ACM ICPC Reference

University of São Paulo

November 9, 2017

Contents

1 Start of the contest

Template

```
#include <bits/stdc++.h>
using namespace std;

#define debug(args...) fprintf(stderr,args)

typedef long long lint;
typedef pair<int, int> pii;
typedef pair<lint, lint> pll;
typedef tuple<int, int, int> tiii;

const int INF = 0x3f3f3f3f;
const lint LINF = 0x3f3f3f3f3f3f3f3fll;

int main() {

    return 0;
}
```

Makefile

```
run:
	g++ $p.cpp -Wall -Wshadow -O2 -std=gnu++0x -DHOME -g -o $p
	for f in $p.in*; do \
		echo File $$f... ;\
		time ./$p < $$f | diff -sNywbB - $p.out$${f#$p.in} ;\
	done

clean:
	rm *~ $p
```

Emacs config

```
(setq c-basic-offset 4)
(global-linum-mode 1)
(electric-indent-mode 1)
(global-hl-line-mode 1)
```

2 Graphs

Graph structures

```
/* A directed graph */
struct Graph {
    int V;
    vector<vector<int>> adj;

    Graph(int _V) : V(_V) {
        adj.resize(V);
    }

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

template<class WTYPE> struct WeightedGraph {
    int V;
    vector<vector<pair<int, WTYPE>>> adj;

    WeightedGraph(int _V) : V(_V) { adj.resize(V); }

    void add_edge(int u, int v, WTYPE w) {
        adj[u].push_back(make_pair(v, w));
    }
};

/* Data structure to represent a network flow graph */
template<class FTYPE> struct FlowGraph {
    struct Edge {
        int v;
        FTYPE cap;
        Edge(int _v, FTYPE _cap) : v(_v), cap(_cap) {}
    };

    int V;
    vector<Edge> edges;
    vector<vector<int>> adj;

    FlowGraph(int _V) : V(_V) { adj.resize(V); }

    void add_edge(int u, int v, FTYPE cap) {
        adj[u].push_back(edges.size());
        edges.push_back(Edge(v, cap));
        adj[v].push_back(edges.size());
        edges.push_back(Edge(u, 0));
    }
};
```

```
/* Data structure to represent a network flow graph with costs */
template<class FTYPE, class CTYPE> struct CostFlowGraph {
    struct Edge {
        int v;
        FTYPE cap;
        CTYPE cst;
        Edge(int _v, FTYPE _cap, CTYPE _cst) : v(_v), cap(_cap), cst(_cst) {}
    };

    int V;
    vector<Edge> edges;
    vector<vector<int>> adj;

    CostFlowGraph(int _V) : V(_V) { adj.resize(V); }

    void add_edge(int u, int v, FTYPE cap, CTYPE cst) {
        adj[u].push_back(edges.size());
        edges.push_back(Edge(v, cap, cst));
        adj[v].push_back(edges.size());
        edges.push_back(Edge(u, 0, -cst));
    }
};
```

Eulerian Path

```
// Eulerian path/circuit

int mat[MAXN][MAXN]; // matriz de adjacencias
vector<int> adj[MAXN]; // lista de adjacencias
int ptr[MAXN];

vector<int> _path; // guarda o caminho

void find_path(int v) {
    for (int &a=ptr[v]; a<adj[v].size(); ++a) {
        int nxt = adj[v][a];
        if (mat[v][nxt]) {
            mat[v][nxt]--; mat[nxt][v]--;
            find_path(nxt);
            break;
        }
    }
    _path.push_back(v);
}

vector<int> eulerian_path(int s) {
    _path.clear();
    memset(ptr,0,sizeof(ptr));
    find_path(s);
    reverse(_path.begin(),_path.end());
    return _path;
}
```

Bridge and articulation point

```
struct BridgeAP {
    vector<int> ind, low;
    const Graph &g;

    set<pii> bridges;
    vector<bool> aps;

    BridgeAP(const Graph &_g) : g(_g) {
        ind.resize(g.V, -1);
        low.resize(g.V);
        aps.resize(g.V);
        for (int v = 0; v < g.V; v++)
            if (ind[v] == -1)
                dfs(v, -1);
    }

    void dfs(int v, int prv) {
        static int cnt = 0;
        ind[v] = low[v] = cnt++;
        int q = 0;
        bool flag = 0;
        for (int nxt: g.adj[v]) {
            if (ind[nxt] == -1) {
                ++q;
                dfs(nxt, v);
                low[v] = min(low[v], low[nxt]);
                if (low[nxt] >= ind[v]) flag = 1;
                if (low[nxt] == ind[nxt]) {
                    pii bridge(v, nxt);
                    if (v > nxt) swap(bridge.first, bridge.second);
                    bridges.insert(bridge);
                }
            }
            else if (nxt != prv) low[v] = min(low[v], ind[nxt]);
        }
        if (prv == -1) {
            if (q > 1) aps[v] = 1;
        }
        else if (flag) aps[v] = 1;
    }

    bool is_bridge(int u, int v) const {
        if (u > v) swap(u, v);
        return bridges.find(pii(u, v)) != bridges.end();
    }

    bool is_ap(int v) const { return aps[v]; }
};
```

Biconnected componentes

```
struct BiconnectedComponents {
    vector<vector<int>> components;

    const Graph &g;
    vector<int> ind, low;
    stack<pii> edges;
    vector<int> last;

    BiconnectedComponents(const Graph &_g) : g(_g) {
        ind.resize(g.V, -1), low.resize(g.V), last.resize(g.V, -1);
        for (int v = 0; v < g.V; v++) {
            if (g.adj[v].empty()) components.push_back({v});
            else if (ind[v] == -1) {
                dfs(v, -1);
                proc_component(-1, -1);
            }
        }
    }

    void dfs(int v, int prv) {
        static int cnt = 0;
        ind[v] = low[v] = cnt++;
        int q = 0;
        for (int nxt: g.adj[v]) {
            if (ind[nxt] == -1) {
                edges.push(pii(v, nxt));
                ++q;
                dfs(nxt, v);
                low[v] = min(low[v], low[nxt]);
                if (prv == -1 && q > 1)
                    proc_component(v, nxt);
                else if (prv != -1 && low[nxt] >= ind[v])
                    proc_component(v, nxt);
            }
            else if (nxt != prv && ind[nxt] < low[v]) {
                edges.push(pii(v, nxt));
                low[v] = ind[nxt];
            }
        }
    }

    void proc_component(int u, int v) {
        vector<int> component;
        while (!edges.empty()) {
            int uu, vv;
            tie(uu, vv) = edges.top();
            edges.pop();
            if (last[uu] != (int)components.size()) {
                last[uu] = components.size();
                component.push_back(uu);
            }
            if (last[vv] != (int)components.size()) {
                last[vv] = components.size();
                component.push_back(vv);
            }
            if (uu == u && vv == v) break;
        }
        components.push_back(component);
    }
};
```

Topological Sort

```
/* Sorts vertices in topological order */
struct TopSort {
    vector<int> ord;

    const Graph &g;
    vector<int> indeg;

    TopSort(const Graph &_g) : g(_g) {
        indeg.resize(g.V);
        for (int u = 0; u < g.V; u++)
            for (auto v: g.adj[u])
                indeg[v]++;
        for (int v = 0; v < g.V; v++)
            if (!indeg[v])
                ord.push_back(v);
        for (int i = 0; i < (int)ord.size(); i++) {
            int u = ord[i];
            for (auto v: g.adj[u])
                if (!--indeg[v])
                    ord.push_back(v);
        }
    }
};
```

