每次BFS后重置指针
            while(long long flow = dfs(source, INF)) {
                total_flow += flow;
            }
        }
        return total_flow;
    }

    void debug_print_flow() {
        for(int i = 1; i <= n; i++) {
            for(int idx : adj[i]) {
                if((idx & 1) == 1) {
                    continue;
                }
                const Edge &e = edges[idx];
                std::cerr << i << "->" << e.to << " " << e.flow << std::endl;
            }
        }
    }
};


int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);
```

```
    int n, m, s, t;
    std::cin >> n >> m >> s >> t;

    Dinic dinic(n, s, t);

    for(int i = 0; i < m; ++i) {
        int u, v;
        long long capacity;
        std::cin >> u >> v >> capacity;
        dinic.add_edge(u, v, capacity);
    }
    std::cout << dinic.max_flow() << std::endl;

    dinic.debug_print_flow();
    return 0;
}
```

EdmondKarp

```
class EdmondsKarp {
    int n, s, t;
    std::vector<std::unordered_map<int, long long>> graph;  // residual graph
    std::vector<int> fa;

    long long bfs() {
        std::fill(fa.begin(), fa.end(), -1);
        std::queue<std::pair<int, long long>> q;
        q.emplace(s, LLONG_MAX);
        fa[s] = s;

        while (!q.empty()) {
            auto [now, flow] = q.front();
            q.pop();

            for (auto &[to, cap] : graph[now]) {
                if (fa[to] == -1 && cap > 0) {  // 没访问过，且容量大于 0
                    fa[to] = now;
                    auto new_flow = std::min(flow, cap);
                    if (to == t) {
                        int cur = t;
                        while (cur != s) {
                            int prev = fa[cur];
                            graph[prev][cur] -= new_flow;
                            graph[cur][prev] += new_flow;
                            cur = prev;
                        }
                        return new_flow;
                    }
                    q.emplace(to, new_flow);
                }
            }
        }
        return 0;
    }
```

```
public:
    EdmondsKarp(int n, int s, int t)
        : n(n), s(s), t(t), fa(n + 1), graph(n + 1) {}

    void add_edge(int u, int v, int w) {
        graph[u][v] += w;
    }

    long long max_flow() {
        long long total_flow = 0;

        long long new_flow;
        while ((new_flow = bfs())) {
            total_flow += new_flow;
        }
        return total_flow;
    }
};
```

```
                                        w += flow;
            }
        }
        return total_flow;
    }
};
```

FordFulkerson

```
class EdmondsKarp {
    int n, s, t;
    std::vector<std::unordered_map<int, int>> graph;  // residual graph
    std::vector<bool> vis;

    int dfs(int now, int flow) {
        if (now == t) {
            return flow;
        }
        vis[now] = true;
        for (auto [to, val] : graph[now]) {  // 找到一条路即可
            if (vis[to] || val == 0) continue;//一定要判断无效边啊！！vis等着你
            int new_flow = dfs(to, std::min(flow, val));
            if (new_flow > 0) {
                graph[now][to] -= new_flow;
                graph[to][now] += new_flow;
                return new_flow;
            }
        }
        return 0;
    }

   public:
```

```
    EdmondsKarp(int n, int s, int t,
                std::vector<std::unordered_map<int, int>> &&graph)
        : n(n), s(s), t(t), vis(n + 1), graph(std::move(graph)) {}

    long long max_flow() {
        long long ret = 0;
        int tmp;
        do {
            std::fill(vis.begin(), vis.end(), false);  // 每次都需要重新标记访问
            tmp = dfs(s, INT_MAX);
            ret += tmp;
        } while (tmp != 0);
        return ret;
    }
};
```

Min Cost Max Flow

DinicSSP

