ProconLibrary

1 Common

1.1 設定ファイル

```
"set tabstop=4 softtabstop=4 shiftwidth=4
set ts=4 sts=4 sw=4
"set expandtab smarttab
set et sta
"set cindent smartindent
set cin si
"set number nocompatible
set nu nocp
syntax enable
noremap j gj
noremap k gk
autocmd BufNewFile  *.cpp  0r ~/tmpl.cpp
```

```
! $ vim ~/filename

remove Lock = Caps_Lock
keysym Caps_Lock = Control_L
add Control = Control_L

! $ xmodmap ~/filename
```

1.2 テンプレート

1.2.1 common.h

```
#include "./stdc++.h" // #include <bits/stdc++.h>

using namespace std;
#define REP(i,n) for(int i=0; i<int(n);i++)
typedef long long LL;

const int INF = 1000000000;
const int MOD = 1000000007;
const double EPS = 1e-8;
```

1.2.2 graph.h

```
typedef vector<int> Node;
typedef vector<Node> Graph;

void add_edge(Graph& G, int a, int b) {
    G[a].push_back(b);
}
```

1.2.3 graph_weight.h

```
struct Edge{
    int dst, cost;
};

typedef vector<Edge> Node;
typedef vector<Node> Graph;

void add_edge(Graph& G, int a, int b, int c) {
    G[a].push_back({b, c});
}
```

1.2.4 graph_flow.h

```
struct Edge{
    int dst, cap, rev;
};
typedef vector<Edge> Node;
typedef vector<Node> Graph;
```

2 グラフ

2.1 用語集

マッチング 辺の集合．どの 2 辺も端点を共有しない．

辺カバー 辺の集合．すべての点に対して，それに接続する辺のいずれかが入っている．(辺で点をカバーする)

安定集合 点の集合．どの 2 点も隣接していない．

点カバー 点の集合．すべての辺に対して，端点のどちらかが入っている．(点で辺をカバーする)

DAG Directed Acyclic Graph．閉路を持たない有向グラフ．

パス 頂点の列 $v_1, v_2, \ldots, v_k$．隣り合う 2 頂点間に辺が存在する．

パスカバー パスの集合．すべての頂点が，集合内のパスによってカバーされる．

独立パスカバー パス同士の間に頂点の共有がないようなパスカバー．

推移性のあるグラフ 頂点 a から b と，b から c に辺があるなら，必ず a から c にも辺があるようなグラフ．

有向グラフの二部グラフ化 1 つの頂点 v を 2 つの頂点 v[入口] と v[出口] に分けて，v から u への辺があるなら，v[出口] から u[入口] に辺を引く．

2.2 公式集

2.2.1 オイラーの多面体定理

連結な平面グラフについて，頂点数 V, 辺数 E, 面数 F の関係は外面も含めて $V - E + F = 2$

2.2.2 マッチングなどの関係式

- | 最大マッチング | ≤ | 最小点カバー | (二部グラフでは等号成立)
- | 最大安定集合 | + | 最小点カバー | = $|V|$

- 孤立点のないグラフについて，| 最大マッチング | + | 最小辺カバー | = $|V|$
- 二部グラフについて，| 最小辺カバー | = | 最大安定集合 |
- $X \in V(G)$ が G の点カバー $\iff$ $V(G) - X$ は G の安定集合

2.2.3　パスカバーの関係式

- | 最小パスカバー | ≤ | 最小独立パスカバー | ≤ | 最大安定集合 |
- 推移性のあるグラフについて，| 最小パスカバー | = | 最大安定集合 |
- DAG について，$|V|$ − | 二部グラフ化したものの最大マッチング | = | 最小独立パスカバー |
- 推移性のある DAG について，| 最大安定集合 | = | 最小独立パスカバー | = | 最小パスカバー | = $|V|$ − | 二部グラフ化したものの最大マッチング |

2.3　最短経路

```
#include "../common/common.h"
#include "../common/graph_weight.h"

// Bellman-Ford
// Complexity: O(VE)
//
// 注意
// - 負の辺があっても問題ない
// - 負閉路がある場合は-INFのリストが返される
vector<int> bellmanford(const Graph& G, int s){
    int n = G.size();
    vector<int> dist(n, INF);

    dist[s] = 0;
    REP(iter, n) {
        bool update = false;
        REP(i, n) if(dist[i] != INF) for(auto& e : G[i]) {
            if(dist[e.dst] > dist[i] + e.cost) {
                dist[e.dst] = dist[i] + e.cost;
                update = true;
            }
        }
        // 更新が完了したときに最短路が求まる
        if(!update) return dist;
    }

    // n回更新が起きたときは負の閉路が存在
    return vector<int>(n, -INF);
}

// Shortest Path Faster Algorithm (SPFA)
// Compexity: O(VE) (for random graph: O(E))
//
// 注意
// - 負の辺があっても問題ない
// - 負閉路がある場合無限ループする
vector<int> SPFA(const Graph& G, int s) {
    int n = G.size();
    vector<int> dist(n, INF);
    vector<bool> inque(n);
    queue<int> que;

    dist[s] = 0;
    que.push(s);
    inque[s] = true;
    while(!que.empty()){
        int v = que.front();
        que.pop();
        inque[v] = false;
        for(Edge e : G[v]) {
            if(dist[e.dst] > dist[v] + e.cost) {
                dist[e.dst] = dist[v] + e.cost;
                if(!inque[e.dst]) {
                    que.push(e.dst);
                    inque[e.dst] = true;
                }
            }
        }
    }
    return dist;
}

// Dijkstra
// Compexity: O((E + V) log V)
//
// 注意
// - 負の辺がある場合は利用できない.
vector<int> dijkstra(const Graph& G, int s){
    typedef pair<int, int> P;
    priority_queue<P, vector<P>, greater<P>> que;
    vector<int> dist(G.size(), INF);
    que.push(P(0, s));
    dist[s] = 0;
    while(!que.empty()){
        P p = que.top(); que.pop();
        int v = p.second, c = p.first;
        if(c > dist[v]) continue;
        for(auto& e : G[v]){
            if(dist[e.dst] > dist[v] + e.cost){
```

```
                dist[e.dst] = dist[v] + e.cost;
                que.push(P(dist[e.dst], e.dst));
            }
        }
    }
    return dist;
}
```

2.4　トポロジカルソート

```
#include "../common/common.h"
#include "../common/graph.h"

bool visit(const Graph &g, int v, vector<int> &order, vector<int> &color) {
    color[v] = 1;
    for(int w : g[v]){
        if (color[w] == 2) continue;
        if (color[w] == 1) return false;
        if (!visit(g, w, order, color)) return false;
    }
    order.push_back(v); color[v] = 2;
    return true;
}
bool topological_sort(const Graph &g, vector<int> &order) {
    int n = g.size();
    vector<int> color(n);
    REP(u, n) if (!color[u] && !visit(g, u, order, color))
        return false;
    reverse(order.begin(), order.end());
    return true;
}
```

2.5　強連結成分分解

```
#include "../common/common.h"
#include "../common/graph.h"

// Tarjan's strongly connected components algorithm
// Complexity: O(|V| + |E|)

// two dfs implementation (reference: spaghetti source)
int scc(const Graph& G, const Graph& RG, vector<int>& cmp) {
    int n = G.size();
    int K = 0; // the number of components

    cmp.assign(n, -1); // cmp[v] := component id of vertex v (0, 1, ..., K-1)
    vector<bool> used(n);
    vector<int> order;

    // ordinary dfs
    function<void(int)> dfs = [&](int u) {
        used[u] = true;
        for(int w : G[u]) if(!used[w]) {
            dfs(w);
        }
        order.push_back(u);
    };
    for(int u = 0; u < n; u++) if(!used[u]) {
        dfs(u);
    }
    reverse(order.begin(), order.end());

    // reverse dfs
    function<void(int)> rdfs = [&](int u) {
        cmp[u] = K;
        for(int w : RG[u]) if(cmp[w] == -1) {
            rdfs(w);
        }
    };
    for(int u : order) if(cmp[u] == -1) {
        rdfs(u);
        K++;
    }

    return K;
}

// one dfs implementation (reference: en.wikipedia.org)
int tarjan(const Graph& G, vector<int>& cmp) {
    int n = G.size();
    cmp.assign(n, -1);

    int K = 0;
    int index = 0;
    vector<int> low(n);
    vector<int> id(n);
    vector<int> color(n, 0);
    stack<int> S;

    function<void(int)> dfs = [&](int v) {
        id[v] = low[v] = index++;
        color[v] = 1;
        S.push(v);
        for(int w : G[v]) {
```

```
            if(color[w] == 0) {
                dfs(w);
                low[v] = min(low[v], low[w]);
            } else if(color[w] == 1) {
                low[v] = min(low[v], id[w]);
            }
        }

        if(low[v] == id[v]) {
            while(true){
                int w = S.top(); S.pop();
                cmp[w] = K;
                color[w] = 2;
                if(w == v) break;
            }
            K++;
        }
    };

    for(int i = 0; i < n; i++) if(cmp[i] == -1) {
        dfs(i);
    }

    for(int i = 0; i < n; i++) {
        cmp[i] = K - 1 - cmp[i];
    }

    return K;
}
```

2.6　2-SAT

```
#include "../common/common.h"
#include "../common/graph.h"
#include "./strongly_connected_component.cpp"

struct SAT2{
    int n, V;
    Graph G, RG;
    vector<int> truth;

    SAT2(int n) : n(n), V(2 * n), G(V) {}

    // p は 真 (<=> "(not p) ならば p")
    void set_t(int p){
        imply(inv(p), p);
    }

    // p または q (<=> "(not a) ならば b" かつ "(not b) ならば a")
    void set_or(int p, int q) {
        imply(inv(p), q);
        imply(inv(q), p);
    }

    // p ならば q
    void imply(int p, int q){
        add_edge(G, p, q);
        add_edge(RG, q, p);
    }


    // 充足可能か判定(各変数の真偽値が‘truth‘に入る)
    bool satisfy(){
        vector<int> comp;
        scc(G, RG, comp);
        truth.assign(n, false);
        for(int i = 0; i < n; i++){
            if(comp[i] == comp[i + n]) return false;
            if(comp[i] > comp[i + n]) truth[i] = true;
        }
        return true;
    }

    // not p
    int inv(int p){
        return (p + n) % V;
    }
};
```

2.7　最大二部マッチング (Ford Furkerson) $O(V(V+E))$

二部グラフの最大マッチングの大きさを求める．

```
#include "../common/common.h"
// 最大二部マッチング O(VE)
//
// /* 頂点の数を指定 */
// BipartiteMatching solver(n);
//
// /* 枝を追加 */
// solver.add_edge(0, 1);
//
// /* 最大マッチングの大きさを出力 */
// cout << solver.matching() << endl; // -> 1
//
```

```
// /* マッチングの相手を出力 */
// cout << solver.match[0] << endl; // -> 1
//
// Verify: RUPC 2014 day2 Problem G
struct BipartiteMatching{
    typedef vector<int> Node;
    typedef vector<Node> Graph;

    Graph G;
    vector<int> match;
    vector<bool> used;

    BipartiteMatching(int N) : G(N) {}

    void add_edge(int u, int v){
        G[u].push_back(v);
        G[v].push_back(u);
    }

    bool dfs(int u){
        used[u] = true;
        for(int v : G[u]){
            int w = match[v];
            if(w < 0 || (!used[w] && dfs(w))){
                match[u] = v;
                match[v] = u;
                return true;
            }
        }
        return false;
    }

    int matching(){
        int res = 0;
        match.assign(G.size(), -1);
        for(int u = 0; u < G.size(); u++){
            if(match[u] < 0){
                used.assign(G.size(), false);
                if(dfs(u)) res++;
            }
        }
        return res;
    }
};
```