Strongly Connected Components

```
/* Finds all SCCs in a directed graph */
struct Tarjan {
    vector<int> cmp_id;      // component of each vertex
    vector<vector<int>> cmp; // list of each component

    const Graph& g;
    vector<int> ind, low;
    vector<bool> in_stack;
    stack<int> st;
    int pre;

    Tarjan(const Graph &_g) : g(_g) {
        cmp_id.resize(g.V);
        ind.resize(g.V, -1);
```

```
        low.resize(g.V);
        in_stack.resize(g.V);
        pre = 0;
        for (int v = 0; v < g.V; v++) {
            if (ind[v] == -1)
                dfs(v);
        }
    }

    void dfs(int v) {
        ind[v] = low[v] = pre++;
        st.push(v);
        in_stack[v] = 1;
        for (int nxt: g.adj[v]) {
            if (ind[nxt] == -1) {
                dfs(nxt);
                low[v] = min(low[v], low[nxt]);
            }
            else if (in_stack[nxt]) {
                low[v] = min(low[v], ind[nxt]);
            }
        }
        if (ind[v] == low[v]) {
            vector<int> component;
            int vv;
            do {
                vv = st.top();
                st.pop();
                in_stack[vv] = 0;
                cmp_id[vv] = cmp.size();
                component.push_back(vv);
            } while (vv != v);
            cmp.push_back(component);
        }
    }

    Graph getContractedGraph() {
        set<pii> seen;
        Graph cg(cmp.size());
        for (int u = 0; u < g.V; u++) {
            for (int v: g.adj[u]) {
                if (cmp_id[u] != cmp_id[v]) {
                    if (seen.find(pii(cmp_id[u], cmp_id[v])) == seen.end()) {
                        seen.insert(pii(cmp_id[u], cmp_id[v]));
                        cg.add_edge(cmp_id[u], cmp_id[v]);
                    }
                }
            }
        }
        return cg;
    }
};
```

2-Sat

```
/* Finds one solution for a 2-sat instance or informs there isn't one */
struct SatSolver {
    bool solvable;
    vector<int> value; // 0 - false, 1 - true

    const Graph &g;

    SatSolver(const Graph &_g) : g(_g) {
        value.resize(g.V);
        Tarjan tarjan(g);
        for (int v = 0; v < g.V; v += 2) {
            if (tarjan.cmp_id[v] == tarjan.cmp_id[v + 1]) {
                solvable = false;
                return;
            }
        }
        solvable = true;
        Graph gc = tarjan.getContractedGraph();
        TopSort ts(gc);
        vector<int> _value(gc.V, -1);
        for (int c: ts.ord) {
            if (_value[c] == -1) {
                _value[c] = 0;
                for (int v: tarjan.cmp[c]) {
                    value[v] = 0;
                    value[v ^ 1] = 1;
                    _value[tarjan.cmp_id[v ^ 1]] = 1;
                }
            }
        }
    }
};
```

Maximum matching

```
// Edmonds' Blossom Algorithm O(N^3)
// Finds maximum matching in generic graphs

const int MAXN = ;      // maximo numero de vertices

int n;                  // numero de vertices
vector<int> adj[MAXN]; // lista de adj
int match[MAXN];        // match[i] eh o par de i. -1 se nao tem par
int p[MAXN], base[MAXN] , q[MAXN];
bool used[MAXN], blossom[MAXN];

int lca ( int a, int b ) {
    bool used [MAXN] = {0};
    while (1) {
        a = base[a];
```

```
        used[a] = true;
        if (match[a] == -1)  break;
        a = p[match[a]];
    }
    while (1) {
        b = base[b];
        if (used[b])  return b;
        b = p[match[b]];
    }
}

void mark_path ( int v, int b, int children ) {
    while (base[v] != b) {
        blossom[base[v]] = blossom[base[match[v]]] = true;
        p[v] = children;
        children = match[v];
        v = p[match[v]];
    }
}

int find_path ( int root ) {
    memset(used, 0, sizeof used);
    memset(p, -1, sizeof p);
    for (int i = 0 ; i < n ; ++i)
        base[i] = i;
    used[root] = true;
    int qh = 0 , qt = 0;
    q[qt++] = root;
    while (qh < qt) {
        int v = q[qh++];
        for (size_t i = 0; i < adj[v].size(); ++i) {
            int to = adj[v][i];
            if (base[v] == base[to] || match[v] == to)  continue;
            if (to == root || match[to] != -1 && p[match[to]] != -1) {
                int curbase = lca (v, to);
                memset(blossom, 0, sizeof blossom);
                mark_path(v, curbase, to);
                mark_path(to, curbase, v);
                for (int i = 0; i < n; ++i) {
                    if (blossom[base[i]]) {
                        base[i] = curbase;
                        if (!used[i]) {
                            used[i] = true;
                            q[qt++] = i;
                        }
                    }
                }
            }
            else if (p[to] == -1) {
                p[to] = v;
                if (match[to] == -1)
                    return to;
                to = match[to];
                used[to] = true;
                q[qt++] = to;
            }
        }
    }
    return -1;
}

int main() {
    // ler grafo
    memset ( match, -1 , sizeof match );
    // otimizacao: comeca com um matching parcial guloso
    for (int i = 0; i < n; ++i) {
        if (match[i] == -1) {
            for (size_t j = 0; j < adj[i].size(); ++j) {
                if (match[adj[i][j]] == -1) {
                    match[adj[i][j]] = i;
                    match[i] = adj[i][j];
                    break;
                }
            }
        }
    }
    for (int i = 0; i < n; ++i) {
        if (match[i] == -1) {
            int v = find_path(i);
            while (v != -1) {
                int pv = p[v], ppv = match[pv];
                match[v] = pv, match[pv] = v;
                v = ppv;
            }
        }
    }
    // ...
}
```

Max flow

```
/*
  Dinic maxflow algorithm - O(EV^2)
  O(EV^(2/3)) for unit capacities
  O(EV^(1/2)) for bipartite graphs
*/
template<class FTYPE> struct Dinic {
    vector<int> ptr, dist;
    FlowGraph<FTYPE> &g;

    Dinic(FlowGraph<FTYPE> &_g) : g(_g) {
        ptr.resize(g.V);
        dist.resize(g.V);
    }

    bool bfs(int s, int t) {
        fill(dist.begin(), dist.end(), -1);
```

```
        dist[s] = 0;
        queue<int> q({s});
        while (!q.empty()) {
            int v = q.front();
            if (dist[v] == dist[t]) break;
            q.pop();
            for (int i: g.adj[v]) {
                int nxt = g.edges[i].v;
                if (dist[nxt] == -1 && g.edges[i].cap) {
                    dist[nxt] = dist[v] + 1;
                    q.push(nxt);
                }
            }
        }
        return dist[t] != -1;
    }

    FTYPE dfs(int v, int t, FTYPE flow) {
        if (v == t) return flow;
        for (int &p = ptr[v]; p < (int)g.adj[v].size(); p++) {
            int i = g.adj[v][p];
            int nxt = g.edges[i].v;
            if (dist[nxt] == dist[v] + 1 && g.edges[i].cap) {
                FTYPE got = dfs(nxt, t, min(flow, g.edges[i].cap));
                if (got) {
                    g.edges[i].cap -= got;
                    g.edges[i^1].cap += got;
                    return got;
                }
            }
        }
        return 0;
    }

    FTYPE max_flow(int s, int t) {
        FTYPE ret = 0;
        while (bfs(s, t)) {
            fill(ptr.begin(), ptr.end(), 0);
            while (FTYPE got = dfs(s, t, numeric_limits<FTYPE>::max()))
                ret += got;
        }
        return ret;
    }
};

/*
   Push-Relabel maxflow algorithm: FIFO rule - O(V^3)
   Implements Gap heuristic.
*/
template<class FTYPE> struct PushRelabelFIFOGap {
    FlowGraph<FTYPE> &g;

    PushRelabelFIFOGap(FlowGraph<FTYPE> &_g) : g(_g) {}

    FTYPE max_flow(int s, int t) {
        vector<int> ptr(g.V, 0), h(g.V, 0), hc(2*g.V, 0);
        vector<FTYPE> e(g.V, 0);
        h[s] = g.V;
        hc[g.V] = 1; hc[0] = g.V - 1;
        queue<int> q;
        for (int i: g.adj[s]) {
            int w = g.edges[i].v;
            if (!g.edges[i].cap) continue;
            if (!e[w] && w != t) q.push(w);
            e[w] += g.edges[i].cap;
            e[s] -= g.edges[i].cap;
            g.edges[i^1].cap = g.edges[i].cap;
            g.edges[i].cap = 0;
        }

        while (!q.empty()) {
            int v = q.front();
            for (int &p = ptr[v]; p < (int)g.adj[v].size(); p++) {
                int i = g.adj[v][p];
                int w = g.edges[i].v;
                if (h[w] < h[v] && g.edges[i].cap) {
                    FTYPE f = min(g.edges[i].cap, e[v]);
                    g.edges[i].cap -= f;
                    g.edges[i^1].cap += f;
                    if (!e[w] && w != t) q.push(w);
                    e[w] += f;
                    e[v] -= f;
                    if (e[v] == 0) break;
                }
            }
            if (e[v]) {
                int cur_h = h[v];
                if (hc[cur_h] == 1 && 0 < cur_h && cur_h < g.V) {
                    for (int u = 0; u < g.V; u++) {
                        if (h[u] >= g.V || h[u] < cur_h) continue;
                        hc[h[u]]--;
                        h[u] = g.V + 1;
                        hc[h[u]]++;
                        ptr[u] = 0;
                    }
                }
                else {
                    ptr[v] = 0;
                    hc[h[v]]--;
                    h[v]++;
                    hc[h[v]]++;
                }
            }
            else q.pop();
        }

        return e[t];
    }
};
```