```
class MinCostMaxFlow {
    const int INF = INT_MAX;
public:
    explicit MinCostMaxFlow(int n) : n(n), adj(n + 1), dis(n + 1), vis(n + 1),
cur(n + 1), ret(0) {}

    void add_edge(int u, int v, int w, int c) {
        adj[u].emplace_back(Edge{v, w, c, static_cast<int>(adj[v].size()),
true});
        adj[v].emplace_back(Edge{u, 0, -c, static_cast<int>(adj[u].size()) - 1,
false});
    }

    int min_cost_max_flow(int s, int t) {
        int max_flow = 0;
        while (spfa(s, t)) {
            std::fill(vis.begin(), vis.end(), false);
            int flow;
            while ((flow = dfs(s, t, INF))) {
                max_flow += flow;
            }
        }
        return max_flow;
    }

    int get_cost() const { return ret; }

    void debug_print_flows() {
        for (int i = 1; i <= n; i++) {
            for(auto &edge : adj[i]) {
                if(edge.is_positive)
                    std::cerr << i << "->" << edge.to << ": " << adj[edge.to]
[edge.rev].cap << '\n';
            }
```

```
        }
    }

private:
    struct Edge {
        int to, cap, cost, rev;
        bool is_positive;
    };

    int n, ret;
    std::vector<std::vector<Edge>> adj;
    std::vector<int> dis, cur;
    std::vector<bool> vis;

    bool spfa(int s, int t) {
        std::fill(dis.begin(), dis.end(), INF);
        dis[s] = 0;
        std::queue<int> q;
        q.push(s);
        vis[s] = true;

        while (!q.empty()) {
            int u = q.front();
            q.pop();
            vis[u] = false;

            for (const auto &e : adj[u]) {
                if (e.cap > 0 && dis[e.to] > dis[u] + e.cost) {
                    dis[e.to] = dis[u] + e.cost;
                    if (!vis[e.to]) {
                        q.push(e.to);
                        vis[e.to] = true;
                    }
                }
            }
        }
        return dis[t] != INF;
    }

    int dfs(int u, int t, int flow) {
        if (u == t) return flow;
        vis[u] = true;
        int total_flow = 0;

        for (auto &e : adj[u]) {
            if (!vis[e.to] && e.cap > 0 && dis[e.to] == dis[u] + e.cost) {
                int pushed = dfs(e.to, t, std::min(flow - total_flow, e.cap));
                if (pushed > 0) {
                    e.cap -= pushed;
                    adj[e.to][e.rev].cap += pushed;
                    ret += pushed * e.cost;
                    total_flow += pushed;
                    if (total_flow == flow) break;
                }
            }
        }
```

```
        vis[u] = false;
        return total_flow;
    }
};


int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n, m, s, t;
    std::cin >> n >> m >> s >> t;
    MinCostMaxFlow mcmf(n);
    for (int i = 1, u, v, w, c; i <= m; i++) {
        std::cin >> u >> v >> w >> c;
        mcmf.add_edge(u, v, w, c);
    }
    auto max_flow = mcmf.min_cost_max_flow(s, t);
    auto min_cost = mcmf.get_cost();
    mcmf.debug_print_flows();
    std::cout << max_flow << " " << min_cost << '\n';
}
```

Segment Tree

```
using ll = long long;
template <class T>
class Segment_tree {
#define out_of_range() (r < tar_l || l > tar_r)
#define in_range() (tar_l <= l && r <= tar_r)
    struct node {
        T sum = 0;
        T add_lzy = 0;
        T mul_lzy = 1;  // 初始值！！！
    };
    // node_val == origin_val * mul_lzy + add_lzy
    int n, mod;
    std::vector<node> data;

    void pull_up(unsigned idx) {
        data[idx].sum = (data[idx << 1].sum + data[idx << 1 | 1].sum) % mod;
    }

    void make_add(unsigned idx, unsigned l, unsigned r, T val) {
        (data[idx].sum += val * (r - l + 1)) %= mod;
        (data[idx].add_lzy += val) %= mod;
    }

    void make_mul(unsigned idx, T val) {
        (data[idx].sum *= val) %= mod;
        (data[idx].add_lzy *= val) %= mod;
        (data[idx].mul_lzy *= val) %= mod;
    }
```