2.8　最大二部マッチング (Hopcroft-Karp) $O(E\sqrt{V})$

Ford Furkerson よりも高速に二部グラフの最大マッチングを求めるアルゴリズム

```
#include "../common/common.h"
// 最大二部マッチング O(E sqrt(V))
// /* 左側の頂点の数と右側の頂点の数を指定する */
// Bipartitematching solver(number_of_leftnodes, number_of_rightnodes)
// /* 辺を追加する
// (左側の頂点のindex(0からV1-1), 右側の頂点のindex(0からV2-1)) */
// solver.add_edge(0, 0);
// solver.add_edge(0, 1);
// solver.add_edge(1, 1);
// /* 最大マッチングの大きさを出力する
// // cout << solver.matching() << endl;
// Verified: SPOJ Fast Maximum Matching(2.66 sec)

struct BipartiteMatching{
    typedef vector<int> Node;
    typedef vector<Node> Graph;

    const int V1, V2, V;
    const int NIL;
    Graph G;
    vector<int> match;
    vector<int> level;

    BipartiteMatching(int V1_, int V2_) :
        V1(V1_), V2(V2_), V(V1 + V2), NIL(V), G(V), match(V + 1), level(V + 1)
            {}

    void add_edge(int u, int v){
        G[u].push_back(v + V1);
    }

    bool bfs(){
        queue<int> que;
        REP(i, V1) {
            if(match[i] == NIL){
                level[i] = 0;
                que.push(i);
            }else{
                level[i] = INF;
            }
        }
        level[NIL] = INF;

```

```
        while(!que.empty()){
            int v = que.front(); que.pop();
            if(level[v] < level[NIL]){
                REP(i, G[v].size()){
                    int u = match[G[v][i]];
                    if(level[u] == INF){
                        level[u] = level[v] + 1;
                        que.push(u);
                    }
                }
            }
        }
        return level[NIL] != INF;
    }

    bool dfs(int v){
        if(v == NIL) return true;

        for(int u : G[v]){
            if(level[match[u]] == level[v] + 1 && dfs(match[u])){
                match[u] = v;
                match[v] = u;
                return true;
            }
        }
        level[v] = INF;
        return false;
    }

    int matching(){
        REP(i, G.size()) match[i] = NIL;

        int res = 0;
        while(bfs()){
            for(int v = 0; v < V1; v++){
                if(match[v] == NIL && dfs(v)){
                    res++;
                }
            }
        }
        return res;
    }
};
```

2.9 最大流 (Dinic) $O(EV^2)$

最大流問題を解くアルゴリズム．実用上高速なアルゴリズム (ランダムケースなどの得意なグラフだと体感で O(VE) とか O(V + E) くらいの速さで動くらしい)

注意:`max_flow()` を何度も実行するときは graph を保存しておくこと．(`max_flow()` でグラフの情報が壊れる)

```
#include "../common/common.h"
#include "../common/graph_flow.h"
// Dinic O(V^2 E)
// Verify: Many Problems
//
// 使い方:
// /* 頂点の数を指定する */
// Dinic dinic(n);
// /* 辺を張る(始点, 終点, 辺の容量) */
// dinic.add_edge(0, 1, 3);
// dinic.add_edge(1, 2, 2);
// /* 最大流の大きさを出力(始点, 終点) */
// cout << dinic.max_flow(0, 2) << endl;
//
// 注意:
// - max_flow()は副作用としてグラフを変更するので,
//   一度実行するとそれ以降は正しくmax_flowが求まらない.
//
struct Dinic{
    Graph G;
    vector<int> level;
    vector<int> iter;

    Dinic(int N) : G(N), level(N), iter(N) {}

    void bfs(int s){
        level.assign(G.size(), -1);
        queue<int> que;
        que.push(s);
        level[s] = 0;
        while(!que.empty()){
            int v = que.front(); que.pop();
            for(const auto& e : G[v]){
                if(e.cap > 0 && level[e.dst] < 0){
                    level[e.dst] = level[v] + 1;
                    que.push(e.dst);
                }
            }
        }
    }
```

```

    int dfs(int v, int t, int f){
        if(v == t) return f;
        for(int& i = iter[v]; i < G[v].size(); i++){
            Edge& e = G[v][i];
            if(e.cap > 0 && level[v] < level[e.dst]){
                int d = dfs(e.dst, t, min(f, e.cap));
                if(d > 0){
                    e.cap -= d;
                    G[e.dst][e.rev].cap += d;
                    return d;
                }
            }
        }
        return 0;
    }

    void add_edge(int src, int dst, int cap){
        G[src].push_back({dst, cap, (int)G[dst].size()});
        G[dst].push_back({src, 0, (int)G[src].size() - 1});
    }

    int max_flow(int src, int dst){
        int flow = 0;
        while(true){
            bfs(src);
            if(level[dst] < 0) break;
            iter.assign(G.size(), 0);
            while(true){
                int f = dfs(src, dst, INF);
                if(f <= 0) break;
                flow += f;
            }
        }
        return flow;
    }
};

// Ford-Fulkerson
// 計算量: O(FE)
// Verify: AOJ 2076
struct FordFulkerson{
    Graph G;
    vector<bool> used;
    FordFulkerson(int N) : G(N) {}

    void add_edge(int src, int dst, int cap){
        G[src].push_back({dst, cap, (int)G[dst].size()});
        G[dst].push_back({src, 0, (int)G[src].size() - 1});
    }

    int max_flow(int src, int dst) {
        int flow = 0;
        while(true) {
            used.assign(G.size(), false);
            int f = dfs(src, dst, INT_MAX);
            if(f == 0) break;
            flow += f;
        }
        return flow;
    }

    int dfs(int v, int t, int f) {
        if(v == t) return f;
        used[v] = true;
        for(Edge& e : G[v]) {
            if(e.cap > 0 && !used[e.dst]) {
                int d = dfs(e.dst, t, min(f, e.cap));
                if(d > 0) {
                    e.cap -= d;
                    G[e.dst][e.rev].cap += d;
                    return d;
                }
            }
        }
        return 0;
    }
};
```

2.10　最小費用流 (Primal-Dual) $O(FElogV)$,F は流量

始点 s から終点 t までの流量 f のフローでコストが最小のものを求めるアルゴリズム.

注意:辺のコストが負のときは使えない, 多重辺や自己辺を入れてはいけない

```
#include "../common/common.h"
// 最小費用流 O(F E logV)
// 使い方:
// /* 頂点の数を指定する */
// MinCostFlow solver(n);
// /* 辺を追加する (始点, 終点, 容量, コスト) */
// solver.add_edge(0, 1, 5, 10);
// /* 最小費用流を求める (始点, 終点, 流量) */
// cout << solver.min_cost_flow(0, 1, 5) << endl;
```

```
struct MinCostFlow{
    typedef pair<int, int> P; // (最短距離, 頂点の番号)
    static const int INF = 100000000; // infinity

    struct Edge{
        int dst, cap, cost, rev;
        Edge() {}
        Edge(int d, int c, int cs, int r) :
            dst(d), cap(c), cost(cs), rev(r) {}
    };

    typedef vector<Edge> Node;
    typedef vector<Node> Graph;

    Graph G;

    MinCostFlow(int N) : G(N) {}

    // from から to へ向かう容量cap, 費用costの辺を追加
    void add_edge(int src, int dst, int cap, int cost){
        G[src].push_back(Edge(dst, cap, cost, G[dst].size()));
        G[dst].push_back(Edge(src, 0, -cost, G[src].size() - 1));
    }

    // 頂点sから頂点tへの流量fの最小費用流を求める
    // 流せない場合は-1を返す
    int min_cost_flow(int s, int t, int f){
        int V = G.size();
        vector<int> h(V);
        vector<int> prevv(V), preve(V);
        int res = 0;
        while(f > 0){
            // dijkstraでhを更新する (負の辺がある場合は使えない!)
            priority_queue<P, vector<P>, greater<P>> que;
            vector<int> dist(V, INF);
            dist[s] = 0;
            que.push(P(0, s));

            while(!que.empty()){
                P p = que.top(); que.pop();
                int v = p.second;
                if(dist[v] < p.first) continue;
                for(int i = 0; i < G[v].size(); i++){
                    Edge& e = G[v][i];
                    int ndist = dist[v] + e.cost + h[v] - h[e.dst];
                    if(e.cap > 0 && dist[e.dst] > ndist){
                        dist[e.dst] = ndist;
                        prevv[e.dst] = v;
                        preve[e.dst] = i;
                        que.push(P(ndist, e.dst));
                    }
                }
            }

            if(dist[t] == INF){
                // これ以上流せない
                return -1;
            }

            for(int v = 0; v < V; v++) h[v] += dist[v];

            int d = f;
            for(int v = t; v != s; v = prevv[v]){
                d = min(d, G[prevv[v]][preve[v]].cap);
            }

            f -= d;
            res += d * h[t];
            for(int v = t; v != s; v = prevv[v]){
                Edge& e = G[prevv[v]][preve[v]];
                e.cap -= d;
                G[v][e.rev].cap += d;
            }
        }
        return res;
    }
};
```

2.11 最近共通祖先

```
#include "../common/common.h"
#include "../common/graph.h"

// 最近共通祖先
// 使い方:
// /* コンストラクタ */
// LCA lca(Graph, Root Vertex);
// /* 二つの頂点を指定すると最近共通祖先が返ってくる */
// cout << lca.query(vertex1, vertex2) << endl;

struct LCA{
    int log_v;
    vector<int> depth;
```

```
    vector<vector<int>> par;

    void dfs(const Graph& G, int v, int p, int d){
        depth[v] = d;
        par[v][0] = p;
        for(int next : G[v]) if(next != p) {
            dfs(G, next, v, d + 1);
        }
    }

    LCA(const Graph& G, int root) {
        int n = G.size();

        for(log_v = 0; (1 << log_v) < n; log_v++) { }

        depth.resize(n);
        par.assign(n, vector<int>(log_v + 1));

        dfs(G, root, root, 0);

        for(int k = 0; k + 1 <= log_v; k++){
            for(int v = 0; v < n; v++){
                par[v][k + 1] = par[ par[v][k] ][k];
            }
        }
    }

    int query(int u, int v){
        if(depth[u] > depth[v]) swap(u, v);
        for(int k = 0; k <= log_v; k++){
            if((depth[v] - depth[u]) >> k & 1){
                v = par[v][k];
            }
        }

        if(u == v) return u;

        for(int k = log_v; k >= 0; k--){
            if(par[u][k] != par[v][k]){
                u = par[u][k];
                v = par[v][k];
            }
        }

        return par[u][0];
    }
};
```

2.12 最小シュタイナー木 $O(n4^t + n^2 2^t + n^3)$

シュタイナー木 := V の部分集合 T に対して，T のすべての頂点を含む木のこと

```
#include "../common/common.h"
// Verified : AOJ 1040 (Chocolate with Heart Marks)
// 使い方:
// T : シュタイナー木が含まなければならない頂点集合
// g : グラフの隣接行列表現
// 計算量:
// O(n 3^t + n^2 2^t + n^3).
// t <= 11 くらい
typedef vector<vector<int>> Matrix;
int minimum_steiner_tree(const vector<int>& T, const Matrix &g) {
    const int n = g.size();
    const int numT = T.size();
    if (numT <= 1) return 0;

    Matrix d(g); // all-pair shortest
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                d[i][j] = min( d[i][j], d[i][k] + d[k][j] );

    int OPT[(1 << numT)][n];
    for (int S = 0; S < (1 << numT); ++S)
        for (int x = 0; x < n; ++x)
            OPT[S][x] = INF;

    for (int p = 0; p < numT; ++p) // trivial case
        for (int q = 0; q < n; ++q)
            OPT[1 << p][q] = d[T[p]][q];

    for (int S = 1; S < (1 << numT); ++S) { // DP step
        if (!(S & (S-1))) continue;

        for (int p = 0; p < n; ++p)
            for (int E = 0; E < S; ++E)
                if ((E | S) == S)
                    OPT[S][p] = min( OPT[S][p], OPT[E][p] + OPT[S-E][p] );

        for (int p = 0; p < n; ++p)
            for (int q = 0; q < n; ++q)
                OPT[S][p] = min( OPT[S][p], OPT[S][q] + d[p][q] );
    }

```

```
    int ans = INF;
    for (int S = 0; S < (1 << numT); ++S)
        for (int q = 0; q < n; ++q)
            ans = min(ans, OPT[S][q] + OPT[((1 << numT)-1)-S][q]);

    return ans;
}
```