Min cost max flow

```
/*
   Minimum cost maximum flow: Successive Shortest Path implementation
   O(E*V + U*V^2), where U is the maximum flow
*/
template<class FTYPE, class CTYPE> struct MinCostMaxFlowSSP {
    static const CTYPE CINF = numeric_limits<CTYPE>::max() / 2;
    static const FTYPE FINF = numeric_limits<FTYPE>::max() / 2;

    CostFlowGraph<FTYPE, CTYPE> &g;
    vector<CTYPE> d, p;

    MinCostMaxFlowSSP(CostFlowGraph<FTYPE, CTYPE> &_g) : g(_g) {
        d.resize(g.V);
        p.resize(g.V);
    }

    void init_p(int s) {
        fill(d.begin(), d.end(), CINF);
        d[s] = 0;
        queue<int> q({s});
        vector<bool> queued(g.V, 0);
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            queued[v] = 0;
            for (int i: g.adj[v]) {
                int w = g.edges[i].v;
                if (!g.edges[i].cap) continue;
                if (d[w] > d[v] + g.edges[i].cst) {
                    d[w] = d[v] + g.edges[i].cst;
                    if (!queued[w]) {
                        q.push(w);
                        queued[w] = 1;
                    }
                }
            }
        }
        for (int v = 0; v < g.V; v++)
            p[v] = d[v];
    }

    pair<FTYPE, CTYPE> augment(int s, int t) {
        fill(d.begin(), d.end(), CINF);
        d[s] = 0;
        vector<bool> done(g.V, 0);
        vector<int> prv(g.V, -1);
        vector<FTYPE> flow(g.V);
        flow[s] = FINF;
        while (1) {
            int v;
            CTYPE dist_v = CINF;
            for (int w = 0; w < g.V; w++) {
                if (!done[w] && d[w] < dist_v) {
                    v = w;
                    dist_v = d[w];
                }
            }
            if (dist_v == CINF) break;
            for (int i: g.adj[v]) {
                int w = g.edges[i].v;
                FTYPE cap = g.edges[i].cap;
                CTYPE cst = g.edges[i].cst;
                if (cap && d[w] > d[v] + cst + p[v] - p[w]) {
                    d[w] = d[v] + cst + p[v] - p[w];
                    prv[w] = i;
                    flow[w] = min(flow[v], cap);
                }
            }
            done[v] = 1;
        }
        if (d[t] == CINF) return make_pair(-1, 0);
        for (int v = 0; v < g.V; v++) p[v] += d[v];

        CTYPE flow_cost = 0;
        for (int v = t; v != s; v = g.edges[prv[v]^1].v) {
            g.edges[prv[v]].cap -= flow[t];
            g.edges[prv[v]^1].cap += flow[t];
            flow_cost += flow[t] * g.edges[prv[v]].cst;
        }
        return make_pair(flow[t], flow_cost);
    }

    // Returns a pair (max flow, min cost)
    pair<FTYPE, CTYPE> mcmf(int s, int t) {
        pair<FTYPE, CTYPE> ret(0, 0), aug;
        init_p(s);
        while ((aug = augment(s, t)).first != -1) {
            ret.first += aug.first;
            ret.second += aug.second;
        }
        return ret;
    }
};
```

Gomory-Hu tree

```
/*
   Computes a Gomory-Hu tree of a given undirected graph.
   Any minimum cut (S, T) in the tree is also a minimum cut in the original
   graph.
*/
template<class FTYPE, template<class> class MF>
WeightedGraph<FTYPE> GomoryHuTree(const FlowGraph<FTYPE> &g) {
    vector<int> parent(g.V, 0), depth(g.V, 0);
    vector<FTYPE> cap(g.V);
    for (int s = 1; s < g.V; s++) {
        int t = parent[s];
        FlowGraph<FTYPE> g2 = g;
        MF<FTYPE> mf(g2);
        int fst = mf.max_flow(s, t);
        cap[s] = fst;
        vector<bool> in_cut(g.V, 0);
        queue<int> q({s});
        in_cut[s] = 1;
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int i: g2.adj[v]) {
                int nxt = g2.edges[i].v;
                if (g2.edges[i].cap > 0 && !in_cut[nxt]) {
                    in_cut[nxt] = 1;
                    q.push(nxt);
                }
            }
        }
        for (int v = 0; v < g.V; v++)
            if (v != s && in_cut[v] && parent[v] == t)
                parent[v] = s;
        if (in_cut[parent[t]]) {
            parent[s] = parent[t];
            parent[t] = s;
            cap[s] = cap[t];
            cap[t] = fst;
        }
    }
    WeightedGraph<FTYPE> tree(g.V);
    for (int v = 1; v < g.V; v++) {
        tree.add_edge(v, parent[v], cap[v]);
        tree.add_edge(parent[v], v, cap[v]);
    }
    return tree;
};
```

Global min cut

```
// Global mincut - O(VElogV) or O(V^3)
template<class FTYPE> struct StoerWagnerMinCut {
    FlowGraph<FTYPE> g;

    StoerWagnerMinCut(FlowGraph<FTYPE> _g) : g(_g) {}

    FlowGraph<FTYPE> contract(int s, int t) {
        FlowGraph<FTYPE> g2(g.V - 1);
        vector<int> idx(g.V);
        for (int v = 0; v < g.V; v++) idx[v] = v;
        swap(idx[t], idx[g.V - 1]);
        for (int v = 0; v < g.V; v++) {
            if (v == s || v == t) continue;
            int cap = 0;
            for (int i: g.adj[v]) {
                int w = g.edges[i].v, c = g.edges[i].cap;
                if (w == s || w == t) cap += c;
                else if (v < w) g2.add_edge(idx[v], idx[w], c);
            }
            if (cap > 0) g2.add_edge(idx[v], idx[s], cap);
        }
        return g2;
    }

    FTYPE mincut() {
        FTYPE ret = numeric_limits<FTYPE>::max();
        while (g.V > 1) {
            vector<bool> seen(g.V);
            set<pair<FTYPE, int>, greater<pair<FTYPE, int>>> pq({{0, 0}});
            vector<FTYPE> c(g.V);
            int s = -1, t = -1;
            while (!pq.empty()) {
                int v = pq.begin()->second;
                pq.erase(pq.begin());
                seen[v] = 1;
                s = t;
                t = v;
                for (int idx: g.adj[v]) {
                    int w = g.edges[idx].v;
                    if (seen[w]) continue;
                    pq.erase(make_pair(c[w], w));
                    c[w] += g.edges[idx].cap;
                    pq.insert(make_pair(c[w], w));
                }
            }
            ret = min(ret, c[t]);
            g = contract(s, t);
        }
        return ret;
    }
};
```

