ACM ICPC Reference

University of São Paulo

November 8, 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;

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

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

Bridge and articulation point

```
namespace bap {
    int ind[MAXN], low[MAXN];
    int cnt;

    void dfs(int v, int prev = -1) {
        ind[v] = low[v] = cnt++;
        int cont = 0;
        bool flag = 0;
        for (int nxt: adj[v]) {
            if (ind[nxt] == -1) {
                ++cont;
                dfs(nxt, v);
                low[v] = min(low[v], low[nxt]);
                if (low[nxt] >= ind[v]) flag = 1;
                if (low[nxt] == ind[nxt]) {
                    // v-nxt is a bridge
                }
            }
            else if (nxt != prev) low[v] = min(low[v], ind[nxt]);
        }
        if (prev == -1) {
            if (cont > 1) {
                // v is an articulation point
            }
        }
        else if (flag) {
            // v is an articulation point
        }
    }

    void init() {
        memset(ind, -1, sizeof(ind));
        cnt = 0;
    }
}
```

Dinic maxflow

```
// Dinic maxflow, min(O(EV^2),O(maxflow*E)(?)) worst case
// O(E*min(V^2/3,sqrt(E))) for unit caps (O(E*sqrt(V)) if bipartite)

typedef int FTYPE;      // define as needed

const int MAXV = 5010;
const FTYPE FINF = INF; // infinite flow

struct Edge {
    int to;
    FTYPE cap;
    Edge(int t, FTYPE c) { to = t; cap = c; }
};

vector<int> adj[MAXV];
vector<Edge> edge, s_edge;
int ptr[MAXV], dinic_dist[MAXV];

// Inserts an edge u->v with capacity c
inline void add_edge(int u, int v, FTYPE c) {
    adj[u].push_back(edge.size());
    edge.push_back(Edge(v, c));
    s_edge.push_back(Edge(v, c));
    adj[v].push_back(edge.size());
    edge.push_back(Edge(u, 0)); // modify to Edge(u, c) if graph is non-directed
    s_edge.push_back(Edge(u, 0));
}

bool dinic_bfs(int _s, int _t) {
    memset(dinic_dist, -1, sizeof(dinic_dist));
    dinic_dist[_s] = 0;
    queue<int> q;
    q.push(_s);
    while (!q.empty() && dinic_dist[_t] == -1) {
        int v = q.front();
        q.pop();
        for (size_t a = 0; a < adj[v].size(); ++a) {
            int ind = adj[v][a];
            int nxt = edge[ind].to;
            if (dinic_dist[nxt] == -1 && edge[ind].cap) {
                dinic_dist[nxt] = dinic_dist[v] + 1;
                q.push(nxt);
            }
        }
    }
    return dinic_dist[_t] != -1;
}

FTYPE dinic_dfs(int v, int _t, FTYPE flow) {
    if (v == _t) return flow;
    for (int &a = ptr[v]; a < (int)adj[v].size(); ++a) {
        int ind = adj[v][a];
        int nxt = edge[ind].to;
        if (dinic_dist[nxt] == dinic_dist[v] + 1 && edge[ind].cap) {
            FTYPE got = dinic_dfs(nxt, _t, min(flow, edge[ind].cap));
            if (got) {
                edge[ind].cap -= got;
                edge[ind^1].cap += got;
                return got;
            }
        }
    }
    return 0;
}
```

```
FTYPE dinic(int _s, int _t) {
    FTYPE ret = 0, got;
    while (dinic_bfs(_s, _t)) {
        memset(ptr, 0, sizeof(ptr));
        while ((got = dinic_dfs(_s, _t, FINF))) ret += got;
    }
    return ret;
}

// Removes all flow but keeps graph structure
void dinic_reset() {
    for (int i = 0; i < (int)edge.size(); i++)
        edge[i].cap = s_edge[i].cap;
}

// Clears dinic structure
inline void dinic_clear() {
    for (int i = 0; i < MAXV; ++i) adj[i].clear();
    edge.clear();
}
```

Edmonds 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;
            }
        }
    }
    // ...
}
```

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;
}
```