2.13 最小全域有向木 $O(VE)$

- 強連結成分分解が必要
- グラフが壊されることに注意

```
// 最小全域有向木(Edmonds' algorithm) O(VE)
// (強連結成分分解が必要)
// (グラフが壊されることに注意)
// 注意: 入力のグラフが自己辺を含むときに正しく動かない.
// Verified: AOJ 2309, UVA 11183
//
// 注意: 強連結成分分解を重み付きversionに書き換える必要あり.
#include "../common/common.h"
struct Edge{
    int src, dst, cost;
};
typedef vector<Edge> Node;
typedef vector<Node> Graph;
int MOB(Graph& G, int root){
    int V = G.size();
    int res = 0;

    // 各ノードに入る最小の辺を求める
    vector<Edge> min_edge(V, {-1, -1, INF});
    REP(v, V) REP(i, G[v].size()){
        Edge& e = G[v][i];
        if(min_edge[e.dst].cost > e.cost){
            min_edge[e.dst] = e;
        }
    }

    // コストを足し合わせる
    REP(v, V) if(v != root) {
        if(min_edge[v].cost == INF) return INF; // rootから到達不可能な頂点が存在する
        res += min_edge[v].cost;
    }

    // 各辺のコストを、最小のコスト分だけ減らす
    REP(v, V) REP(i, G[v].size()){
        Edge& e = G[v][i];
        if(e.dst != root) e.cost -= min_edge[e.dst].cost;
    }

    // 強連結成分分解で、ループがあるかどうか調べる
    Graph sG(V);
    Graph sGR(V);
    REP(v, V) if(v != root) {
        Edge& e = min_edge[v];
        sG[e.src].push_back(e);
        sGR[e.dst].push_back({e.dst, e.src, e.cost});
    }
    vector<int> comp;
    int m = scc(sG, sGR, comp); // 強連結成分分解
    if(m == V) return res; // ループがなければ終了

    // 成分の間に辺を張った新しいグラフを作る
    Graph nG(m);
    REP(v, V) REP(i, G[v].size()){
        Edge& e = G[v][i];
        if(comp[v] != comp[e.dst]) nG[comp[v]].push_back({comp[v], comp[e.dst], e.cost});
    }
    return min(INF, res + MOB(nG, comp[root]));
}
```

2.14 支配集合問題

```
#include "../common/common.h"
// 支配集合問題 ref. Operafan library
// Verify: AOJ 1015
//
// N(u) = {u} \cup {v | (u, v) \in E} とする
// 支配集合問題とは，頂点集合 S の 要素 v の N(v) の集合和 が
// V と一致するような S のうち最小のものを求める問題.
//
// 別の言葉で言い換えると，グラフの点を選んで，
// 自分自身と，その点に接続している点に色を塗るような操作を考えたとき，
// グラフのすべての点を塗るために必要な操作の最小回数を求める問題.
//
// 支配集合問題はNP完全であることが知られている.
// 以下のコードは，支配集合問題をバックトラックで解く.
// |V| <= 40 程度なら解ける.
//
// 入力Gは以下を満たすようにする
//     G[u][v] := u から v に辺がつながっているかどうか
```

```
//     (有向グラフの場合はVerifyしていない)
//
// 計算量: O(2^|V|) ( |V| <= 40 程度ならOK )
typedef vector<bool> Array;
typedef vector<Array> Matrix;

int dfs(int n, int k, LL G[], LL cover[], int ord[], LL now, int ans, int &
    bound){
    if(ans >= bound) {
        return bound;
    }
    if(now == (1LL << n) - 1) {
        return bound = ans;
    }
    if(k >= n) {
        return bound;
    }
    if((now | cover[k]) != (1LL << n) - 1) {
        return bound;
    }

    int u = ord[k];
    if((now & G[u]) == G[u]) {
        return dfs(n, k + 1, G ,cover, ord, now, ans, bound);
    }

    return min(dfs(n, k + 1, G ,cover, ord, now | G[u], ans + 1, bound),
            dfs(n, k + 1, G ,cover, ord, now, ans, bound));
}

int dominating_set(const Matrix &G){
    int N = G.size();
    LL M[N];
    memset(M, 0, sizeof(M));
    int cnt[N];
    REP(i, N){
        M[i] = 1 << i;
        cnt[i] = 1;
        REP(j, N) if(G[i][j]){
            M[i] |= 1 << j;
            cnt[i] ++;
        }
    }
    int ord[N];
    REP(i, N) ord[i] = i;

    // sort
    REP(i, N) REP(j, N - 1){
        if(cnt[ord[j]] < cnt[ord[j+1]]){
            swap(ord[j], ord[j+1]);
        }
    }

    LL cover[N + 1];
    cover[N] = 0;
    for(int i = N - 1; i >= 0; i--) {
        cover[i] = cover[i + 1] | M[ord[i]];
    }

    int bound = N;
    return dfs(N, 0, M, cover, ord, 0, 0, bound);
}
```

2.15 極大/最大 独立集合・クリーク

```
#include "../common/common.h"
#include "../common/graph.h"

typedef unsigned long long ULL;

// 重み付き最大独立集合 (重み付き最大クリーク)
// ref. https://sites.google.com/site/indy256/algo/bron_kerbosh
//
// BronKerbosch は 極大クリーク を列挙するアルゴリズム
// 補グラフのクリークは 独立集合 に対応するので,
// 重み付き最大独立集合を解くことができる.
//
// verify: AOJ 2403 (0.53 sec) (重み付き最大独立集合)
// (最速は0.03 secなのでそんなに速くない.
// 速さを求めるならちゃんとした枝刈りを書くべき)
//
// verify: 模擬地区予選2014 I問題 (1sec) (極大独立集合列挙)
// (自前DFSが1.7secなので少し速い)

inline int trail0(ULL s) { return (s ? __builtin_ctzll(s) : 64); }

int BronKerbosch(const vector<ULL>& g, ULL cur, ULL allowed, ULL forbidden,
    const vector<int>& weights) {
    if (allowed == 0 && forbidden == 0) {
        // 極大クリークに対する処理をここに書く
        int res = 0;
        for (int u = trail0(cur); u < g.size(); u += trail0(cur >> (u + 1)) +
            1)
```

```
            res += weights[u];
        return res;
    }
    if (allowed == 0) return -1;
    int res = -1;
    int pivot = trail0(allowed | forbidden);
    ULL z = allowed & ~g[pivot];
    for (int u = trail0(z); u < g.size(); u += trail0(z >> (u + 1)) + 1) {
        res = max(res, BronKerbosch(g, cur | (1ULL << u), allowed & g[u],
            forbidden & g[u], weights));
        allowed ^= 1ULL << u;
        forbidden |= 1ULL << u;
    }
    return res;
}

int maximum_clique(const Graph& G, const vector<int>& weights) {
    int n = G.size();
    assert(n < 64);
    vector<ULL> g(n, 0);
    REP(i, n) for(int j : G[i]) g[i] |= (1ULL << j);
    return BronKerbosch(g, 0, (1ULL << n) - 1, 0, weights);
}

int maximum_independet_set(const Graph& G, const vector<int>& weights) {
    int n = G.size();
    assert(n < 64);
    vector<ULL> g(n);
    REP(i, n) REP(j, n) if(i != j) g[i] |= (1ULL << j);
    REP(i, G.size()) for(int j : G[i]) g[i] ^= (1ULL << j);
    return BronKerbosch(g, 0, (1ULL << n) - 1, 0, weights);
}
```

2.16 オンライントポロジカルソート

```
#include "../common/common.h"
#include "../common/graph.h"
// オンライントポロジカルソート
// 概要:
// オンライン版のトポロジカルソート.
// 辺が加えられるたびに頂点のトポロジカル順序を変更する.
//
// 注意点:
// - 閉路ができるような場合(DAGにならない場合)は考慮していない
//
// 計算量:
// O((V + E) * Q)
// Qはクエリ数
//
// 使い方:
// /* 頂点の個数を指定 */
// OnlineTopologicalSort solver(n);
// /* 辺を追加 */
// solver.add_edge(1, 0);
// solver.add_edge(2, 1);
// /* 頂点uのトポロジカル順序を出力 */
// cout <<< solver.get_order(u) << endl;
// /* トポロジカル順序がiの頂点を出力 */
// for(int i = 0; i < n; i++){
//     cout << solver.get_node(i) << endl;
// }
//
//
// Verify:
// IOPC2014 D Problem
//
class OnlineTopologicalSort{
    Graph G; // グラフ
    vector<int> order; // 頂点uの順位
    vector<int> order_inv; // 順位iの頂点
    vector<bool> color; // dfsで辿れるノードを保存する
    int lb, ub;

    // 頂点xの順位をiに定める
    inline void allocate(int x, int i){
        order[x] = i;
        order_inv[i] = x;
    }

    // 順位がub以下の頂点を経由して辿れる頂点に色を塗る
    void dfs(int v){
        color[v] = true;
        for(int next : G[v]){
            if(order[next] < ub && !color[next]){
                dfs(next);
            }
        }
    }

    // 色が塗られた頂点を右側に寄せる
    void shift_node(){
        vector<int> shift;
        for(int i = lb; i <= ub; i++){
```

```
            int w = order_inv[i];
            if(color[w]){
                color[w] = false;
                shift.push_back(w);
            }else{
                allocate(w, i - shift.size());
            }
        }
        for(int i = 0; i < shift.size(); i++){
            allocate(shift[i], ub - shift.size() + i + 1);
        }
    }

public:

    // 頂点数N
    OnlineTopologicalSort(int N) :
        G(N), order(N), order_inv(N), color(N)
    {
        for(int i = 0; i < N; i++){
            allocate(i, i);
        }
    }

    // 有向辺 a -> b を加える
    void add_edge(int a, int b){
        G[a].push_back(b);
        lb = order[b];
        ub = order[a];
        if(lb < ub){ // bがaよりも左側にあるとき
            dfs(b); // bから辿れる頂点を列挙する
            shift_node(); // bから辿れる頂点を右側に，残りの頂点を左側に，
                          // 順序を保ったまま移動する
        }
    }

    // 頂点uのトポロジカル順序 (0-based index)
    int get_order(int u) const {
        return order[u];
    }

    // トポロジカル順序がnth番目の頂点 (0-based index)
    int get_node(int nth) const {
        return order_inv[nth];
    }

};
```

3 データ構造

3.1 UnionFind

```
#include "../common/common.h"

// UnionFind
struct UnionFind {
    vector<int> data;
    UnionFind(int N) : data(N, -1) { }
    // xとyを併合する
    bool unite(int x, int y) {
        x = root(x); y = root(y);
        if (x != y) {
            if (data[x] > data[y]) swap(x, y);
            data[x] += data[y]; data[y] = x;
        }
        return x != y;
    }
    // xとyが同じ集合にあるか判定する
    bool same(int x, int y) {
        return root(x) == root(y);
    }
    // xを含む集合の要素数を求める
    int size(int x) {
        return -data[root(x)];
    }
    int root(int x) {
        return data[x] < 0 ? x : data[x] = root(data[x]);
    }
};

// UnionFind (重み付き)
struct UnionFindW{
    vector<pair<int, int>> uf; // (parent, offset from parent)
    UnionFindW(int N) {
        for(int i = 0; i < N; i++){
            uf.push_back(make_pair(i, 0));
        }
    }

    // return (root, offset from root)
    pair<int, int> root(int a){
```

```
        if(uf[a].first != a){
            pair<int, int> p = root(uf[a].first);
            uf[a] = make_pair(p.first, p.second + uf[a].second);
        }
        return uf[a];
    }

    // (a, b, offset of [b] - offset of [a])
    bool unite(int a, int b, int d){
        pair<int, int> pa = root(a), pb = root(b);
        int ra = pa.first, rb = pb.first;
        if(ra != rb) {
            uf[ra] = make_pair(rb, pb.second - pa.second + d);
        }
        return ra != rb;
    }

    // 同じ集合に含まれるかどうか
    bool same(int x, int y) {
        return root(x).first == root(y).first;
    }
};
```