Heavy-Light decomposition

```
// Max segment tree

int tree[4*MAXN];
int tree_val[MAXN];
int value[MAXN]; // initial value of node [i]

void tree_build(int i, int l, int r) {
    if (l == r) {
        tree[i] = tree_val[l];
        return;
    }
    int L = 2*i + 1, R = 2*i + 2, mid = (l + r)/2;
    tree_build(L, l, mid); tree_build(R, mid + 1, r);
    tree[i] = max(tree[L], tree[R]);
}

void tree_update(int i, int l, int r, int pos, int val) {
    if (l > pos || r < pos) return;
    if (l == r) {
        tree[i] = val;
        return;
    }
    int L = 2*i + 1, R = 2*i + 2, mid = (l + r)/2;
    tree_update(L, l, mid, pos, val); tree_update(R, mid + 1, r, pos, val);
    tree[i] = max(tree[L], tree[R]);
}

int tree_query(int i, int l, int r, int ql, int qr) {
    if (ql <= l && r <= qr) return tree[i];
    if (ql > r || qr < l) return -INF;
    int L = 2*i + 1, R = 2*i + 2, mid = (l + r)/2;
    return max(tree_query(L, l, mid, ql, qr), tree_query(R, mid + 1, r, ql, qr));
}

int hld_dfs(int v) {
    int hi = 0, ret = 1;
    for (int i = 0; i < (int)adj[v].size(); i++) {
        int nxt = adj[v][i].first, cst = adj[v][i].second;
        if (nxt == parent[v]) continue;
        parent[nxt] = v;
        value[nxt] = cst;
        depth[nxt] = depth[v] + 1;
        int got = hld_dfs(nxt);
        if (got > hi) {
            hi = got;
            heavy[v] = nxt;
        }
        ret += got;
    }
    return ret;
}

void hld_preprocess(int s, int n) {
    memset(heavy, -1, sizeof(heavy));
    parent[s] = -1; depth[s] = 0; value[s] = -INF;
    hld_dfs(s);
    int cur = 0;
    // 1-indexed
    for (int v = 1; v <= n; v++) {
        if (parent[v] == -1 || heavy[parent[v]] != v) {
            for (int j = v; j != -1; j = heavy[j]) {
                root[j] = v, hld_pos[j] = cur++;
                tree_val[hld_pos[j]] = value[j];
            }
        }
    }
    tree_build(0, 0, n-1);
}

int hld_query(int u, int v, int n) {
    int ret = -INF;
    for (; root[u] != root[v]; v = parent[root[v]]) {
        if (depth[root[u]] > depth[root[v]]) swap(u, v);
        ret = max(ret, tree_query(0, 0, n-1, hld_pos[root[v]], hld_pos[v]));
    }
    if (depth[u] > depth[v]) swap(u, v);
    if (u != v) ret = max(ret, tree_query(0, 0, n - 1, hld_pos[u] + 1, hld_pos[v]));
    return ret;
}

inline void hld_update(int v, int val, int n) {
    tree_update(0, 0, n - 1, hld_pos[v], val);
}
```

HLD with no data structure

```
// Finds heavy paths without building a data structure over them.
struct HeavyLightSimple {
    const Graph &g;
    vector<int> head;   // Head of the heavy path containing i
    vector<int> parent; // Parent of node i

    vector<int> depth, heavy;
    int root;

    HeavyLightSimple(const Graph &_g, int _root) : g(_g), root(_root) {
        head.resize(g.V), parent.resize(g.V), depth.resize(g.V), heavy.resize(g.V, -1);
        parent[root] = -1, depth[root] = 0;
        dfs(root);
        for (int v = 0; v < g.V; v++)
            if (parent[v] == -1 || heavy[parent[v]] != v)
                for (int w = v; w != -1; w = heavy[w])
                    head[w] = v;
    }

    int dfs(int v) {
        int hi = 0, ret = 1;
        for (int nxt: g.adj[v]) {
```

```
            if (nxt == parent[v]) continue;
            parent[nxt] = v;
            depth[nxt] = depth[v] + 1;
            int got = dfs(nxt);
            if (got > hi) {
                hi = got;
                heavy[v] = nxt;
            }
            ret += got;
        }
        return ret;
    }
};
```

Dominator tree

```
// O(ElogV) implementation of a dominator tree
struct DominatorTree {
    const Graph &g;

    Graph tree; // The dominator tree
    vector<int> dfs_l, dfs_r;

    // Auxiliary data
    Graph rg;
    vector<vector<int>> bucket;
    vector<int> idom, sdom, prv, pre;
    vector<int> ancestor, label, preorder;

    DominatorTree(const Graph &_g, int s) : g(_g) {
        rg = tree = Graph(g.V);
        idom.resize(g.V), sdom.resize(g.V, -1);
        prv.resize(g.V), pre.resize(g.V, -1), bucket.resize(g.V);
        ancestor.resize(g.V, -1), label.resize(g.V);
        dfs_l.resize(g.V), dfs_r.resize(g.V);
        dfs(s);
        if (preorder.size() == 1) return;
        for (size_t i = preorder.size() - 1; i >= 1; i--) {
            int w = preorder[i];
            for (int v: rg.adj[w]) {
                int u = eval(v);
                if (pre[sdom[u]] < pre[sdom[w]]) sdom[w] = sdom[u];
            }
            bucket[sdom[w]].push_back(w);
            link(prv[w], w);
            for (int v: bucket[prv[w]]) {
                int u = eval(v);
                idom[v] = (u == v) ? sdom[v] : u;
            }
            bucket[prv[w]].clear();
        }
        for (size_t i = 1; i < preorder.size(); i++) {
            int w = preorder[i];
            if (idom[w] != sdom[w]) idom[w] = idom[idom[w]];
            tree.add_edge(idom[w], w);
        }
        idom[s] = sdom[s] = -1;
        dfs2(s);
    }

    void dfs(int v) {
        static int t = 0;
        pre[v] = ++t;
        sdom[v] = label[v] = v;
        preorder.push_back(v);
        for (int nxt: g.adj[v]) {
            if (sdom[nxt] == -1) {
                prv[nxt] = v;
                dfs(nxt);
            }
            rg.add_edge(nxt, v);
        }
    }

    int eval(int v) {
        if (ancestor[v] == -1) return v;
        if (ancestor[ancestor[v]] == -1) return label[v];
        int u = eval(ancestor[v]);
        if (pre[sdom[u]] < pre[sdom[label[v]]]) label[v] = u;
        ancestor[v] = ancestor[u];
        return label[v];
    }

    inline void link(int u, int v) {
        ancestor[v] = u;
    }

    void dfs2(int v) {
        static int t = 0;
        dfs_l[v] = t++;
        for (int nxt: tree.adj[v]) dfs2(nxt);
        dfs_r[v] = t++;
    }

    // Whether every path from s to v passes through u
    inline bool dominates(int u, int v) {
        if (pre[v] == -1) return 1; // vacuously true
        return dfs_l[u] <= dfs_l[v] && dfs_r[v] <= dfs_r[u];
    }
};
```

3 Strings

Knuth-Morris-Pratt

```
struct KMP {
    string pattern;
    int len;
    // f[i] = the size of longest preffix that is a suffix of p[0..i-1]
    vector<int> f;

    KMP(string p) {
        pattern = p;
        len = p.size();
        f.resize(len + 2);
        f[0] = f[1] = 0;
        for (int i = 2; i <= len; i++) {
            int now = f[i - 1];
            while (1) {
                if (p[now] == p[i - 1]) {
                    f[i] = now + 1;
                    break;
                }
                if (now == 0) {
                    f[i] = 0;
                    break;
                }
                now = f[now];
            }
        }
    }

    // returns a vector of indices with the beginning of each match
    vector<int> match(string text) {
        vector<int> ret;
        int size = text.size();
        int i = 0, j = 0;
        while (j < size) {
            if (text[j] == pattern[i]) {
                i++; j++;
                if (i == len) {
                    ret.push_back(j - len);
                    i = f[i];
                }
            }
            else if (i > 0) i = f[i];
            else j++;
        }
        return ret;
    }
};
```

Z algorithm

```
// Z-algorithm, O(N)
// Builds array z such that z[i] = size of longest prefix substring
// starting at index i
vector<int> Z(string s) {
    vector<int> z(1, s.size());
    int l = 0, r = 0;
    for (int a = 1; a < (int)s.size(); ++a) {
        if (r < a) {
            l = r = a;
            while (r < (int)s.size() && s[r] == s[r-l]) ++r;
            z.push_back(r - l);
            r--;
        }
        else if (z[a - l] < r - a + 1)
            z.push_back(min<int>(z[a - l], s.size() - a));
        else {
            l = a;
            while (r < (int)s.size() && s[r] == s[r - l]) ++r;
            z.push_back(r - l);
            r--;
        }
    }
    return z;
}
```