```
    void push_down(unsigned idx, unsigned l, unsigned r) {
        auto mid = l + (r - l) / 2;
        // mul firstly and add secondly
        // new_node_val == origin_val * mul_lzy_1 * mul_lzy_2 + add_lzy *
        // mul_lzy_2
        // + add_lzy_2
        make_mul(idx << 1, data[idx].mul_lzy);
        make_add(idx << 1, l, mid, data[idx].add_lzy);

        make_mul(idx << 1 | 1, data[idx].mul_lzy);
        make_add(idx << 1 | 1, mid + 1, r, data[idx].add_lzy);

        data[idx].add_lzy = 0;
        data[idx].mul_lzy = 1;
    }

    T ask(const unsigned idx, const unsigned l, const unsigned r,
          const unsigned tar_l, const unsigned tar_r) {
        if (out_of_range()) {
            return 0;
        }
        if (in_range()) {
            return data[idx].sum;
        }
        push_down(idx, l, r);
        auto mid = l + (r - l) / 2;
        return (ask(idx << 1, l, mid, tar_l, tar_r) +
                ask(idx << 1 | 1, mid + 1, r, tar_l, tar_r)) %
               mod;
    }

    void add(const unsigned idx, const unsigned l, const unsigned r,
             const unsigned tar_l, const unsigned tar_r, const T val) {
        if (out_of_range()) {
            return;
        }
        if (in_range()) {
            make_add(idx, l, r, val);
            return;
        }
        auto mid = l + (r - l) / 2;
        push_down(idx, l, r);
        add(idx << 1, l, mid, tar_l, tar_r, val);
        add(idx << 1 | 1, mid + 1, r, tar_l, tar_r, val);
        pull_up(idx);
    }

    void mul(const unsigned idx, const unsigned l, const unsigned r,
             const unsigned tar_l, const unsigned tar_r, const T val) {
        if (out_of_range()) {
            return;
        }
        if (in_range()) {
            make_mul(idx, val);
            return;
        }
```

```
        const auto mid = l + (r - l) / 2;
        push_down(idx, l, r);
        mul(idx << 1, l, mid, tar_l, tar_r, val);
        mul(idx << 1 | 1, mid + 1, r, tar_l, tar_r, val);
        pull_up(idx);
    }

   public:
    explicit Segment_tree(const std::vector<T> &nums,
                          const int &p = 571373) noexcept
        : n(nums.size()), mod(p), data(nums.size() * 4 + 5) {
        std::function<void(unsigned, unsigned, unsigned)> build_helper =
            [&](const unsigned idx, const unsigned l, const unsigned r) {
                if (l == r) {
                    data[idx].sum = nums[l - 1] % mod;  // 减一减一啊啊啊啊
                    return;
                }
                const auto mid = l + (r - l) / 2;
                build_helper(idx << 1, l, mid);
                build_helper(idx << 1 | 1, mid + 1, r);
                pull_up(idx);
            };
        build_helper(1, 1, n);
    }

    T ask(const unsigned l, const unsigned r) { return ask(1, 1, n, l, r); }

    void add(const unsigned l, const unsigned r, T val) {
        add(1, 1, n, l, r, val);
    }

    void mul(const unsigned l, const unsigned r, T val) {
        mul(1, 1, n, l, r, val);
    }

#undef out_of_range
#undef in_range
};
```

Sparse Table

```
class SparseTable {
    int n, log_n;
    std::vector<std::vector<int>> vec;

   public:
    SparseTable(int n, const std::vector<int> &arr)
        : n(n),
          log_n(std::floor(std::log2(n))),
          vec(n + 1, std::vector<int>(log_n + 1)) {
        for(int i = 1; i <= n; i++) {
            vec[i][0] = arr[i];
        }
        for(int j = 1; j <= log_n; j++) {
            for(int i = 1; i + (1 << j) - 1 <= n; i++) {
```

```
                vec[i][j] = std::max(vec[i][j - 1], vec[i + (1 << (j - 1))][j -
1]);
            }
        }
    }
    int query(int l, int r) {
        int k = std::floor(std::log2(r - l + 1));
        return std::max(vec[l][k], vec[r - (1 << k) + 1][k]);
    }
};
```

Trie