3.2 BIT

```
#include "../common/common.h"

// cf. http://hos.ac/slides/20140319_bit.pdf

// Binary Indexed Tree (Fenwick Tree) (0-indexed)
// two queries in O(log n)
// 1. add w to v[at]
// 2. the sum of v[0], v[1], .., v[at]
struct BIT{
    vector<LL> sums;
    BIT(int n) : sums(n) {}

    // v[at] += by
    void add(int at, LL by) {
        while(at < sums.size()){
            sums[at] += by;
            at |= at + 1;
        }
    }

    // v[0] + ... + v[at]
    LL get(int at) {
        LL res = 0;
        while(at >= 0) {
            res += sums[at];
            at = (at & (at + 1)) - 1;
        }
        return res;
    }

    // --- optional ---
    int size() const { return sums.size(); }
    LL operator [](int idx) const { return sums[idx]; }
};

// BIT (range-version) (0-indexed)
// two queries in O(log n)
// 1. add w to v[a], v[a+1], ..., v[b-1]
// 2. get the sum of v[0], v[1], ..., v[c-1]
//
struct BITRange{
    BIT bit0, bit1;
    BITRange(int n) : bit0(n + 1), bit1(n + 1) {}

    // v[a], v[a+1], ..., v[b-1] += by
    void add(int a, int b, LL by) {
        bit0.add(a, -by * a);
        bit0.add(b, +by * b);
        bit1.add(a, by);
        bit1.add(b, -by);
    }

    // v[0] + v[1] + ... + v[c-1]
    LL get(int c) {
        LL A = bit0.get(c);
        LL B = bit1.get(c);
        return A + B * c;
    }
};

// BIT (2D-version) (0-indexed)
// two queries in O(logw * lowh)
// 1. add w to v[y][x]
// 2. get the sum of
// v[0][0], ..., v[0][a],
// v[1][0], ..., v[1][a],
// ...
// v[b][0], ..., v[b][a]
```

```
struct BIT2D{
    typedef vector<LL> vec;
    vector<vec> sums;
    int H, W;

    BIT2D(int h, int w) : sums(h, vec(w)), H(h), W(w) {}

    // v[y][x] += w
    void add(int x, int y, int w) {
        for(int i = y; i < H; i |= i + 1) {
            for(int j = x; j < W; j |= j + 1) {
                sums[i][j] += w;
            }
        }
    }

    // for y in [0, b]:
    //     for x in [0, a]:
    //         ret += v[y][x]
    LL get(int a, int b) {
        LL res = 0;
        for(int i = b; i >= 0; i = (i & (i + 1)) - 1){
            for(int j = a; j >= 0; j = (j & (j + 1)) - 1){
                res += sums[i][j];
            }
        }
        return res;
    }
};

// Integer set implemented by BIT
// Time: O(log n)
// operation:
// 1. insert
// 2. erase
// 3. nth_element (operator [])
// 4. index (nth_element(index(x)) == x)
// 5. etc
struct BITSet{
    BIT bit;
    BITSet(int n) : bit(n) {}
    bool insert(int x) {
        if(count(x) == 1) return false;
        bit.add(x, +1);
        return true;
    }

    bool erase(int x) {
        if(count(x) == 0) return false;
        bit.add(x, -1);
        return true;
    }

    int size() {
        return bit.get(bit.size() - 1);
    }

    void clear() {
        bit = BIT(bit.size());
    }

    int operator[](int idx) {
        if(idx < 0 || idx >= size()) return -1;
        idx ++;
        int x = -1;
        int k = 1;
        while(2 * k < bit.size()){
            k *= 2;
        }
        while(k > 0) {
            if(x + k < bit.size() && bit[x + k] < idx) {
                idx -= bit[x + k];
                x += k;
            }
            k >>= 1;
        }
        return x + 1;
    }

    int index(int x) {
        if(!count(x)) return -1;
        return bit.get(x) - 1;
    }

    int count(int x) {
        return bit.get(x) - bit.get(x - 1);
    }
};
```

3.3 Treap

```
#include "../common/common.h"

```

```
struct Node{
    int val;
    double pri;
    int cnt;
    int sum;
    int min_v;
    Node* lch;
    Node* rch;

    Node(int v, double p) :
        val(v), pri(p), cnt(1), sum(v), min_v(v), lch(NULL), rch(NULL) {}
};

int count(Node* t) {
    return t ? t->cnt : 0;
}
int sum(Node* t) {
    return t ? t->sum : 0;
}
int min(Node* t) {
    return t ? t->min_v : INT_MAX;
}

Node* update(Node* t) {
    t->cnt = 1 + count(t->lch) + count(t->rch);
    t->sum = t->val + sum(t->lch) + sum(t->rch);
    t->min_v = min(t->val, min( min(t->lch), min(t->rch) ));
    return t;
}

Node* merge(Node* l, Node* r) {
    if(!l || !r) return l ? l : r;

    if(l->pri > r->pri) {
        l->rch = merge(l->rch, r);
        return update(l);
    } else {
        r->lch = merge(l, r->lch);
        return update(r);
    }
}

pair<Node*, Node*> split(Node* t, int k) { // [0, k) [k, n)
    if(!t) return pair<Node*, Node*>(NULL, NULL);

    int c = count(t->lch);
    if(k <= c) {
        pair<Node*, Node*> s = split(t->lch, k);
        t->lch = s.second;
        return make_pair(s.first, update(t));
    } else {
        pair<Node*, Node*> s = split(t->rch, k - (c + 1));
        t->rch = s.first;
        return make_pair(update(t), s.second);
    }
}

Node* insert(Node* t, int k, int v) {
    auto p = split(t, k);
    return merge(merge(p.first, new Node(v, rand())), p.second);
}

Node* erase(Node* t, int k) {
    auto p1 = split(t, k);
    auto p2 = split(p1.second, 1);
    return merge(p1.first, p2.second);
}

int minimum(Node *t, int l, int r) {
    if(!t) return INT_MAX;
    int c = count(t->lch);
    int n = count(t);
    // [0, c - 1] c [c + 1, n - 1]
    if(l <= 0 && n - 1 <= r) {
        return min(t);
    }
    int res = INT_MAX;
    if(!(c - 1 < l || r < 0)) {
        res = min(res, minimum(t->lch, l, r));
    }
    if(l <= c && c <= r) {
        res = min(res, t->val);
    }
    if(!(r < c + 1 || n - 1 < l)) {
        int nl = l - (c + 1);
        int nr = r - (c + 1);
        res = min(res, minimum(t->rch, nl, nr));
    }
    return res;
}

// --- for debug ---
```

```cpp

Node* get(Node* t, int k) {
    int c = count(t->lch);
    if(k < c) {
        return get(t->lch, k);
    } else if(k > c) {
        return get(t->rch, k - (c + 1));
    } else {
        return t;
    }
}

void output(Node* t) {
    int n = count(t);
    REP(i, n) {
        cout << get(t, i)->val << "␣";
    }
    cout << endl;
}
```

3.4 Lower Envelope

```cpp
#include "../common/common.h"
// 下側エンベロープ
// 複数の直線が与えられたときに，ある x座標での一番下側のy座標を効率的に計算する．
// この実装では，追加する直線の傾きが単調に減少することを仮定している．
// この仮定を外すときは，2分探索木などが必要となる．
//
// 直線の個数をn，クエリの回数をm としたとき，計算量はO(n + m)．
struct LowerEnvelope{
    int s, t;
    vector<LL> deq_a;
    vector<LL> deq_b;
    // f_i(x) = deq_a[i] * x + deq_b[i]
    // f2が最小値を取る可能性があるか判定
    inline bool check(LL a1, LL b1, LL a2, LL b2, LL a3, LL b3) const {
        return (a2 - a1) * (b3 - b2) >= (b2 - b1) * (a3 - a2);
    }
    LowerEnvelope(int n) :
        s(0), t(0), deq_a(n), deq_b(n) {}
    // 直線 ax + b を追加する． 前回追加した直線より傾きが小さい必要がある．
    void push(LL a, LL b){
        while(s + 1 < t && check(deq_a[t - 2], deq_b[t - 2], deq_a[t - 1], deq_b[t - 1], a, b)) t--;
        deq_a[t] = a;
        deq_b[t++] = b;
    }
    // f_i(x) のうち最小のものを返す． 前回のクエリよりもxが大きい必要がある．
    LL minimum(LL x){
        assert(s < t);
        while(s + 1 < t && deq_a[s] * x + deq_b[s] >= deq_a[s + 1] * x + deq_b[s + 1]) s++;
        return deq_a[s] * x + deq_b[s];
    }
};
```

4 動的計画法

4.1 編集距離

```cpp
#include "../common/common.h"
int edit_distance(const string& a, const string& b){
    int n = a.size(), m = b.size();

    const int MAX_L = 1000;
    int dp[MAX_L + 1][MAX_L + 1] = {};
    int type[MAX_L + 1][MAX_L + 1] = {}; // 0 - nothing, 1 - remove, 2 - add, 3 - replace

    for(int i = 1; i <= n; i++){
        dp[i][0] = i;
        type[i][0] = 1;
    }

    for(int j = 1; j <= m; j++){
        dp[0][j] = j;
        type[0][j] = 2;
    }

    for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++){
        if(a[i] == b[j]){
            dp[i + 1][j + 1] = dp[i][j];
        }else{
            dp[i + 1][j + 1] = min({dp[i][j + 1] + 1, dp[i + 1][j] + 1, dp[i][j] + 1});

        }

        // for restoring
        if(a[i] == b[j]){
```

```
            type[i + 1][j + 1] = 0; // do nothing
        }else if(dp[i + 1][j + 1] == dp[i][j + 1] + 1){
            type[i + 1][j + 1] = 1; // remove
        }else if(dp[i + 1][j + 1] == dp[i + 1][j] + 1){
            type[i + 1][j + 1] = 2; // add
        }else {
            type[i + 1][j + 1] = 3; // replace
        }
    }

    // aからbの変換手順を復元 (s = a)
    for(int i = n, j = m; i > 0 || j > 0;){
        if(type[i][j] == 0){
            i--; j--;
            // do nothing
        }else if(type[i][j] == 1){
            i--;
            // remove a[i] (s.erase(s.begin() + i))
        }else if(type[i][j] == 2){
            j--;
            // insert b[j] (s.insert(s.begin() + i, b[j]))
        }else if(type[i][j] == 3){
            i--; j--;
            // replace a[i] to b[j] (s[i] = b[j])
        }
    }

    return dp[n][m];
}
```

4.2 ナップサック問題 (近似)

```
#include "../common/common.h"
// 0-1ナップサック問題(近似アルゴリズム) (https://gist.github.com/spaghetti-source/9565504_)
//
// 概要:
// 0-1ナップサック問題を解くアルゴリズム.
// 0-1ナップサック問題はアイテムは1度しか使えないナップサック問題.
// このライブラリは嘘解法,近似アルゴリズムなので注意.
// 落とすのは難しいらしい.
//
// 計算量:
// ???
//
// 使い方:
// Knapsack solver;
// /* アイテムを追加 (価値, 重み) *;/
// solver.add_item(1, 2);
// solver.add_item(3, 4);
// /* 重みを指定して最大の価値を出力 */
// cout << solver.solve(W) << endl;
//

class Knapsack {
    typedef long long LL;
    struct item {
        LL v, w;
    };

    LL W;
    vector<item> items;
    LL lv;

    LL solve_rec(size_t k, LL v, LL w) {
        if (w + items[k].w > W) return solve_rec(k+1, v, w);
        LL cv = v, cw = w;
        for (size_t i = k; i < items.size(); ++i) {
            if (cw + items[i].w <= W) {
                cw += items[i].w;
                cv += items[i].v;
            }
        }
        if (lv < cv) lv = cv;
        double fv = v, fw = w;
        for (size_t i = k; i < items.size(); ++i) {
            if (fw + items[i].w <= W) {
                fw += items[i].w;
                fv += items[i].v;
            } else {
                fv += items[i].v * (W - fw) / items[i].w;
                break;
            }
        }
        if (fv - lv < 1 || fv < lv) return lv;
        solve_rec(k+1, v+items[k].v, w+items[k].w);
        return solve_rec(k+1, v, w);
    }

public:
    // 価値v, 重みwのアイテムを追加
    void add_item(LL v, LL w) {
        items.push_back({v, w});
```

```
    }

    // 重みの合計がW以下になるようにアイテムを選んだときの価値の和の最大値を返す.
    LL solve(LL W_) {
        if(items.empty()) return 0;
        W = W_;
        sort(items.begin(), items.end(), [](const item &a, const item &b) {
                return a.v * b.w > a.w * b.v;
                });
        lv = 0;
        return solve_rec(0, 0, 0);
    };
};
```