Aho-Corasick

```
// NEEDS TESTING
struct AhoCorasick {
    const static int MAXC = 300; // alphabet size
    const static int MAXND = (int)1e4 + 10;

    struct Node {
        vector<int> matches;
        int nxt[MAXC];
        int fail;

        Node() { memset(nxt, -1, sizeof(nxt)); }
    };

    vector<Node> nodes;
    vector<string> words;
    int cur_node;
    bool built;
    int fail_mem[MAXND][MAXC];

    void add(int node, int i, int idx) {
        int c = words[idx][i];
        if (i == (int)words[idx].size()) {
            nodes[node].matches.push_back(idx);
```

```
        return;
    }
    if (nodes[node].nxt[c] == -1) {
        nodes[node].nxt[c] = nodes.size();
        nodes.push_back(Node());
    }
    add(nodes[node].nxt[c], i + 1, idx);
}

void add_word(string word) {
    words.push_back(word);
    add(0, 0, words.size() - 1);
}

void build() {
    built = 1;
    queue<int> q;
    nodes[0].fail = 0;
    for (int c = 0; c < MAXC; c++) {
        int nxt = nodes[0].nxt[c];
        if (nxt != -1) {
            nodes[nxt].fail = 0;
            q.push(nxt);
        }
    }
    while (!q.empty()) {
        int cur = q.front();
        q.pop();
        for (int c = 0; c < MAXC; c++) {
            int nxt = nodes[cur].nxt[c];
            if (nxt == -1) continue;
            int fail = nodes[cur].fail;
            while (fail && nodes[fail].nxt[c] == -1) fail = nodes[fail].fail;
            if (nodes[fail].nxt[c] == -1) nodes[nxt].fail = 0;
            else nodes[nxt].fail = nodes[fail].nxt[c];
            for (int match: nodes[nodes[nxt].fail].matches)
                nodes[nxt].matches.push_back(match);
            q.push(nxt);
        }
    }
}

void walk(int c) {
    assert(built);
    int prv_node = cur_node;
    if (fail_m[cur_node][c] != -1) cur_node = fail_m[cur_node][c];
    else {
        if (nodes[cur_node].nxt[c] != -1)
            cur_node = nodes[cur_node].nxt[c];
        else {
            int fail = nodes[cur_node].fail;
            while (fail && nodes[fail].nxt[c] == -1) fail = nodes[fail].fail;
            if (nodes[fail].nxt[c] == -1) cur_node = 0;
            else cur_node = nodes[fail].nxt[c];
        }
        fail_m[prv_node][c] = cur_node;
    }
}

vector<int> get_matches() { return nodes[cur_node].matches; }

void reset() { cur_node = 0; }

void clear() {
    built = 0;
    nodes.clear();
    words.clear();
    nodes.push_back(Node());
    memset(fail_m, -1, sizeof(fail_m));
    reset();
}

AhoCorasick() { clear(); }
};
```

Hash

```
struct Hash {
    static const int P1 = 31, P2 = 37, MOD = (int)1e9 + 7;
    lint h1, h2;
    Hash(lint a = 0, lint b = 0) { h1 = a; h2 = b; }
    Hash(const string &s) {
        h1 = 0, h2 = 0;
        for (char c: s) {
            h1 = (P1 * h1 + c) % MOD;
            h2 = (P2 * h2 + c) % MOD;
        }
    }
    void append(char c) {
        h1 = (P1 * h1 + c) % MOD;
        h2 = (P2 * h2 + c) % MOD;
    }
    bool operator==(const Hash &that) const { return h1 == that.h1 && h2 == that.h2; }
    bool operator!=(const Hash &that) const { return h1 != that.h1 || h2 != that.h2; }
    Hash operator*(const Hash &that) const {
        return Hash((h1 * that.h1) % MOD, (h2 * that.h2) % MOD);
    }
    Hash operator-(const Hash &that) const {
        return Hash((h1 - that.h1 + MOD) % MOD, (h2 - that.h2 + MOD) % MOD);
    }
    bool operator<(const Hash &that) const {
        if (h1 == that.h1) return h2 < that.h2;
        return h1 < that.h1;
    }
};

struct HashArray {
```

```
    vector<Hash> pot;
    vector<Hash> array;

    HashArray(string &s) {
        pot.resize(s.size());
        pot[0] = Hash(1,1);
        Hash acc;
        for (size_t i = 0; i < s.size(); i++) {
            acc.append(s[i]);
            array.push_back(acc);
            if (i > 0) pot[i] = pot[i - 1] * Hash(Hash::P1, Hash::P2);
        }
    }

    inline Hash get_hash(int l, int r) {
        if (l == 0) return array[r];
        return array[r] - array[l - 1] * pot[r - l + 1];
    }
};
```

4 Data Structures

Convex trick optimization

```
// Convex Hull Optimization (constant query variant)
// Requires:
//   1. Lines are inserted in strict order of slope:
//      increasing -> max, decreasing -> min
//   2. Queries are made in increasing order of x.

template<class C_TYPE> struct ConvexHullOpt {
    struct Line {
        C_TYPE a, b; // a + bx
        double end_l;
        C_TYPE get(C_TYPE x) { return a + b*x; }
        Line(){}
        Line(C_TYPE aa, C_TYPE bb) { a = aa; b = bb; end_l = -LINF; }
    };

    vector<Line> deq;
    int deq_l;

    double cross(const Line &r, const Line &s) {
        return double(s.a - r.a) / (r.b - s.b);
    }

    ConvexHullOpt() { clear(); }

    void add_line(C_TYPE a, C_TYPE b) {
        Line newline(a, b);
        while (deq_l < (int)deq.size() &&
               cross(newline, deq.back()) < deq.back().end_l)
            deq.pop_back();
        if (deq_l < (int)deq.size()) newline.end_l = cross(newline, deq.back());
        deq.push_back(newline);
    }

    C_TYPE get(C_TYPE x) {
        if (deq_l >= (int)deq.size()) {
            // can't query with no lines in structure =P
            abort();
        }
        while (deq_l + 1 < (int)deq.size() && deq[deq_l + 1].end_l <= x)
            deq_l++;
        return deq[deq_l].get(x);
    }

    void clear() {
        deq.clear();
        deq_l = 0;
    }
};

// Convex Hull Optimization (general case)
// O(logn) for insertion and query
// Test: SPOJ GOODG

template< class C_TYPE, class Compare = less<C_TYPE> > struct ConvexHullOpt {
    struct Line {
        C_TYPE a, b; // a + bx
        double end_l, end_r;
        Line(){}
        Line(C_TYPE aa, C_TYPE bb) { a = aa; b = bb; end_l = -1e80, end_r = 1e80; }
        inline C_TYPE get(C_TYPE x) const { return a + b*x; }
    };

    struct by_slope {
        bool operator()(const Line &a, const Line &b) const {
            return Compare()(a.b, b.b);
        }
    };

    struct by_end {
        bool operator()(const Line &a, const Line &b) const {
            return a.end_r < b.end_r;
        }
    };

    inline double cross(const Line &a, const Line &b) {
        return double(b.a - a.a) / (a.b - b.b);
    }

    set<Line, by_slope> set_slope;
    set<Line, by_end> set_end;

    void add_line(C_TYPE a, C_TYPE b) {
        Line newline(a, b);
```

```
        auto itr = set_slope.lower_bound(newline);
        auto itr2 = itr == set_slope.end() ?
            set_end.end() : set_end.lower_bound(*itr);
        if (itr != set_slope.end()) {
            if (cross(*itr, newline) < itr->end_l) return;
            while (itr != set_slope.end() && cross(*itr, newline) > itr->end_r) {
                itr = set_slope.erase(itr);
                itr2 = set_end.erase(itr2);
            }
            if (itr != set_slope.end()) {
                double x = cross(*itr, newline);
                Line tmp = *itr;
                newline.end_r = tmp.end_l = x;
                itr = set_slope.erase(itr);
                itr = set_slope.insert(itr, tmp);
                itr2 = set_end.erase(itr2);
                itr2 = set_end.insert(itr2, tmp);
            }
        }
        auto itl = itr;
        auto itl2 = itr2;
        while (itl != set_slope.begin()) {
            itl--, itl2--;
            double x = cross(*itl, newline);
            if (x > itl->end_l) {
                Line tmp = *itl;
                newline.end_l = tmp.end_r = x;
                itl = set_slope.erase(itl);
                itl = set_slope.insert(itl, tmp);
                itl2 = set_end.erase(itl2);
                itl2 = set_end.insert(itl2, tmp);
                break;
            }
            itl = set_slope.erase(itl);
            itl2 = set_end.erase(itl2);
        }
        set_slope.insert(itr, newline);
        set_end.insert(itr2, newline);
    }

    C_TYPE get(C_TYPE x) {
        if (set_end.empty()) abort(); // structure has no lines
        Line dummy;
        dummy.end_r = x;
        auto it = set_end.lower_bound(dummy);
        return it->get(x);
    }
};
```