```
class ZO_Trie {
    static bool get_bit(unsigned x, int i) { return (x >> i) & 1; }

    struct Node {
        std::array<size_t, 2> nxt = {0, 0};
        int is_end = 0;
        int is_suffix = 0;  // include end
    };
    static constexpr int root = 0;
    std::vector<Node> nodes;

   public:
    explicit ZO_Trie() : nodes(1) {}
    void insert(unsigned s) {
        size_t now = root;
        for (int i = 31; i >= 0; --i) {
            if (nodes[now].nxt[get_bit(s, i)] == 0) {
                nodes[now].nxt[get_bit(s, i)] = nodes.size();
                nodes.emplace_back();
            }
            now = nodes[now].nxt[get_bit(s, i)];
            nodes[now].is_suffix++;
        }
        nodes[now].is_end++;
    }
    [[nodiscard]] int count(const unsigned s) const {
        size_t now = root;
        for (int i = 31; i >= 0; --i) {
            if (nodes[now].nxt[get_bit(s, i)] == 0) {
                return 0;
            }
            now = nodes[now].nxt[get_bit(s, i)];
        }
        return nodes[now].is_suffix;
    }
    [[nodiscard]] unsigned find_max_xor(const unsigned s) const {
        size_t now = root;
        unsigned res = 0;
        for (int i = 31; i >= 0; --i) {
            if (nodes[now].nxt[!get_bit(s, i)] != 0) {
                res |= (1 << i);
                now = nodes[now].nxt[!get_bit(s, i)];
```

```
            } else {
                now = nodes[now].nxt[get_bit(s, i)];
            }
        }
        return res;
    }
};
```

```
class Trie {
    struct Node {
        std::unordered_map<char, size_t> nxt;
        int is_end = 0;
        int is_suffix = 0;//include end
    };
    static constexpr int root = 0;
    std::vector<Node> nodes;
public:
    explicit Trie() : nodes(1) {}
    void insert(const std::string &s) {
        size_t now = root;
        for (const char c : s) {
            if (!nodes[now].nxt.count(c)) {
                nodes[now].nxt[c] = nodes.size();
                nodes.emplace_back();
            }
            now = nodes[now].nxt[c];
            nodes[now].is_suffix++;
        }
        nodes[now].is_end++;
    }
    int count(const std::string &s) {
        size_t now = root;
        for (const char c : s) {
            if (!nodes[now].nxt.count(c)) {
                return 0;
            }
            now = nodes[now].nxt[c];
        }
        return nodes[now].is_suffix;
    }
};
```

LCA

Tree Decomposition

```
class LCA {
    int n, s;
    std::vector<std::vector<int>> tr;
    std::vector<int> fa, siz, dep, son, top, dfn;

    void dfs1(int now, int father, int depth) {
        fa[now] = father;
        dep[now] = depth;
```

```
        siz[now] = 1;
        son[now] = 0;
        for(int nxt : tr[now]) {
            if(nxt == father) {
                continue;
            }
            dfs1(nxt, now, depth + 1);
            siz[now] += siz[nxt];
            if(siz[nxt] > siz[son[now]]) {
                son[now] = nxt;
            }
        }
    }
    void dfs2(int now, int tp) {
        static int  tot = 0;
        top[now] = tp;
        dfn[now] = ++tot;
        if(son[now] == 0)
            return;
        dfs2(son[now], tp);
        for(int nxt : tr[now]) {
            if(nxt == son[now] || nxt == fa[now])
                continue;
            dfs2(nxt, nxt);
        }
    }

   public:
    LCA(int n, int s, std::vector<std::vector<int>> &&tr)
        : n(n),
          s(s),
          tr(std::move(tr)),
          fa(n + 1),
          siz(n + 1),
          dep(n + 1),
          son(n + 1),
          top(n + 1),
          dfn(n + 1) {

        dfs1(s, 0, 1);
        dfs2(s, s);
    }

    int operator()(int u, int v) {
        while(top[u] != top[v]) {
            if(dep[top[u]] > dep[top[v]]) {
                u = fa[top[u]];
            } else {
                v = fa[top[v]];
            }
        }
        return dep[u] < dep[v] ? u : v;
    }