4.3 最長共通部分列

```
#include "../common/common.h"
// 最大共通部分列 (longest common sequence)
// 計算量: O(nm)
vector<int> lcs(const vector<int>& a, const vector<int>& b){
    // dp part
    const int MAX_L = 1000;
    int dp[MAX_L + 1][MAX_L + 1] = {};
    int type[MAX_L + 1][MAX_L + 1] = {};
    for(int i = 0; i < a.size(); i++)
    for(int j = 0; j < b.size(); j++){
        if(a[i] == b[j]){
            dp[i + 1][j + 1] = dp[i][j] + 1;
            type[i + 1][j + 1] = 0;
        }else if(dp[i + 1][j] < dp[i][j + 1]){
            dp[i + 1][j + 1] = dp[i][j + 1];
            type[i + 1][j + 1] = 1;
        }else{
            dp[i + 1][j + 1] = dp[i + 1][j];
            type[i + 1][j + 1] = 2;
        }
    }

    // restore part
    vector<int> res;
    for(int i = a.size(), j = b.size(); i > 0 && j > 0;){
        if(type[i][j] == 0){
            i--; j--;
            res.push_back(a[i]);
        }else if(type[i][j] == 1){
            i--;
        }else if(type[i][j] == 2){
            j--;
        }
    }
    reverse(res.begin(), res.end());
    return res;
}
```

5 文字列

5.1 Trie

```
#include "../common/common.h"

struct Node{
    int value;
    map<char, Node*> next;
    Node() : value(0) {}
    ~Node(){ for(auto p : next) if(p.second) delete p.second; }
};

Node* find(Node* root, string s){
    Node* p = root;
    for(int i = 0; i < s.size(); i++){
        char c = s[i];
        if(!p->next[c]) p->next[c] = new Node();
        p = p->next[c];
    }
    return p;
}
```

5.2 Aho corasick

```
#include "../common/common.h"
// Aho Corasick
// 複数のパターンのマッチングを, 文字列の長さに線形な時間で行う.
//
// build(patterns)
// パターンマッチングオートマトンを構築する.
// 計算量: O(sum of |patterns_i|)
//
// next_node(p, c):
// オートマトンにおける, 移動先を計算する.
// 引数は, 現在のノードと, 入力文字
//
// match(root, query):
```

```cpp
// マッチするパターンとその位置をベクトルで返す.
// 引数はオートマトンのルートノードと検索文字列.
// 計算量は: O(|query|)
//
// 例:
// vector<string> patterns = {"aaa", "abc"};
//
// Node* root = build(patterns);
//
// vector<P> v = match(root, "aaaabc");
//
// assert(v == vector<P>({{2, 0}, {3, 0}, {5, 1}}));
// // s[0..2] == patterns[0]
// // s[1..3] == patterns[0]
// // s[2..5] == patterns[1]
//
// Verified: AOJ 2212

struct Node{
    map<char, Node*> next;
    Node* fail;
    vector<int> match;
    Node() : fail(NULL) {}
    ~Node(){ for(auto p : next) if(p.second) delete p.second; }
};

Node *build(vector<string> pattens){
    // 1. trie木 をつくる
    Node* root = new Node();
    root->fail = root;
    for(int i = 0; i < pattens.size(); i++){
        Node* p = root;
        for(auto c : pattens[i]){
            if(p->next[c] == 0) p->next[c] = new Node();
            p = p->next[c];
        }
        p->match.push_back(i);
    }

    // 2. failure link を作る
    queue<Node*> que;
    for(int i = 0; i < 128; i++){
        if(!root->next[i]){
            root->next[i] = root;
        }else{
            root->next[i]->fail = root;
            que.push(root->next[i]);
        }

    }
    while(!que.empty()){
        Node* p = que.front(); que.pop();
        for(int i = 0; i < 128; i++) if(p->next[i]) {
            Node* np = p->next[i];

            // add que
            que.push(np);

            // search failure link
            Node* f = p->fail;
            while(!f->next[i]) f = f->fail;
            np->fail = f->next[i];

            // update matching list
            np->match.insert(np->match.end(), np->fail->match.begin(), np->fail->match.end());
        }
    }
    return root;
}

// Trie木のノード p からの 文字 c に対応する移動先
Node* next_node(Node* p, char c) {
    while(!p->next[c]) p = p->fail;
    return p->next[c];
}

// クエリにマッチしたパターンについて
// (last index, pattern id)のリストを返す
typedef pair<int, int> P;
vector<P> match(Node* root, string query){
    int n = query.size();
    vector<P> res;

    Node* p = root;
    REP(i, n) {
        int c = query[i];
        p = next_node(p, c);
        for(int k : p->match){
            res.push_back(P(i, k));
        }
    }
```

```cpp
    return res;
}
```

5.3 Suffix Array

```cpp
#include "../common/common.h"
// Suffix Array (プログラミングコンテストチャレンジブック 2nd edition p.335)
// Suffix Array と 高さ配列を計算する.
// 計算量は: O(n (log n)^2)
//
// sa[0..n]: s[sa[i]..n-1] は i番目のSuffix
// lcp[0..n-1] := s[sa[i]..n-1] と s[sa[i+1]..n-1] の 共通するSuffixの長さ
//
namespace SA{
    const int MAX_N = 1000000; // 入力文字列の最大長

    int n, k;
    int rank[MAX_N + 1];
    int tmp[MAX_N + 1];

    // (rank[i], rank[i + k]) と (rank[j], rank[j + k]) を比較
    bool comp(int i, int j){
        if(rank[i] != rank[j]) return rank[i] < rank[j];
        int ri = i + k <= n ? rank[i + k] : -1;
        int rj = j + k <= n ? rank[j + k] : -1;
        return ri < rj;
    }

    vector<int> buildSA(const string& s){
        n = s.size();
        vector<int> sa(n + 1);

        // 最初は1文字, ランクは文字コードにすればよい
        for(int i = 0; i <= n; i++){
            sa[i] = i;
            rank[i] = i < n ? s[i] : -1;
        }

        // k文字についてソートされているところから、2k文字でソートする
        for(k = 1; k <= n; k *= 2){
            sort(sa.begin(), sa.end(), comp);

            tmp[sa[0]] = 0;
            for(int i = 1; i <= n; i++){
                tmp[sa[i]] = tmp[sa[i - 1]] + (comp(sa[i - 1], sa[i]) ? 1 : 0);
            }
            for(int i = 0; i <= n; i++){
                rank[i] = tmp[i];
            }
        }

        return sa;
    }

    vector<int> buildLCP(string s, const vector<int>& sa){
        n = s.size();
        vector<int> lcp(n);

        for(int i = 0; i <= n; i++) rank[sa[i]] = i;

        int h = 0;
        lcp[0] = 0;
        for(int i = 0; i < n; i++){
            // 文字列中での位置iの接尾辞と、接尾辞配列中でその一つ前の接尾辞のLCPを求める
            int j = sa[rank[i] - 1];

            // hを先頭の分1減らし、後ろが一致しているだけ増やす
            if(h > 0) h--;
            for(; j + h < n && i + h < n; h++){
                if(s[j + h] != s[i + h]) break;
            }

            lcp[rank[i] - 1] = h;
        }

        return lcp;
    }
}
```

5.4 Rolling Hash

```cpp
#include "../common/common.h"
typedef unsigned long long ULL;

// mod 2^64 の ローリングハッシュ
template<ULL B>
struct RHash{
    vector<ULL> pow;
    vector<ULL> hash;
    RHash(const string& s) {
        int n = s.size();
```

```
        pow.assign(n + 1, 1);
        hash.assign(n + 1, 0);
        REP(i, n) {
            pow[i + 1] = pow[i] * B;
            hash[i + 1] = s[i] + hash[i] * B;
        }
    }
    // hash of s[0..i)
    ULL h(int i) {
        return hash[i];
    }
    // hash of s[i..j)
    ULL h(int i, int j) {
        return h(j) - h(i) * pow[j-i];
    }
};

// mod 2^64 が攻撃されているときに使う. (ref. http://hos.ac/blog/)
template<int B, int M>
struct RMHash{
    vector<int> pow;
    vector<int> hash;
    RMHash(const string& s) {
        int n = s.size();
        pow.assign(n + 1, 1);
        hash.assign(n + 1, 0);
        REP(i, n) {
            pow[i + 1] = ((long long)pow[i] * B) % M;
            hash[i + 1] = (s[i] + (long long)hash[i] * B % M) % M;
        }
    }
    // hash of s[0..i)
    int h(int i) {
        return hash[i];
    }
    // hash of s[i..j)
    int h(int i, int j) {
        return (h(j) + M - (long long)h(i) * pow[j-i] % M) % M;
    }
};

// ---
// a が b に含まれているか
bool contain(string a, string b) {
    typedef RHash<1000000007> Hash;
    int al = a.size(), bl = b.size();
    if(al > bl) return false;

    Hash A(a);
    ULL ah = A.h(al);

    Hash B(b);
    for(int i = 0; i + al <= bl; i++) {
        if(B.h(i, i + al) == ah) {
            return true;
        }
    }
    return false;
}
```

6 数学

6.1 組み合わせ数

```
#include "../common/common.h"
// combination 1
// 計算量: O(MAX_N * MAX_K)
// 制約: n < MAX_N, k < MAX_K
const int MAX_N = 1010;
const int MAX_K = 1010;
LL memo[MAX_N][MAX_K];
LL comb1(int n, int k){
    if(k < 0 || k > n) return 0;
    if(n == 0) return 1;
    if(memo[n][k] > 0) return memo[n][k];
    return memo[n][k] = comb1(n - 1, k - 1) + comb1(n - 1, k);
}

// combination 2
// 前計算: O(MAX_P)
// クエリ処理: O(1)
// 制約: n < MAX_P, k < MAX_P
//       MODは素数
const int MAX_P = 100010;
LL inv[MAX_P];
int fact[MAX_P], rfact[MAX_P];
void init(){
    inv[1] = 1;
    for (int i = 2; i < MAX_P; ++i){
        inv[i] = inv[MOD % i] * (MOD - MOD / i) % MOD;
    }
```

```
    fact[0] = rfact[0] = 1;
    for(int i = 1; i < MAX_P; i++){
        fact[i] = ((LL)fact[i - 1] * i) % MOD;
        rfact[i] = ((LL)rfact[i - 1] * inv[i]) % MOD;
    }
}
int comb2(int n, int k){
    return (((LL)fact[n] * rfact[n - k]) % MOD) * rfact[k] % MOD;
}
```

6.2 二次方程式・三次方程式

```
#include "../common/common.h"
// 2次方程式 ax^2 + bx + c = 0の解 (重解を一つにまとめる)
vector<double> quadratic(double a, double b, double c){
    if(abs(a) < EPS){
        // bx + c = 0
        if(abs(b) < EPS){
            // c = 0 のとき任意のxが解. c != 0 のとき解なし.
            return vector<double>();
        }
        return vector<double>(1, -c / b);
    }
    double D = b*b - 4*a*c;
    if(D < 0) return vector<double>();
    if(D == 0) return vector<double>(1, -b/(2.0 * a));

    // |b| >> |ac|の時の桁落ちを避けるために
    // x_1 = (-b-sign(b)*sqrt(D))/(2*a), x_2 = c / (a*x_1)を利用する
    vector<double> res;
    int sign = (b >= 0) ? 1 : -1;
    double x1 = (-b - sign * sqrt(D))/(2.0 * a);
    double x2 = c / (a * x1);
    res.push_back(x1);
    res.push_back(x2);
    return res;
}


//3次方程式 ax^3 + bx^2 + cx + d = 0 の解 (重解を重複して返す)
vector<double> cubic(double a, double b, double c, double d){
    auto f = [&](double x) -> double {
        return a*x*x*x + b*x*x + c*x + d;
    };

    // a を正にする
    if(a < 0){
        a *= -1;
        b *= -1;
        c *= -1;
        d *= -1;
    }

    // 解の一つを二分探索で求める
    double lb = -1e8, ub = 1e8;
    REP(_, 80){
        double x = (ub + lb) / 2;
        if(f(x) > 0){
            ub = x;
        }else if(f(x) < 0){
            lb = x;
        }
    }
    double x1 = (ub + lb) / 2;

    // 残りの二次方程式を解く
    // f(x) = (x - x1) (Ax^2 + Bx + C)
    double A = 1;
    double B = b/a + x1;
    double C = c/a + B * x1;
    vector<double> ans = quadratic(A, B, C);

    if(ans.size() == 1) ans.push_back(ans[0]); // 重解を重複して数える
    ans.push_back(x1);
    return ans;
}
```

6.3 ユークリッドの互除法

```
#include "../common/common.h"

LL gcd(LL a, LL b){
    return b > 0 ? gcd(b, a % b) : a;
}

LL lcm(LL a, LL b){
    return a / gcd(a, b) * b;
}

// a x + b y = gcd(a, b) なる x, y を求める
LL extgcd(LL a, LL b, LL& x, LL& y){
    LL d = a;
    if(b != 0){
        d = extgcd(b, a % b, y, x);
```

```
        y -= (a / b) * x;
    }else{
        x = 1; y = 0;
    }
    return d;
}
```