Ordered set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

// Supports *find_by_order() and order_of_key()
template<class T> using OrderedSet = tree<
    T,
    null_type,
    less<T>,
    rb_tree_tag,
    tree_order_statistics_node_update>;
```

Static RMQ

```
// Static RMQ structure.
// Builds in O(NlogN) and computes queries in O(1).
template<class T> struct StaticRMQ {
    static const int LOG = 20;
    vector<T> rmq[LOG];
    int n;

    StaticRMQ(const vector<T> &v) : n(v.size()) {
        for (int i = 0; i < LOG; i++)
            rmq[i].resize(n);
        for (int i = 0; i < n; i++)
            rmq[0][i] = v[i];
        for (int k = 1; k < LOG; k++)
            for (int i = 0; i < n; i++)
                rmq[k][i] = min(rmq[k - 1][i], rmq[k - 1][min(n - 1, i + (1 << (k - 1))
    }

    T get(int l, int r) {
        int k = 31 - __builtin_clz(r - l + 1);
        return min(rmq[k][l], rmq[k][r - (1 << k) + 1]);
    }
};
```

Time

```
tm* get_tm(int year, int month, int day, int hour = 0, int min = 0, int sec = 0) {
    tm *date = new tm();
    date->tm_year = year - 1900;
    date->tm_mon = month - 1;
    date->tm_mday = day;
    date->tm_hour = hour;
    date->tm_min = min;
    date->tm_sec = sec;
```

```
    mktime(date);
    return date;
}

// Returns the Unix timestamp for the given date interpreted as local time
int get_timestamp(int year, int month, int day, int hour = 0, int min = 0, int sec = 0) {
    tm *date = get_tm(year, month, day, hour, min, sec);
    return mktime(date);
}

// Get day of the week of given date
int day_of_week(int year, int month, int day) {
    tm *date = get_tm(year, month, day);
    return date->tm_wday;
}
```

Treap

```
// Supports insertion, deletion, querying kth-element and finding element
// Keeps duplicate elements as different nodes

template <class T> class Treap {
    struct Node {
        T val;
        int h,cnt;
        Node *l, *r;
        Node(T val2) {
            val = val2;
            h = rand();
            cnt = 1;
            l = r = NULL;
        }
    };
    Node *root;

    inline Node* newNode(T val) { return new Node(val); }

    inline void refresh(Node *node) {
        if (node == NULL) return;
        node->cnt = (node->l == NULL ? 0 : node->l->cnt) +
            (node->r == NULL ? 0 : node->r->cnt) + 1;
    }

    void _insert(Node *&node, T val) {
        if (node == NULL) {
            node = newNode(val);
            return;
        }
        if (val <= node->val) {
            _insert(node->l, val);
            if (node->l->h > node->h) {
                Node *aux = node->l;
                node->l = aux->r;
                aux->r = node;
                node = aux;
                refresh(node->r);
                refresh(node);
            }
            else refresh(node);
        }
        else {
            _insert(node->r, val);
            if (node->r->h > node->h) {
                Node *aux = node->r;
                node->r = aux->l;
                aux->l = node;
                node = aux;
                refresh(node->l);
                refresh(node);
            }
            else refresh(node);
        }
    }

    Node* merge(Node *L,Node *R) {
        if (L == NULL) return R;
        if (R == NULL) return L;
        if (L->h < R->h) {
            L->r = merge(L->r,R);
            refresh(L);
            return L;
        }
        else {
            R->l = merge(L,R->l);
            refresh(R);
            return R;
        }
    }

    // not used. splits node into two trees a(<=val) and b(>val)
    void split(T val, Node *node, Node *&a, Node *&b) {
        if (node == NULL) {
            a = b = NULL;
            return;
        }
        Node *aux;
        if (val >= node->val) {
            split(val,node->r,aux,b);
            node->r = aux;
            a = node;
            refresh(a);
        }
        else {
            split(val,node->l,a,aux);
            node->l = aux;
            b = node;
            refresh(b);
```

```
        }
    }

    // erases a single appearence of val
    void _erase(Node *&node, T val) {
        if (node == NULL) return;
        if (node->val > val) _erase(node->l, val);
        else if (node->val < val) _erase(node->r, val);
        else node = merge(node->l,node->r);
        refresh(node);
    }

    // 0-indexed (not safe if element doesnt exist)
    T _kth(Node *node,int k) {
        int ql = (node->l == NULL ? 0 : node->l->cnt);
        if (k < ql) return _kth(node->l,k);
        if (k == ql) return node->val;
        k -= ql + 1;
        return _kth(node->r,k);
    }

    // returns position (0-indexed) of element 'val' in 'node'. -1 if it doesn't exist
    int _find(Node *node, T val) {
        if (node == NULL) return -1;
        if (node->val == val) return (node->l == NULL ? 0 : node->l->cnt);
        else if (node->val > val) return _find(node->l,val);
        else {
            int pos = _find(node->r,val);
            if (pos == -1) return -1;
            return 1 + (node->l == NULL ? 0 : node->l->cnt) + pos;
        }
    }

    void _clear(Node *&node) {
        if (node == NULL) return;
        _clear(node->l); _clear(node->r);
        delete node;
        node = NULL;
    }

public:
    Treap() { root = NULL; }
    void insert(T val) { _insert(root,val); }
    T kth(int k) { return _kth(root,k); }
    int size() { return root == NULL ? 0 : root->cnt; }
    void clear() { _clear(root); }
    void erase(T val) { _erase(root,val); }
    int find(T val) { return _find(root,val); }
};
```

5 Math

Combinatorics

```
const int P = (int)1e9 + 7;

const int MAXV = ;

lint fat[MAXV], inv[MAXV], invfat[MAXV];

lint choose(int n, int k) {
    k = min(k, n - k);
    if (k < 0) return 0;
    return fat[n] * invfat[k] % P * invfat[n - k] % P;
}

lint arrange(int n, int k) {
    if (k > n) return 0;
    return fat[n] * invfat[n - k] % P;
}

lint modexp(lint b, lint e) {
    lint ret = 1, aux = b;
    while (e) {
        if (e & 1) ret = ret * aux % P;
        aux = aux * aux % P;
        e >>= 1;
    }
    return ret;
}

void precalc() {
    fat[0] = fat[1] = 1;
    invfat[0] = invfat[1] = 1;
    inv[1] = 1;
    for (int n = 2; n < MAXV; n++) {
        fat[n] = fat[n - 1] * n % P;
        inv[n] = P - P/n * inv[P%n] % P;
        invfat[n] = invfat[n - 1] * inv[n] % P;
    }
}
```

Gaussian elimination

```
const int MAXN = 110;

typedef double Number;
const Number EPS = 1e-9;

Number mat[MAXN][MAXN];
int idx[MAXN];   // row index
int pivot[MAXN]; // pivot of row i