};
```

Tarjan

```
class DisjointSetUnion {
    std::vector<int> fa, siz;

   public:
    explicit DisjointSetUnion(const int n) : fa(n + 1), siz(n + 1, 1) {
        std::iota(fa.begin(), fa.end(), 0);
    }

    int find(const int x) {
        if (fa[x] == x) return x;
        return fa[x] = find(fa[x]);
    }

    void merge(int x, int y) {//不能优化
        x = find(x);
        y = find(y);
        if (x == y) return;
        fa[y] = x;
    }
};

class LCA {
    int n, m, s;
    std::vector<std::vector<int>> tr;
    std::vector<std::vector<std::pair<int, int>>> qes;
    std::vector<bool> vis;
    std::vector<int> ans;
    DisjointSetUnion dsu;

    void dfs(int now) {
        vis[now] = true;
        for(int nxt : tr[now]) {
            if(vis[nxt]) {
                continue;
            }
            dfs(nxt);
            dsu.merge(now, nxt);
        }
        for(auto [i, id] : qes[now]) {
            if(vis[i]) {
                ans[id] = dsu.find(i);
            }
        }
    }

   public:
    LCA(int n, int m, int s, std::vector<std::vector<int>> &&tr,
        std::vector<std::vector<std::pair<int, int>>> &&qes)
        : n(n), m(m), s(s), tr(std::move(tr)), qes(std::move(qes)), vis(n + 1),
ans(m), dsu(n) {}
    std::vector<int> get_ans() {
        dfs(s);
        return ans;
```

```
    }
};
```

倍增

```
class LCA {
    int n, s, log_n;
    std::vector<std::vector<int>> tr, fa;
    std::vector<int> depth;

    void build(int now, int father, int dep) {
        depth[now] = dep;
        fa[now][0] = father;
        for (const int nxt : tr[now]) {
            if (nxt == father) {
                continue;
            }
            build(nxt, now, dep + 1);
        }
    }
public:
    LCA(int n, int s, std::vector<std::vector<int>> &&tr)
        : n(n),
          s(s),
          log_n(static_cast<int>(std::ceil(std::log2(n)))),
          tr(std::move(tr)),
          fa(n + 1, std::vector<int>(log_n + 1)),
          depth(n + 1) {

        build(s, 0, 1);

        for (int j = 1; j <= log_n; j++) {
            for (int i = 1; i <= n; i++) {
                fa[i][j] = fa[fa[i][j - 1]][j - 1];
            }
        }
    }

    int operator()(int u, int v) {
        if(depth[u] < depth[v])
            std::swap(u, v);
        for(int j = log_n; j >= 0; j--) {
            if(depth[fa[u][j]] >= depth[v])
                u = fa[u][j];
        }
        if(u == v)
            return u;
        for(int j = log_n; j >= 0; j--) {
            if(fa[u][j] != fa[v][j]) {
                u = fa[u][j];
                v = fa[v][j];
            }
        }
        return fa[u][0];
    }
```

```
};
```

Graph

Shortest Path

Floyd

```
//
// Created by 24087 on 9/19/2024.
//
#include <climits>
#include <iostream>
#include <vector>

int main() {
    int n, m, s;
    std::cin >> n >> m >> s;

    std::vector graph(n + 1, std::vector<long long>(n + 1, INT_MAX));

    for (int i = 1; i <= m; i++) {
        long long u, v, w;
        std::cin >> u >> v >> w;
        graph[u][v] = std::min(graph[u][v], w);//神金，有重边
    }
    for (int i = 1; i <= n; i++) {
        graph[i][i] = 0;
    }

    for (int k = 1; k <= n; k++) {
        for (int x = 1; x <= n; x++) {
            for (int y = 1; y <= n; y++) {
                graph[x][y] = std::min(graph[x][y], graph[x][k] + graph[k][y]);
            }
        }
    }
    for (int i = 1; i <= n; i++) {
        std::cout << graph[s][i] << ' ';
    }
}
```

Dijkstra

```
struct edge {
    int to, val;
};
struct node {
    int idx, m_dis;
    // node(int _i, int _d) : idx(_i), m_dis(_d) {}
    friend bool operator<(const node &a, const node &b) { return a.m_dis >
b.m_dis; }
};
```