6.4 ガウスジョルダン

```
#include "../common/common.h"

typedef vector<double> Vec;
typedef vector<Vec> Mat;

// verified : AOJ 2564 Tree Reconstruction
int rank_of_matrix(Mat M){
    int H = M.size();
    int W = M[0].size();
    int cy, cx;
    for(cy = 0, cx = 0; cy < H && cx < W; cy++, cx++){
        for(int y = cy + 1; y < H; y++){
            if(abs(M[cy][cx]) < abs(M[y][cx])){
                swap(M[cy], M[y]);
            }
        }
        if(abs(M[cy][cx]) < EPS){
            cy--;
            continue;
        }
        for(int y = cy + 1; y < H; y++){
            double p = M[y][cx] / M[cy][cx];
            for(int x = cx; x < W; x++){
                M[y][x] -= p * M[cy][x];
            }
        }
    }
    return cy;
}

// verified : some problems
Vec gauss_jordan(const Mat& A, const Vec& b){
    int W = A[0].size();
    int H = A.size();

    Mat B(H, Vec(W + 1));

    for(int y = 0; y < H; y++)
        for(int x = 0; x < W; x++)
            B[y][x] = A[y][x];

    for(int y = 0; y < H; y++)
        B[y][W] = b[y];

    bool unique = true;  // 解が一意かどうか
    int cy = 0; // 現在注目している式

    // 現在注目している変数
    for(int x = 0; x < W; x++){
        int pivot = cy;
        // 注目している変数の係数の絶対値が一番大きい式を選ぶ
        for(int y = cy; y < H; y++){
            if(abs(B[y][x]) > abs(B[pivot][x])) pivot = y;
        }


        // 解が一意でないか,解が存在しない
        if(pivot >= H || abs(B[pivot][x]) < EPS) {
            unique = false;
            continue;
        }

        swap(B[cy], B[pivot]);

        // 注目している変数の係数を1にする
        for(int x2 = x + 1; x2 <= W; x2++) {
            B[cy][x2] /= B[cy][x];
        }


        // y番目の式からx2番目の変数を消去
        for(int y = 0; y < H; y++) if(y != cy)
            for(int x2 = x + 1; x2 <= W; x2++)
                B[y][x2] -= B[y][x] * B[cy][x2];

        // 次の式に注目する
        cy++;
    }


    // 解が存在するかどうか
    for(int y = cy; y < H; y++)
        if(abs(B[y][W]) > EPS)
            return Vec();
```

```

    // 解が複数存在するかどうか
    if(!unique) return Vec();

    // 一意な解を返す
    Vec V(W);
    int cur_x = 0;
    for(int y = 0; y < H; y++){
        if(abs(B[y][cur_x]) > EPS){
            V[cur_x++] = B[y][W];
        }
    }
    return V;
}
```

6.5 Givens Elimination

```
#include "../common/common.h"
// Givens消去法(QR分解)
// 説明
// n x nの正方行列Aとベクトルbを入力として,
// A x = bをみたすベクトルxを返す.
// この実装ではrankA = nを仮定している.
//
// 計算量
// O(n ^ 3)
//
// 使い方
// [Matrix A]
//   n x n の行列A
// [Vector b]
//   要素数nのベクトルb
//
// Verified
// AOJ 2171 Strange Couple

typedef vector<double> Vector;
typedef vector<Vector> Matrix;

// [r, 0]を[x, y]に変換するc, sを計算する
inline void make_param(double x, double y, double& c, double &s){
    double r = sqrt(x * x + y * y);
    c = x / r, s = y / r;
}
// 回転行列[[c, s], [-s, c]]を[x, y]に適用する
inline void rotate(double& x, double& y, double c, double s){
    double u = c * x + s * y;
    double v = -s * x + c * y;
    x = u, y = v;
}

// Ax = bを解く
Vector givens_elimination(Matrix A, Vector b){
    int n = A.size();
    for(int i = 0; i < n; i++){
        for(int i2 = i + 1; i2 < n; i2++){
            double c, s;
            make_param(A[i][i], A[i2][i], c, s);
            rotate(b[i], b[i2], c, s);
            for(int j = i; j < n; j++){
                rotate(A[i][j], A[i2][j], c, s);
            }
        }
    }
    for(int i = n - 1; i >= 0; i--){
        b[i] /= A[i][i];
        for(int j = i - 1; j >= 0; j--){
            b[j] -= A[j][i] * b[i];
        }
    }
    return b;
}
```

6.6 行列演算

```
#include "../common/common.h"
typedef vector<LL> Array;
typedef vector<Array> Matrix;

// 行列の掛け算 O(N * M * S)
Matrix mul(const Matrix& a, const Matrix& b){
    int N = a.size(), M = b[0].size(), S = a[0].size();
    assert(S == b.size());
    Matrix c(N, Array(M));
    REP(i, N) REP(k, S) REP(j, M) {
        c[i][j] += a[i][k] * b[k][j];
        c[i][j] %= MOD;
    }
    return c;
}
// 正方行列の累乗 O(N^3 * logn)
Matrix pow(Matrix a, LL b){
    int N = a.size();
    Matrix c(N, Array(N));
```

```
    REP(i, N) c[i][i] = 1;
    while(b > 0){
        if(b & 1) c = mul(c, a);
        a = mul(a, a);
        b >>= 1;
    }
    return c;
}
```

6.7 miller rabin 素数判定

```
#include "../common/common.h"
#include "./mod.cpp"

// 与えられた数が素数かどうかをO(log^^cb^^863 n)で確率的に判定する.
// 以下の関数はn < 341,550,071,728,321について決定的である.
bool miller_rabin(LL n){
    if(n == 2) return true;
    if(n % 2 == 0 || n <= 1) return false;

    vector<LL> a = {2, 3, 5, 7, 11, 13, 17};

    LL d = n - 1, s = 0;
    while((d & 1) == 0){
        d >>= 1;
        s++;
    }

    for(int i = 0; i < a.size() && a[i] < n; i++){
        LL x = pow_mod(a[i], d, n);
        if(x == 1) continue;
        for(int r = 0; r < s; r++){
            if(x == n - 1) break;
            if(r + 1 == s) return false;
            x = mul_mod(x, x, n);
        }
    }

    return true;
}
```

6.8 剰余演算

```
#include "../common/common.h"

LL extgcd(LL a, LL b, LL& x, LL& y){
    LL d = a;
    if(b != 0){
        d = extgcd(b, a % b, y, x);
        y -= (a / b) * x;
    }else{
        x = 1; y = 0;
    }
    return d;
}

// mod * mod が long long に収まらない場合,
// 足し算でオーバーフローを避けてO(log b)で計算する
// mod * mod が long long に収まるときはreturn a * b % mod;に書き換える
LL mul_mod(LL a, LL b, LL mod){
    if(b == 0) return 0;
    LL res = mul_mod((a + a) % mod, b / 2, mod);
    if(b & 1) res = (res + a) % mod;
    return res;
}

// aのb乗をmodで割った余りをO(log b)で計算する
LL pow_mod(LL a, LL b, LL mod){
    if(b == 0) return 1;
    LL res = pow_mod(mul_mod(a, a, mod), b / 2, mod);
    if(b & 1) res = mul_mod(res, a, mod);
    return res;
}

// a * b % mod == 1 をみたすbを計算する. 計算量はO(log mod)
// modが素数のときは b = a ^ (mod - 2)でも計算できる
LL inv_mod(LL a, LL mod){
    LL x, y;
    extgcd(a, mod, x, y);
    return (x % mod + mod) % mod;
}

// 素数pを法とする, 1..nの逆元のリストを求める.  計算量はO(n)
vector<LL> inverse_list(int n, int p){
    vector<LL> inv(n + 1);
    inv[1] = 1;
    for (int i = 2; i <= n; ++i){
        inv[i] = inv[p % i] * (p - p / i) % p;
    }
    return inv;
}
```

6.9 剰余クラス

```
#include "../common/common.h"
```

```
// 剰余を自動で行うためのクラス
static const unsigned MODVAL = 1000000007;
struct mint {
    unsigned val;
    mint():val(0){}
    mint(int      x):val(x%MODVAL) {}
    mint(unsigned x):val(x%MODVAL) {}
    mint(LL       x):val(x%MODVAL) {}
};
mint& operator+=(mint& x, mint y) { return x = x.val+y.val; }
mint& operator-=(mint& x, mint y) { return x = x.val-y.val+MODVAL; }
mint& operator*=(mint& x, mint y) { return x = LL(x.val)*y.val; }
mint operator+(mint x, mint y) { return x+=y; }
mint operator-(mint x, mint y) { return x-=y; }
mint operator*(mint x, mint y) { return x*=y; }
```

6.10 二分探索・三分探索

```
#include "../common/common.h"

// 単調関数 f の零点を [l, r] の範囲で求める
double find_root(double l, double r, double f(double)){
    int sign = (f(l) > 0 ? +1 : -1);
    REP(_, 50) {
        double x = (l + r) / 2;
        if(sign * f(x) > 0) {
            l = x;
        } else {
            r = x;
        }
    }
    return (l + r) / 2;
}

// 凸関数 f の極大値を [a, b] の範囲で求める
double find_max(double a, double b, double f(double)) {
    REP(_, 86) {
        double c = (a * 2 + b) / 3;
        double d = (a + b * 2) / 3;
        if(f(c) > f(d)) { // '>': maximum, '<': minimum
            b = d;
        } else {
            a = c;
        }
    }
    return (a + b) / 2;
}
```

6.11 有理数クラス

```
#include "../common/common.h"

struct Rational{
    // p : 分子 q : 分母
    LL p, q;
    void normalize(){
        if(q < 0) {
            p *= -1;
            q *= -1;
        }
        LL d = __gcd(abs(p), q);
        if(d == 0){
            p = 0;
            q = 1;
        }else{
            p /= d;
            q /= d;
        }
    }
    Rational(LL p, LL q) : p(p), q(q) {
        normalize();
    }

};

Rational operator + (const Rational& a, const Rational& b){
    return Rational(a.p * b.q + b.p * a.q, a.q * b.q);
}

Rational operator - (const Rational& a, const Rational& b){
    return Rational(a.p * b.q - b.p * a.q, a.q * b.q);
}

Rational operator * (const Rational& a, const Rational& b){
    return Rational(a.p * b.p, a.q * b.q);
}

Rational operator / (const Rational& a, const Rational& b){
    return Rational(a.p * b.q, a.q * b.p);
}

bool operator == (const Rational& a, const Rational& b){
    return (a.p == b.p) && (a.q == b.q);
```

```cpp
}

bool operator < (const Rational& a, const Rational& b){
    // overflowを避けるためにlong doubleを用いる
    return (long double) a.p * b.q < (long double) b.p * a.q;
}
```

6.12 区間篩

```cpp
#include "../common/common.h"

// [a, b)の整数に対して素数テーブルを作る. is_prime[i - a] = true <-> iが素数
vector<bool> segment_sieve(LL a, LL b){
    int q = int(sqrt(b) + 2);
    vector<bool> is_prime(b - a, true);
    vector<bool> is_prime_small(q, true);

    for(int i = 2; (LL)i * i < b; i++){
        if(is_prime_small[i]){
            for(int j = 2 * i; (LL)j * j < b; j += i)
                is_prime_small[j] = false;
            for(LL j = max(2LL, (a + i - 1) / i) * i; j < b; j += i)
                is_prime[j - a] = false;
        }
    }

    return is_prime;
}
```

6.13 数値積分

```cpp
#include "../common/common.h"

// 区間[l, r]をN分割し、各区間を2次関数に近似する
// 計算時間: O(N * f(x))
double simpson(double l, double r, int N, double f(double)){
    double h = (r - l) / (2 * N);
    double S = f(l) + f(r);
    for(int i = 1; i < 2 * N; i += 2){
        S += 4.0 * f(l + h * i);
    }
    for(int i = 2; i < 2 * N; i += 2){
        S += 2.0 * f(l + h * i);
    }
    return S * h / 3.0;
}
```

6.14 Z 変換

```cpp
#include "../common/common.h"

void transform1(int N, int a[]){
    // 変換前 a[S] := f(S)
    // 変換後 a[S] := sum of f(T). T is subset of S.
    REP(i, N)REP(S, 1 << N){
        if(0 == (S & (1 << i))){
            a[S | 1 << i] += a[S];
        }
    }
}

void transform2(int N, int a[]){
    // 変換前 a[S] := f(S)
    // 変換後 a[S] := sum of f(T). T is superset of S.
    REP(i, N)REP(S, 1 << N){
        if(0 == (S & (1 << i))){
            a[S] += a[S | (1 << i)];
        }
    }
}
```