// Solves Ax = B, where A is a neq x nvar matrix and B is mat[*][nvar]
// Returns a vector of free variables (empty if system is defined,
// or {-1} if no solution exists)
// Reduces matrix to reduced echelon form
vector<int> solve(int nvar, int neq) {
    for (int i = 0; i < neq; i++) idx[i] = i;
    int currow = 0;
    vector<int> freeVars;
    for (int col = 0; col < nvar; col++) {
        int pivotrow = -1;
        Number val = 0;
        for (int row = currow; row < neq; row++) {
            if (fabs(mat[idx[row]][col]) > val + EPS) {
                val = fabs(mat[idx[row]][col]);
                pivotrow = row;
            }
        }
        if (pivotrow == -1) { freeVars.push_back(col); continue; }
        swap(idx[currow], idx[pivotrow]);
        pivot[currow] = col;
        for (int c = 0; c <= nvar; c++) {
            if (c == col) continue;
            mat[idx[currow]][c] = mat[idx[currow]][c] / mat[idx[currow]][col];
        }
        mat[idx[currow]][col] = 1;
        for (int row = 0; row < neq; row++) {
            if (row == currow) continue;
            Number k = mat[idx[row]][col] / mat[idx[currow]][col];
            for (int c = 0; c <= nvar; c++)
                mat[idx[row]][c] -= k * mat[idx[currow]][c];
        }
        currow++;
    }
    for (int row = currow; row < neq; row++)
        if (mat[idx[row]][nvar] != 0) return vector<int>(1, -1);
    return freeVars;
}
```

Gaussian elimination mod P

```
const int MAXN = 110;

int MOD;
int inv[MAXN];

int mat[MAXN][MAXN];
int idx[MAXN];   // row index
int pivot[MAXN]; // pivot of row i

// Solves Ax = B, where A is a neq x nvar matrix and B is mat[*][nvar].
// Returns a vector of free variables
// (empty if system is defined, {-1} if no solution exists).
// Reduces matrix to reduced echelon form.
vector<int> solve(int nvar, int neq) {
    for (int i = 0; i < neq; i++) idx[i] = i;
    int currow = 0;
    vector<int> freeVars;
    for (int col = 0; col < nvar; col++) {
        int pivotrow = -1;
        for (int row = currow; row < neq; row++) {
            if (mat[idx[row]][col] != 0) {
                pivotrow = row;
                break;
            }
        }
        if (pivotrow == -1) { freeVars.push_back(col); continue; }
        swap(idx[currow], idx[pivotrow]);
        pivot[currow] = col;
        for (int c = 0; c <= nvar; c++) {
            if (c == col) continue;
            mat[idx[currow]][c] =
                (mat[idx[currow]][c] * inv[mat[idx[currow]][col]]) % MOD;
        }
        mat[idx[currow]][col] = 1;
        for (int row = 0; row < neq; row++) {
            if (row == currow) continue;
            int k = (mat[idx[row]][col] * inv[mat[idx[currow]][col]]) % MOD;
            for (int c = 0; c <= nvar; c++)
                mat[idx[row]][c] =
                    ((mat[idx[row]][c] - k * mat[idx[currow]][c]) % MOD + MOD) % MOD;
        }
        currow++;
    }
    for (int row = currow; row < neq; row++)
        if (mat[idx[row]][nvar] != 0) return vector<int>(1, -1);
    return freeVars;
}
```

Miller-Rabin primality test

```
namespace MillerRabin {
    typedef unsigned long long ulint;
    vector<ulint> magic = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

    ulint prod(ulint a, ulint b, ulint m) {
        ulint ret = 0, p = a;
        while (b) {
            if (b & 1) ret = (ret + p) % m;
            p = 2 * p % m;
            b >>= 1;
        }
        return ret;
    }

    ulint modexp(ulint b, ulint e, ulint m) {
        ulint ret = 1, p = b;
        while (e) {
            if (e & 1) ret = prod(ret, p, m);
            p = prod(p, p, m);
            e >>= 1;
        }
        return ret;
    }

    // O(log^2 n), works for any n < 2^63
    bool is_prime(ulint n) {
        if (n < 1) return 0;
        ulint d = n - 1;
        int s = 0;
        while (!(d & 1)) d >>= 1, s++;
        for (const ulint &a: magic) {
            if (n == a) return 1;
            ulint ad = modexp(a, d, n);
            if (ad == 1) continue;
            bool composite = 1;
            for (int r = 0; r < s; r++) {
                if (ad == n - 1) {
                    composite = 0;
                    break;
                }
                ad = prod(ad, ad, n);
            }
            if (composite) return 0;
        }
        return 1;
    }
}
```

Simpson integration rule

```
// Uses Simpson's rule to integrate f from x0 to x1 using 2*n subintervals
double integrate(function<double(double)> f, double x0, double x1, int n) {
    n *= 2; // n must be even
    double h = (x1 - x0) / n;
    double sum = f(x0) + f(x1);
    for (int i = 1; 2*i <= n; i++) {
        if (2*i < n) sum += 2*f(x0 + 2*i*h);
        sum += 4*f(x0 + (2*i - 1)*h);
    }
    return sum * h / 3;
}
```

Fast Fourier Transform

```
typedef complex<long double> Complex;
const long double PI = acos(-1.0L);

// Computes the DFT of vector v if type = 1, or the IDFT if type = -1
// If you are calculating the product of polynomials, don't forget to set both
// vectors' degrees to at least the sum of degrees of both polynomials, regardless
// of whether you will use only the first few elements of the resulting array
vector<Complex> FFT(vector<Complex> v, int type) {
    int n = v.size();
    while (n & (n - 1)) { v.push_back(0); n++; }
    int logn = __builtin_ctz(n);
    vector<Complex> v2(n);
    for (int i = 0; i < n; i++) {
        int mask = 0;
        for (int j = 0; j < logn; j++)
            if (i & (1 << j))
                mask |= (1 << (logn - 1 - j));
        v2[mask] = v[i];
    }
    for (int len = 1; 2 * len <= n; len <<= 1) {
        Complex wm(cos(type * PI / len), sin(type * PI / len));
        for (int i = 0; i < n; i += 2 * len) {
            Complex w = 1;
            for (int j = 0; j < len; j++) {
                Complex t = w * v2[i + j + len], u = v2[i + j];
                v2[i + j] = u + t; v2[i + j + len] = u - t;
                w *= wm;
            }
        }
    }
    if (type == -1) for (Complex &c: v2) c /= n;
    return v2;
}
```

Number Theoretic Transform

```
/*
  Number Theoretic Transform - O(N*logN)
  Performs FFT modulo prime P = c * 2^k + 1.
  k must be strictly greater than ceil(logN).
  r is a primitive root of P: it satisfies r^((P - 1) / f) != 1 (mod P),
  for every prime factor f of P - 1.
*/

vector<lint> NTT(vector<lint> v, int type, int c, int k, int r) {
    lint P = c * (1 << k) + 1;
    int n = v.size();
    while (n & (n - 1)) { v.push_back(0); n++; }
    int logn = __builtin_ctz(n);
    vector<lint> v2(n);
    for (int i = 0; i < n; i++) {
        int mask = 0;
        for (int j = 0; j < logn; j++)
            if (i & (1 << j))
                mask |= (1 << (logn - 1 - j));
        v2[mask] = v[i];
    }
    for (int len = 1, s = 1; 2 * len <= n; len <<= 1, s++) {
        lint wm = modexp(r, c * (1 << (k - s)), P);
        if (type == -1) wm = modexp(wm, P - 2, P);
        for (int i = 0; i < n; i += 2 * len) {
            lint w = 1;
            for (int j = 0; j < len; j++) {
                lint t = w * v2[i + j + len] % P;
                v2[i + j + len] = v2[i + j] - t;
                v2[i + j] = v2[i + j] + t;
                w = w * wm % P;
            }
        }
    }
    if (type == -1) {
        lint invn = modexp(n, P - 2, P);
        for (lint &x: v2) x = (x * invn) % P;
    }
    for (lint &x: v2) x = (x % P + P) % P;
    return v2;
}
```

Fast Walsh-Hadamard Transform

```
/*
  Fast Walsh-Hadamard Transform - O(N*logN)
  Similar to FFT, but instead of sum, computes OR, AND or XOR.
*/

enum FWHT_OP { OR, AND, XOR };
const vector<vector<int>> T = {{1, 1, 1, 0},  {0, 1, 1, 1},  {1, 1, 1, -1}};
const vector<vector<int>> I = {{0, 1, 1, -1}, {-1, 1, 1, 0}, {1, 1, 1, -1}};
vector<lint> FWHT(vector<lint> v, int type, FWHT_OP op) {
    while (v.size() & (v.size() - 1)) v.push_back(0);
    for (size_t len = 1; 2 * len <= v.size(); len *= 2) {
        for (size_t i = 0; i < v.size(); i += 2 * len) {
            for (size_t j = 0; j < len; j++) {
                lint x = v[i + j], y = v[i + len + j];
                if (type == 1) {
                    v[i + j] = T[op][0] * x + T[op][1] * y;
                    v[i + len + j] = T[op][2] * x + T[op][3] * y;
                }
                else {
                    v[i + j] = I[op][0] * x + I[op][1] * y;
                    v[i + len + j] = I[op][2] * x + I[op][3] * y;
                }
            }
        }
    }
    if (op == XOR && type == -1)
        for (auto &x: v)
            x /= v.size();
    return v;
}
```

6 Geometry (lolwut)

2D Geometry