```cpp
int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n, m, s;
    std::cin >> n >> m >> s;

    std::vector graph(n + 1, std::vector<edge>());
    std::vector<int> dis(n + 1, INT_MAX);
    std::vector<bool> vis(n + 1);

    for (int i = 1; i <= m; i++) {
        int u, v, w;
        std::cin >> u >> v >> w;
        graph[u].push_back({v, w});
    }

    std::priority_queue<node> q;
    dis[s] = 0;

    q.push({s, 0});

    while (!q.empty()) {
        const int now = q.top().idx;
        q.pop();
        if (vis[now])
            continue;
        vis[now] = true;
        for (auto [to, val]: graph[now]) {
            if (vis[to])
                continue;
            if (dis[to] > static_cast<long long>(dis[now]) + val) {
                dis[to] = dis[now] + val;
                q.push({to, dis[to]});
                // vis[to] = true; !!!!NO!!!!!!
            }
        }
    }


    for (int i = 1; i <= n; i++) {
        std::cout << dis[i] << ' ';
    }
}
```

SPFA

```cpp
struct edge {
    int to, val;
};

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
```

```
    std::cout.tie(nullptr);

    int n, m, s;
    std::cin >> n >> m >> s;

    std::vector graph(n + 1, std::vector<edge>());
    std::vector<int> dis(n + 1, INT_MAX);
    std::vector<bool> vis(n + 1);

    for (int i = 1; i <= m; i++) {
        int u, v, w;
        std::cin >> u >> v >> w;
        graph[u].push_back({v, w});
    }

    std::deque<int> q;
    dis[s] = 0;
    vis[s] = true;
    q.push_back(s);

    while (!q.empty()) {
        const int now = q.front();
        q.pop_front();
        vis[now] = false;


        for (auto [to, val]: graph[now]) {
            if (dis[to] > static_cast<long long>(dis[now]) + val) {
                dis[to] = dis[now] + val;
                if(!vis[to]) {
                    q.push_back(to);
                    vis[to] = true;
                }
            }
        }
    }

    for (int i = 1; i <= n; i++) {
        std::cout << dis[i] << ' ';
    }
}
```

SPFA(2)

```
struct edge {
    int to, val;
};
struct node {
    int idx, m_dis;
    // node(int _i, int _d) : idx(_i), m_dis(_d) {}
    friend bool operator<(const node &a, const node &b) { return a.m_dis >
b.m_dis; }
};

int main() {
```

```
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n, m, s;
    std::cin >> n >> m >> s;

    std::vector graph(n + 1, std::vector<edge>());
    std::vector<int> dis(n + 1, INT_MAX);

    for (int i = 1; i <= m; i++) {
        int u, v, w;
        std::cin >> u >> v >> w;
        graph[u].push_back({v, w});
    }

    std::priority_queue<node> q;
    dis[s] = 0;

    q.push({s, 0});

    while (!q.empty()) {
        const int now = q.top().idx;
        q.pop();
        for (auto [to, val]: graph[now]) {
            if (dis[to] > static_cast<long long>(dis[now]) + val) {
                dis[to] = dis[now] + val;
                q.push({to, dis[to]});
                //vis[to] = true; !!!!NO!!!!!!
            }
        }
    }

    for (int i = 1; i <= n; i++) {
        std::cout << dis[i] << ' ';
    }
}
```

SCC

Kosaraju

```
class Kosaraju {
    int n, cnt = 0;
    std::vector<std::vector<int>> g, scc, rev, reduced;
    std::vector<int> topo, color;
    std::vector<bool> vis;

    void topo_sort(int now) {
        vis[now] = true;
        for (auto nxt : g[now]) {
            if (!vis[nxt]) {
                topo_sort(nxt);
            }
        }
```

```
        topo.push_back(now);
    }
    void dfs(int now) {
        color[now] = cnt;
        scc.back().push_back(now);
        for (auto nxt : rev[now]) {
            if (!color[nxt]) {
                dfs(nxt);
            }
        }
    }
    void calculate() {
        for (int i = 1; i <= n; i++) {
            if (!vis[i]) {
                topo_sort(i);
            }
        }
        for (auto it = topo.rbegin(); it != topo.rend(); it++) {
            if (!color[*it]) {
                scc.emplace_back();
                cnt++;
                dfs(*it);
            }
        }
    }