7 幾何

7.1 公式集

正弦定理 外接円の半径を R とすると,

$$a = 2R\sin A$$

余弦定理

$$a^2 = b^2 + c^2 - 2bc\cos A$$

$$\cos A = \frac{b^2 + c^2 - a^2}{2bc}$$

ヘロンの公式 $s = \frac{a+b+c}{2}$ とおくと面積 S は,

$$S = \sqrt{s(s-a)(s-b)(s-c)}$$

三角形の面積 (1 辺と両端角)

$$S = \frac{a^2 \sin B \sin C}{2\sin(B+C)}$$

7.2 Point

```
typedef complex<double> P;
const double EPS = 1e-8;

// 誤差を加味した符号判定
/*
    if(a > EPS) return +1;
    if(a < -EPS) return -1;
    return 0;
*/
int sign(double x){ return x > EPS ? 1 : x < -EPS ? -1 : 0; }

// 内積・外積
double dot(P a, P b){return real(conj(a) * b);}
double cross(P a, P b){return imag(conj(a) * b);}

// OAとOBのなす符号付き角度 [-pi, pi]
// example : (1, 0), (0, 1) -> pi/2
double angle(P a, P b){return arg(conj(a) * b);}

// aをc中心にb[rad]回転
// verify : not yet.
P rotate(P a, double b, P c = P()){return (a - c) * polar(1.0, b) + c;}

/*
   if (cross(b, c) > +EPS)   return +1; // 反時計回り
   if (cross(b, c) < -EPS)   return -1; // 時計回り
   if (dot(b, c) < 0)      return +2; // c--a--b の順番で一直線上
   if (norm(b) < norm(c)) return -2; // a--b--c の順番で一直線上
   return 0;                          // 点が線分ab上にある
*/
int ccw(P a, P b, P c) {
    b -= a; c -= a;
    return sign(cross(b,c)) ? : dot(b,c) < 0 ? 2 : norm(b) < norm(c) ? -2 : 0;
}

enum{ OUT, ON, IN };

// Pointの比較をしたいときだけ定義する.
namespace std{
    bool operator < (const P& a, const P& b) {
        return a.real() != b.real() ? a.real() < b.real() : a.imag() < b.imag
            ();
    }
};
```

7.3 Line

```
typedef vector<P> L;
P vec(L l){return l[1] - l[0];}

// 注意: 端点で交わったり直線が重なったりする場合も交差していると判定する

// 二直線の平行判定
// verify : AOJ0021
bool paralell(L l, L m){return sign(cross(vec(l), vec(m))) == 0;}

// 二直線の同一判定
bool equalLL(L l, L m){return paralell(l, m) && sign(cross(vec(l), m[0] - l
    [0])) == 0;}

// 直線と点の交差判定
// 直線lとl[0]からpへの直線が平行
bool iLP(L l, P p) {return sign(cross(vec(l), p - l[0])) == 0;}

// 線分と点の交差判定(端点の処理に注意)(端点は含むけれども誤差に注意)
// verify : AOJ1279, AOJ2506
bool iSP(L s, P p) {return ccw(s[0], s[1], p) == 0;}

// 直線と線分の交差判定(線分が重なっている時に注意)
// 直線lについて、線分sの端点が異なる側にある
bool iLS(L l, L s) {return sign(cross(vec(l), s[0] - l[0]) * cross(vec(l), s
    [1] - l[0])) <= 0;}

// 二つの線分の交差判定(線分が重なっている時や端点の処理に注意)
bool iSS(L s, L t) {
    auto ok = [](L a, L b){return ccw(a[0],a[1],b[0]) * ccw(a[0],a[1],b[1]) <=
        0;};
    return ok(s,t) && ok(t,s);
}


// 点pから直線lに対する射影
P proj(L l, P p){
    double t = dot(p - l[0], vec(l)) / norm(vec(l));
    return l[0] + t * vec(l);
}

// 点pの直線lに関する反射
P refl(L l, P p){return 2.0 * proj(l, p) - p;}

// 直線と点の距離
```

```cpp
// Verified: AOJ 2201
double dLP(L l, P p){return abs(cross(vec(l), p - l[0])) / abs(vec(l));}

// 線分と点の距離
double dSP(L s, P p){
    if(sign(dot(vec(s), p - s[0])) <= 0) return abs(p - s[0]);
    if(sign(dot(-vec(s), p - s[1])) <= 0) return abs(p - s[1]);
    return dLP(s, p);
}

// 直線と直線の距離
// 平行でないときは0, 平行のときは垂線の長さ
double dLL(L l, L m){ return paralell(l, m) ? dLP(l, m[0]) : 0; }

// 直線と線分の距離
double dLS(L l, L s){ return iLS(l,s) ? 0.0 : min(dLP(l, s[0]), dLP(l, s[1])); }

// 線分と線分の距離
// Verified: AOJ 1157
double dSS(L s, L t){return iSS(s,t) ? 0.0 : min({dSP(s, t[0]), dSP(s, t[1]), dSP(t, s[0]), dSP(t, s[1])});}

// 直線と直線の交点
// Verified: AOJ2579
// size()によって場合分け
// *4 : 直線が重なる  *1 : 交点が1つ存在 *0 : 直線が交差しない
vector<P> pLL(L l, L m){
    double A = cross(vec(l), vec(m));
    double B = cross(vec(l), l[1] - m[0]);
    if(sign(A) == 0 && sign(B) == 0) return {l[0], l[1], m[0], m[1]}; // 二直線が重なっている
    if(sign(A) == 0) return{}; // 直線が交わらない
    return {m[0] + vec(m) * B / A};
}

// 線分と線分の交点
// TODO: {(0, 0), (1, 0)} {(1, 0), (2, 0)}を投げたときに空集合が返ってきてしまう.(←直った?)
vector<P> pSS(L l, L m){
    vector<P> res;
    auto find = [&](P p){
        for(P r : res)
            if(sign(abs(r - p)) == 0)
                return true;
        return false;
    };
    for(P p : pLL(l, m)){
        // 片方が直線の場合は適宜変えること
        if(iSP(l,p) && iSP(m,p) && !find(p)){
            res.push_back(p);
        }
    }
    return res;
}
```

7.4 Polygon

```cpp
typedef vector<P> Pol; // 反時計回りを仮定

// 点が多角形のどこにあるのか判定する
// verify : AOJ0012
int contains(const Pol& A, P p){
    // 点pから半直線をひき、辺と交差する回数を数える
    int in = 0;
    int n = A.size();
    for(int i = 0; i < n; i++){
        P a = A[i] - p;
        P b = A[(i + 1) % n] - p;
        if(a.imag() > b.imag()) swap(a, b);
        // aからbの直線がy=0と交わり、その交点は原点の右側である
        in ^= a.imag() <= 0 && 0 < b.imag() && cross(a, b) < 0;

        if(sign(cross(a, b)) == 0 && sign(dot(a, b)) <= 0) return ON;
    }
    return in ? IN : OUT;
}

// 多角形の面積
// verify : AOJ0079 AOJ1100
double area(const Pol& A) {
    double res = 0;
    int n = A.size();
    for(int i = 0; i < n; i++){
        res += cross(A[i], A[(i + 1) % n]);
    }
    return abs(res) / 2.0;
}
```

7.5 Circle

```cpp
struct C{P p;double r;};

// 円と点の内外判定
```

```cpp
// Verified: AOJ2181, AOJ2579
int contains(C c, P p){
    double d = abs(c.p - p);
    if(sign(d - c.r) > 0) return OUT;
    if(sign(d - c.r) == 0) return ON;
    return IN;
}

// 円と線分の交差判定(境界を含む)
// 接するときに注意!!
// Verified: AOJ0129, AOJ2506, AOJ2579
bool iCS(C c, L l){
    int c1 = contains(c, l[0]);
    int c2 = contains(c, l[1]);
    if(c1 > c2) swap(c1, c2);

    // (OUT, OUT) (OUT, ON) (OUT, IN) (ON, ON) (ON, IN) (IN, IN) の6通り
    if(c1 == OUT && c2 == IN) return true;
    if(c1 == IN  && c2 == IN) return false;
    if(c1 == ON) return true; // (接するとき)
    double d = dSP(l, c.p);
    if(sign(d - c.r) < 0) return true;
    if(sign(d - c.r) == 0) return true; // (接するとき)
    if(sign(d - c.r) > 0) return false;
    assert(false);
}

// 二つの円の交差判定(接する時を含む)
// verified: AOJ 2181
bool iCC(C c, C d){
    // 円の中心同士の距離が、半径の和以下であり、半径の差以上である
    double e = abs(d.p - c.p);
    return sign(e - (c.r + d.r)) <= 0 && sign(e - abs(c.r - d.r)) >= 0;
}

// 円cに対して円dがどの位置にあるか(IN:内側, ON: 交差, OUT: それ以外)
// Verified: AOJ 2181
int contains(C c, C d) {
    double e = abs(c.p - d.p);
    if(sign(c.r - d.r) > 0 && sign(c.r - (d.r + e)) > 0) return IN;
    if(iCC(c, d)) return ON;
    return OUT;
}

// 円と直線の交点
// verify : AOJ2045
vector<P> pCL(C c, L l) {
    vector<P> res;
    P center = proj(l, c.p);
    double d = abs(center - c.p);
    double tt = c.r * c.r - d * d;
    if(tt < 0 && tt > -EPS) tt = 0;
    if(tt < 0) return res;
    double t = sqrt(tt);
    P vect = vec(l);
    vect /= abs(vect);
    res.push_back(center - vect * t);
    if (t > EPS) {
        res.push_back(center + vect * t);
    }
    return res;
}

// 円と線分の交点
vector<P> pCS(C c, L s) {
    vector<P> ret;
    vector<P> nret = pCL(c, s);
    for (int i = 0; i < nret.size(); i++) {
        if (iSP(s, nret[i])) ret.push_back(nret[i]);
    }
    return ret;
}

// 円と円の交点
// verify : AOJ1190
vector<P> pCC(C a, C b) {
    vector<P> res;

    double d = abs(b.p - a.p);
    double l1 = abs(a.r - b.r);
    double l2 = a.r + b.r;

    if(sign(d) == 0 && sign(l1) == 0) assert(false); // 円が等しい
    if(sign(d - l1) < 0 || sign(d - l2) > 0) return res; // 交わらない

    double th1 = arg(b.p - a.p);

    if(sign(d - l1) == 0 || sign(d - l2) == 0) { // 一点で交わる
        res.push_back(a.p + polar(a.r, th1));
    } else { // 二点で交わる
        double th2 = acos( (a.r * a.r - b.r * b.r + d * d) / (2 * a.r * d) );
        res.push_back(a.p + polar(a.r, th1 - th2));
```

```
        res.push_back(a.p + polar(a.r, th1 + th2));
    }

    return res;
}

// 2点を通る半径rの円の中心
// verify : AOJ1132
vector<P> touching_circle2(P a, P b, double r){
    vector<P> res;

    double d = abs(b - a);
    if(d > 2 * r) return res;

    P mid = 0.5 * (a + b);
    P dir = polar(sqrt(r * r - d * d / 4), arg(b - a) + M_PI / 2);
    res.push_back(mid + dir);
    res.push_back(mid - dir);
    return res;
}

// 3点を通る円
C touching_circle3(P a, P b, P c){
    // 2つの垂直二等分線の交点が円の中心
    P mid_ab = (a + b) / 2.0;
    L bis_ab = {mid_ab, (mid_ab - a) * P(0.0, 1.0)};
    P mid_bc = (b + c) / 2.0;
    L bis_bc = {mid_bc, (mid_bc - b) * P(0.0, 1.0)};

    assert(!paralell(bis_ab, bis_bc));

    P center = pLL(bis_ab, bis_bc)[0];
    return {center, abs(a - center)};
}

// 円と円の共通部分の面積を求める.
// ref: nya3j
double cc_area(C c1, C c2) {
    double d = abs(c1.p - c2.p);
    if (c1.r + c2.r < d + EPS) {
        return 0.0;
    } else if (d < abs(c1.r - c2.r) + EPS) {
        double r = min(c1.r, c2.r); // 元は c1.r >? c2.r だった.
        return r * r * M_PI;
    } else {
        double rc = (d*d + c1.r*c1.r - c2.r*c2.r) / (2*d);
        double theta = acos(rc / c1.r);
        double phi = acos((d - rc) / c2.r);
        return c1.r*c1.r*theta + c2.r*c2.r*phi - d*c1.r*sin(theta);
    }
}
```

```
// 円の接線(中心から偏角thの点で接する接線)
// verified: AOJ 2201
L circle_tangent(const C& c, double th){
    P p0 = c.p + polar(c.r, th);
    P p1 = p0 + polar(1.0, th + M_PI / 2);
    return {p0, p1};
}

// 二つの円の共通接線(cの中心から接点へのベクトルの偏角を返す)
// verified: AOJ 2201
// 参考: http://geom.web.fc2.com/geometry/circle-circle-tangent.html
vector<double> common_tangents(const C& c, const C& d){
    vector<double> res;
    P v = d.p - c.p;
    double l = abs(v); // 二円の中心間の距離
    double a = arg(v); // 二円の中心間の偏角
    if(sign(l - abs(c.r - d.r)) > 0){
        // 交わる or 外接 or 離れている
        // 二つの外側接線
        double a1 = acos((c.r - d.r) / l);
        res.push_back(a + a1);
        res.push_back(a - a1);
        if(sign(l - (c.r + d.r)) > 0){
            // 離れている
            // 二つの内側接線
            double a2 = acos((c.r + d.r) / l);
            res.push_back(a + a2);
            res.push_back(a - a2);
        }
    }
    if((sign(l - abs(c.r - d.r)) == 0 || sign(l - (c.r + d.r)) == 0) && sign(l) != 0){
        // 内接 or 外接
        // 一つの接線
        res.push_back(a);
    }
    return res;
}

// 1点を通る円の接線( pがcの外側にあることが前提条件 )
```

```
// verified : AOJ 2579
vector<L> tangents_through_point(const C& c, const P& p){
    vector<L> tangents;
    double d = abs(c.p - p);
    // d ^ 2 == r ^ 2 + e ^ 2
    double e = sqrt(d * d - c.r * c.r); // 点pと円の接点の距離
    // d * sin(th) = r
    double th = asin(c.r / d);
    P q1 = p + (c.p - p) * polar(1.0, +th) * e / d;
    P q2 = p + (c.p - p) * polar(1.0, -th) * e / d;
    tangents.push_back({p, q1});
    tangents.push_back({p, q2});
    return tangents;
}
```