```
const double EPS = 1e-9;
const double PI = acos(-1.0);

typedef int CTYPE;

//( cmp(a,b) _ 0 ) means (a _ b)
inline int cmp(double a, double b = 0) {
    return (a < b + EPS) ? (a + EPS < b) ? -1 : 0 : 1;
}

struct Point {
    CTYPE x,y;
    Point() {}
    Point(CTYPE xx,CTYPE yy) { x = xx; y = yy; }
    int _cmp(Point q) const {
```

```
        if (int t = cmp(x, q.x)) return t;
        return cmp(y, q.y);
    }
    bool operator==(Point q) const { return _cmp(q) == 0; }
    bool operator!=(Point q) const { return _cmp(q) != 0; }
    bool operator<(Point q) const { return _cmp(q) < 0; }
};

typedef Point Vector;
typedef vector<Point> Poly;

double norm(Vector &v) { return sqrt(v.x * v.x + v.y * v.y); }
Vector operator*(double k, const Vector &v) { return Vector(k * v.x, k * v.y); }
Vector operator/(const Vector &v, double k) { return Vector(v.x / k, v.y / k); }
Point operator+(const Point &a, const Point &b) { return Point(a.x + b.x, a.y + b.y); }
Point operator-(const Point &a, const Point &b) { return Point(a.x - b.x, a.y - b.y); }
CTYPE operator*(const Vector &u, const Vector &v) { return u.x * v.x + u.y * v.y; }
CTYPE operator^(const Vector &u, const Vector &v) { return u.x * v.y - u.y * v.x; }

// SIGNED area
double area(vector<Point>& polygon) {
    double ret = 0;
    int n = polygon.size();
    for(int i = 0; i < n; ++i) {
        int j = (i == n - 1 ?  0 : i + 1);
        ret += polygon[i] ^ polygon[j];
    }
    return 0.5 * ret;
}

// finds polygon centroid (needs SIGNED area)
double centroid(vector<Point>& polygon) {
    double S = area(polygon);
    double ret = 0;
    int n = polygon.size();
    for (int i = 0; i < n; ++i) {
        int j = (i == n-1 ? 0 : i + 1);
        ret += (polygon[i].x + polygon[j].x) * (polygon[i] ^ polygon[j]);
    }
    return ret / 6 / S;
}

// Distance from r to segment pq
double point_seg_dist(const Point &r, const Point &p, const Point &q) {
    Point A = r-q, B = r-p, C = q-p;
    double a = A*A, b = B*B, c = C*C;
    if(cmp(b, a+c) >= 0) return sqrt(a);
    else if(cmp(a, b+c) >= 0) return sqrt(b);
    else return fabs(A^B)/sqrt(c);
}

// Whether segments pq and rs have a common point
bool seg_intersects(const Point &p, const Point &q, const Point &r, const Point &s) {
    Point A = q - p, B = s - r, C = r - p, D = s - q;
    int a = cmp(A ^ C) + 2 * cmp(A ^ D);
    int b = cmp(B ^ C) + 2 * cmp(B ^ D);
    if (a == 3 || a == -3 || b == 3 || b == -3) return false;
    if (a || b || p == r || p == s || q == r || q == s) return true;
    int t = (p < r) + (p < s) + (q < r) + (q < s);
    return t != 0 && t != 4;
}

// Returns the intersection of lines pq and rs. Assumes pq and rs are not parallel.
Point intersection(const Point &p, const Point &q, const Point &r, const Point &s) {
    Point a = q - p, b = s - r, c = Point(p ^ q, r ^ s);
    assert(cmp(a ^ b));
    return Point(Point(a.x, b.x) ^ c, Point(a.y, b.y) ^ c) / (a ^ b);
}

// Returns convex hull in clockwise-order.
Poly convex_hull(vector<Point> poly) {
    sort(poly.begin(), poly.end(), [](const Point &a, const Point &b) {
        if (a.x == b.x) return a.y < b.y;
        return a.x < b.x;
    });
    Poly top, bot;
    int tlen = 0, blen = 0;
    for (const Point &p: poly) {
        while (tlen > 1 &&
                ((top[tlen - 2] - top[tlen - 1])^(p - top[tlen - 1])) <= 0) {
            tlen--;
            top.pop_back();
        }
        while (blen > 1 &&
                ((p - bot[blen - 1])^(bot[blen - 2] - bot[blen - 1])) <= 0) {
            blen--;
            bot.pop_back();
        }
        top.push_back(p);
        bot.push_back(p);
        tlen++;
        blen++;
    }
    for (int i = blen - 2; i > 0; i--)
        top.push_back(bot[i]);
    return top;
}

// Checks whether given point is inside a convex polygon.
// Assumes polygon vertices are given in CCW order.
bool in_polygon(const Point &p, const Poly &poly) {
    Vector vp = p - poly[0];
    if (((poly[1] - poly[0]) ^ vp) < 0) return 0;
    int l = 1, r = poly.size() - 1;
    while (l < r) {
        int mid = (l + r + 1) / 2;
        if (((poly[mid] - poly[0]) ^ vp) > 0) l = mid;
        else r = mid - 1;
    }
    if (l == (int)poly.size() - 1) return 0;
```

```
    return ((poly[l + 1] - poly[l]) ^ (p - poly[l])) > 0;
}
```

3D Geometry

```
typedef double CTYPE;

struct Point {
    CTYPE x, y, z;
    Point(CTYPE xx = 0, CTYPE yy = 0, CTYPE zz = 0) {
        x = xx, y = yy, z = zz;
    }
};

typedef Point Vector;

double norm(const Vector &v) { return sqrt(v.x*v.x + v.y*v.y + v.z*v.z); }

Point operator+(const Point &p, const Vector &v) {
    return Point(p.x + v.x, p.y + v.y, p.z + v.z);
}

Vector operator-(const Point &p, const Point &q) {
    return Vector(p.x - q.x, p.y - q.y, p.z - q.z);
}

CTYPE operator*(const Vector &u, const Vector &v) {
    return u.x*v.x + u.y*v.y + u.z*v.z;
}

Vector operator*(CTYPE k, const Vector &v) {
    return Vector(k*v.x, k*v.y, k*v.z);
}

Vector operator^(const Vector &u, const Vector &v) {
    return Vector(u.y*v.z - u.z*v.y,
                  u.z*v.x - u.x*v.z,
                  u.x*v.y - u.y*v.x);
}

// finds Ax + By + Cz + D = 0, given three points on the plane
tuple<double, double, double, double>
get_plane_equation(const Point &p1, const Point &p2, const Point &p3) {
    Vector u = p2 - p1, v = p3 - p1;
    Vector n = u ^ v;
    return make_tuple(n.x, n.y, n.z, -n.x*p1.x - n.y*p1.y - n.z*p1.z);
}

// finds Ax + By + Cz + D = 0, given a point and a normal vector
tuple<double, double, double, double>
get_plane_equation(const Point &p, const Vector &n) {
    return make_tuple(n.x, n.y, n.z, -(p * n));
}

pair<Point, bool> get_line_plane_intersection(const Point &p1, const Point &p2,
                                              double a, double b, double c, double d) {
    Vector v = p2 - p1, n(a, b, c);
    double t = -(d + p1*n) / (v * n);
    Point p = p1 + t*v;
    bool intersects = (0 - EPS <= t && t <= 1 + EPS);
    return make_pair(p, intersects);
}

double get_point_line_dist(const Point &p, const Point &pr, const Vector &dir) {
    Vector u = p - pr;
    return norm(u ^ dir) / norm(dir);
}
```