    void reduce() {
        reduced.resize(cnt + 1);
        for (int i = 1; i <= n; i++) {
            for (auto j : g[i]) {
                if (color[i] != color[j]) {
                    reduced[color[i]].push_back(color[j]);
                }
            }
        }
        for (auto &vec : reduced) {
            std::sort(vec.begin(), vec.end());
            vec.erase(std::unique(vec.begin(), vec.end()), vec.end());
        }
    }

public:
    Kosaraju(const int _n, std::vector<std::vector<int>> _vec)
        : n(_n),
          g(std::move(_vec)),
          scc(1),
          rev(_n + 1),
          color(_n + 1),
          vis(_n + 1) {
        for (auto &vec : g) {
            std::sort(vec.begin(), vec.end());
            vec.erase(std::unique(vec.begin(), vec.end()), vec.end());
        }
        for (auto i = 1; i <= _n; i++) {
            for (auto j : g[i]) {
                rev[j].push_back(i);
```

```
            }
        }
        calculate();
    }
    auto &get_color() { return color; }
    auto &get_topo() { return topo; }
    auto get_cnt() const { return cnt; }
    auto &get_scc() {
        for (auto &vec : scc) {
            std::sort(vec.begin(), vec.end());
        }
        return scc;
    }
    auto &get_reduced() {
        reduce();
        return reduced;
    }
};
```

Topo Sort

Kahn

```
class TopoSort {
    int n;
    std::vector<std::vector<int>> g;
    std::vector<int> in_degree, ans;
    void bfs() {
        std::queue<int> q;
        for(int i = 1; i <= n; i++) {
            if(in_degree[i] == 0) {
                q.push(i);
            }
        }
        while(!q.empty()) {
            int now = q.front();
            q.pop();
            ans.push_back(now);
            for(auto nxt : g[now]) {
                if(--in_degree[nxt] == 0) {
                    q.push(nxt);
                }
            }
        }
    }
public:
    TopoSort(int _n, std::vector<std::vector<int>> _g) : n(_n), g(std::move(_g)),
in_degree(_n + 1) {
        for(int i = 1; i <= _n; i++) {
            for(int j : g[i]) {
                in_degree[j]++;
            }
        }
    }
    auto sort() {
```

```
        bfs();
        return ans;
    }

};
```

DFS

```
class TopoSort {
    int n, tot = 0;
    std::vector<std::vector<int>> g;
    std::vector<int> ans, topo_order;

    void dfs(int now) {
        for(auto i : g[now]) {
            if(topo_order[i]) {
                continue;
            }
            dfs(i);
        }
        topo_order[now] = ++tot;
        ans.push_back(now);
    }

   public:
    TopoSort(int _n, std::vector<std::vector<int>> _g)
        : n(_n), g(std::move(_g)), topo_order(_n + 1) {}

    auto sort() {
        for(int i = 1; i <= n; i++) {
            if(!topo_order[i]) {
                dfs(i);
            }
        }
        std::reverse(ans.begin(), ans.end());
        return ans;
    }
};
```

calculate_all_topo_order

```
//
// Created by 24087 on 24-11-6.
//
#include <algorithm>
#include <functional>
#include <iostream>
#include <set>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);
```

```cpp
    int n;
    std::cin >> n;
    std::vector<std::vector<int>> g(n + 1);
    std::vector<int> in_degree(n + 1), ans;

    for (int i = 1, x; i <= n; i++) {
        while (true) {
            std::cin >> x;
            if (x == 0) break;
            g[i].push_back(x);
        }
    }

    for(int i = 1; i <= n; i++) {
        for(int j : g[i]) {
            in_degree[j]++;
        }
    }

    std::function<void(int)> dfs = [&](int dep) {
        if(dep == n) {
            for(int & an : ans) {
                std::cout << an << ' ';
            }
            std::cout << '\n';
        }
        for(int i = 1; i <= n; i++) {
            if(in_degree[i] == 0) {
                in_degree[i] = -1;
                for(int j : g[i]) {
                    in_degree[j]--;
                }
                ans.push_back(i);
                dfs(dep + 1);
                ans.pop_back();
                for(int j : g[i]) {
                    in_degree[j]++;
                }
                in_degree[i] = 0;
            }
        }
    };

    dfs(0);

}
```