7.6 Convex

```
// 凸包
Pol convex_hull(vector<P> ps) {
    int n = ps.size(), k = 0;
    sort(ps.begin(), ps.end(),[](const P&a, const P&b){
            return real(a) != real(b) ? real(a) < real(b) : imag(a) < imag(b);
            });

    vector<P> ch(2*n);
    for (int i = 0; i < n; ch[k++] = ps[i++]){ // lower-hull
        while (k >= 2 && ccw(ch[k-2], ch[k-1], ps[i]) <= 0) --k;
    }
    for (int i = n-2, t = k+1; i >= 0; ch[k++] = ps[i--]){ // upper-hull
        while (k >= t && ccw(ch[k-2], ch[k-1], ps[i]) <= 0) --k;
    }
    ch.resize(k-1);
    return ch;
}

bool is_convex(const Pol& A){
    int n = A.size();
    for(int i = 0; i < n; i++){
        if(ccw(A[i], A[(i + 1) % n], A[(i + 2) % n]) > 0) return false;
    }
    return true;
}

// 凸多角形の直線による切断。直線の左側だけ残す
// verify : aoj1283
Pol convex_cut(const Pol& A, L l){
    int n = A.size();
    Pol B;
    for(int i = 0; i < n; i++){
        P a = A[i], b = A[(i + 1) % n];
        if(ccw(l[0], l[1], a) != -1) B.push_back(a); //Aが直線lの右側でない
        if(ccw(l[0], l[1], a) * ccw(l[0], l[1], b) < 0)
            B.push_back(pLL(l, {a, b})[0]);
    }
    return B;
}
// 垂直二等分線
// verify: maximamcup2013 D
L bisector(P a, P b){
    P mid = (a + b) / 2.0;
    P vec = (mid - a) * P(0.0, 1.0);
    return {mid, mid + vec};
}
// 点集合psのうちs番目のボロノイ領域
// verify: maximamcup2013 D
Pol voronoi_cell(Pol A, const vector<P>& ps, int s){
    for(int i = 0; i < ps.size(); i++){
        if(i != s) A = convex_cut(A, bisector(ps[s], ps[i]));
    }
    return A;
}
```

7.7 3次元幾何

```
struct P{
    double x, y, z;
    P() : x(0), y(0), z(0) {}
    P(double x, double y, double z) :
        x(x), y(y), z(z) {}
};

P operator + (P a, P b){ return P(a.x + b.x, a.y + b.y, a.z + b.z); }
P operator - (P a, P b){ return P(a.x - b.x, a.y - b.y, a.z - b.z); }
P operator * (double t, P a){ return P(t * a.x, t * a.y, t * a.z); }

double dot(P a, P b){ return a.x * b.x + a.y * b.y + a.z * b.z; }
double abs(P a){ return sqrt(dot(a, a)); }
P cross(P a, P b){
    return P(
            a.y * b.z - a.z * b.y,
            a.z * b.x - a.x * b.z,
            a.x * b.y - a.y * b.x
            );
}
```

```cpp

// 点aと点bを通る直線と，点cの距離
double distanceLP(P a, P b, P c){
    b = b - a; c = c - a;
    double t = dot(b, c) / dot(b, b);
    return abs(c - b * t);
}

// a，bを通る直線と，c，dを通る直線の距離
double distanceLL(P a, P b, P c, P d){
    P v = cross(b - a, d - c); // 法線ベクトル
    P p = c - a;
    if(abs(v) < EPS) return distanceLP(a, b, c);
    double dst = abs(dot(v, p)) / abs(v);
    return dst;
}
```

7.8 Triangle

```cpp
// 三角形の内心
// verify : aoj 1301
P incenter(P p1, P p2, P p3){
    double a = abs(p2 - p3);
    double b = abs(p3 - p1);
    double c = abs(p1 - p2);
    return (a * p1 + b * p2 + c * p3) / (a + b + c);
}
```

7.9 線分アレンジメント

複数の線分が与えられたとき, 次のグラフを作成する.

- 線分の端点と線分対の交点を頂点として持ち,
- 二頂点が同一線分に乗っていて二頂点の間に別の頂点は無いときに二頂点間距離を重みとする辺を持つ

ここではグラフのサイズを減らすために追加条件を課している. この条件は推移的な辺を省略したことになる. 省略したくない場合はグラフ構成のときに cut のすべての対について辺を張ればよい.

注意. 入力の線分の集合に オーバーラップする線分は無いものとする. オーバーラップする線分がある可能性がある場合は, あらかじめ merge すること.

```cpp
// 入力
// ss : 線分のリスト
//
// 出力
// ps : グラフの頂点番号に対応する点が入る
// 返り値 : 上の説明のグラフ
//
// Verified
// AOJ 2113
struct Edge{ int src, dst; double cost; };
typedef vector<Edge> Node;
typedef vector<Node> Graph;
Graph segment_arrangement(const vector<L> &ss, vector<P> &ps) {
    for (int i = 0; i < ss.size(); i++) {
        ps.push_back( ss[i][0] );
        ps.push_back( ss[i][1] );
        for (int j = i+1; j < ss.size(); j++){
            if (iSS(ss[i], ss[j])) {
                ps.push_back( pLL(ss[i], ss[j])[0] );
            }
        }
    }
    sort(ps.begin(), ps.end());
    ps.erase(unique(ps.begin(), ps.end()), ps.end());

    Graph g(ps.size());
    for (int i = 0; i < ss.size(); i++) {
        vector<int> on;
        for (int j = 0; j < ps.size(); j++){
            if (iSP(ss[i], ps[j])){
                on.push_back(j);
            }
        }
        for (int j = 0; j + 1 < on.size(); j++) {
            int a = on[j], b = on[j + 1];
            g[a].push_back( {a, b, abs(ps[a]-ps[b])} );
            g[b].push_back( {b, a, abs(ps[a]-ps[b])} );
        }
    }
    return g;
}

// 線分併合
//
// 線分のリストからオーバーラップするものたちをまとめ，新しい線分のリストを作る．
// 元々の線分のリストにおける順番は破壊される．
//
// Verified
// AOJ 2113
void merge_segments(vector<L>& segs) {
    auto merge_if_able = [](L& s, L t){
        if (abs(cross(s[1]-s[0], t[1]-t[0])) > EPS) return false;
```

```
        if (ccw(s[0], t[0], s[1]) == +1 ||
                ccw(s[0], t[0], s[1]) == -1) return false; // not on the same line
        if (ccw(s[0], s[1], t[0]) == -2 ||
                ccw(t[0], t[1], s[0]) == -2) return false; // separated
        s = { min(s[0], t[0]), max(s[1], t[1]) };
        return true;
    };
    for (int i = 0; i < segs.size(); ++i)
        if (segs[i][1] < segs[i][0])
            swap(segs[i][1], segs[i][0]);
    for (int i = 0; i < segs.size(); ++i)
        for (int j = i+1; j < segs.size(); ++j)
            if (merge_if_able(segs[i], segs[j]))
                segs[j--] = segs.back(), segs.pop_back();
}
```

8 その他

8.1 bit 演算

```
#include "../common/common.h"

void combination(int n, int k) {
    // nCkのビットコンビネーションを辞書順で列挙する
    for(int comb = (1 << k) - 1; comb < (1 << n);){
        // do something here
        int x = comb & -comb, y = comb + x;
        comb = ((comb & ~y) / x >> 1) | y;
    }
}

void subset(int sup) {
    // 集合supの部分集合subを列挙する
    int sub = sup;
    do{
        // do something here
        sub = (sub - 1) & sup;
    } while(sub != sup);
}

/* int __builtin_clz(unsigned int);      | 最上位ビットから数えて0の連続する個数
 * int __builtin_ctz(unsigned int);      | 最下位ビットから数えて0の連続する個数
 * int __builtin_popcount(unsigned int); | 2進数表記中に出現する1の個数
 * int __builtin_ffs(unsigned int);      | 最下位ビットから数えて最初に出現する1の位置
 * - unsigned long longのときは，関数名の末尾にllを加える
 * - 0に対する動作が未定義なことに注意する
 */
```

8.2 日付計算

```
// 1年1月1日からy年m月d日までの日数を計算する((y, m, d) = (1, 1, 1) は 1)
// 7で割った余りで曜日が判定できる(日, 月, 火, ..., 土)
int days(int y, int m, int d) {
    if(m <= 2){ y--; m += 12; }
    return 365*y + y/4 - y/100 + y/400 + 153*(m+1)/5 + d - 428;
}
```

8.3 ダイス

```
#include "../common/common.h"
// サイコロ
enum FACE { TOP, BOTTOM, FRONT, BACK, LEFT, RIGHT };

struct Dice {
    vector<int> val;
    Dice(vector<int> init) : val(init) {
        assert(val.size() == 6);
    }
    void roll_x() {
        roll(TOP, BACK, BOTTOM, FRONT);
    }
    void roll_y() {
        roll(TOP, LEFT, BOTTOM, RIGHT);
    }
    void roll_z() {
        roll(FRONT, RIGHT, BACK, LEFT);
    }
    void roll_r(int r){
        if(r == 0) roll(TOP, LEFT, BOTTOM, RIGHT); // 右
        if(r == 1) roll(TOP, BACK, BOTTOM, FRONT); // 下
        if(r == 2) roll(TOP, RIGHT, BOTTOM, LEFT); // 左
        if(r == 3) roll(TOP, FRONT, BOTTOM, BACK); // 上
    }
    void roll(int a, int b, int c, int d){ // a, b, c, d -> b, c, d, a
        int tmp = val[a];
        val[a] = val[b]; val[b] = val[c];
        val[c] = val[d]; val[d] = tmp;
    }
};
```

```

vector<Dice> all_roll(Dice a){
    vector<Dice> dices;
    for(int i = 0; i < 6; i++){
        for(int j = 0; j < 4; j++){
            dices.push_back(a);
            a.roll_z();
        }
        if(i & 1) a.roll_x();
        else a.roll_y();
    }
    return dices;
}
```