Gomory-Hu Tree (flow-equivalent tree)

```
// Gomory-Hu Tree
// Finds flow-equivalent tree of a graph
// Minimum edge in an s-t path corresponds to the maxflow of s-t.
// Output is NOT necessarily a cut tree (i.e. minimum edge might not partition
// the graph into a minimum s-t cut).
// Uses Gusfield's algorithm: Performs V - 1 maxflows
// Requires: dinic

const int LOG = 20;

int cut[MAXV], up[LOG][MAXV], val[LOG][MAXV];
int level[MAXV];

void gomory_hu(int n) {
    for (int v = 1; v <= n; v++) up[0][v] = 1;
    level[1] = 0;
    for (int s = 2; s <= n; s++) {
        dinic_reset();
        val[0][s] = dinic(s, up[0][s]);
        level[s] = level[up[0][s]] + 1;

        memset(cut, 0, sizeof(cut));
        queue<int> q;
        q.push(s);
        cut[s] = 1;
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int i: adj[v]) {
                int nxt = edge[i].to;
                if (edge[i].cap && !cut[nxt]) {
                    cut[nxt] = 1;
                    q.push(nxt);
                }
            }
        }

        for (int v = s + 1; v <= n; v++)
            if (cut[v] && up[0][v] == up[0][s])
                up[0][v] = s;
    }

    // prepares LCA
    for (int i = 1; i < LOG; i++) {
        for (int v = 1; v <= n; v++) {
            up[i][v] = up[i - 1][up[i - 1][v]];
            val[i][v] = min(val[i - 1][v], val[i - 1][up[i - 1][v]]);
        }
    }
}
```

Min-cost maxflow

```
// Min-cost Max-flow ( O(V*E + V^2*MAXFLOW) )

typedef int FTYPE; // type of flow
typedef int CTYPE; // type of cost
typedef pair<FTYPE,CTYPE> pfc; // pair<flow,cost>

const int MAXV = 510;
const CTYPE CINF = INF; // infinite cost
const FTYPE FINF = INF; // infinite flow

void operator+=(pfc &p1,pfc &p2) { p1.first += p2.first; p1.second += p2.second; }

struct Edge {
    int to;
    FTYPE cap;
    CTYPE cost;
    Edge(int a,FTYPE cp,CTYPE ct) { to = a; cap = cp; cost = ct; }
};

vector<int> adj[MAXV];
vector<Edge> edge;
int V =; // number of vertices (don't forget to set!)

// Inserts an edge u->v with capacity c and cost cst
inline void add_edge(int u,int v,FTYPE c,CTYPE cst) {
    adj[u].push_back(edge.size());
    edge.push_back(Edge(v,c,cst));
    adj[v].push_back(edge.size());
    edge.push_back(Edge(u,0,-cst));
}

FTYPE flow[MAXV];
CTYPE dist[MAXV], pot[MAXV];
int prv[MAXV], e_ind[MAXV];
bool foi[MAXV];

void bellman_ford(int _s) {
    for (int a = 0; a < V; ++a) dist[a] = CINF;
    dist[_s] = 0;
    for (int st = 0; st < V; ++st) {
        for (int v = 0; v < V; ++v) {
            for (size_t a = 0; a < adj[v].size(); ++a) {
                int ind = adj[v][a];
                int nxt = edge[ind].to;
                if (!edge[ind].cap) continue;
                dist[nxt] = min(dist[nxt],dist[v] + edge[ind].cost);
            }
        }
    }
}

pfc dijkstra(int _s,int _t) { // O(V^2)
    for (int a = 0; a<V; ++a) {
        dist[a] = CINF;
        foi[a] = 0;
    }
    dist[_s] = 0;
    flow[_s] = FINF;
    while (1) {
        int v;
        CTYPE d = CINF;
        for (int a = 0; a < V; ++a) {
            if (foi[a] || dist[a] >= d) continue;
            d = dist[a];
            v = a;
        }
        if (d == CINF) break;
        foi[v] = 1;
        for (size_t a = 0; a < adj[v].size(); ++a) {
            int ind = adj[v][a];
            int nxt = edge[ind].to;
            if (!edge[ind].cap || dist[nxt] <= dist[v] + edge[ind].cost + pot[v] - pot[
            dist[nxt] = dist[v] + edge[ind].cost + pot[v] - pot[nxt];
            prv[nxt] = v;
            e_ind[nxt] = ind;
            flow[nxt] = min(flow[v],edge[ind].cap);
        }
    }
    if (dist[_t] == CINF) return pfc(FINF,CINF);
    for (int a = 0; a < V; ++a) pot[a] += dist[a];
    pfc ret(flow[_t],0);
    for (int cur = _t; cur != _s; cur = prv[cur]) {
        int ind = e_ind[cur];
        edge[ind].cap -= flow[_t];
        edge[ind^1].cap += flow[_t];
        ret.second += flow[_t] * edge[ind].cost; // careful with overflow!
    }
    return ret;
}

// Returns a pair (max-flow, min-cost)
pfc mcmf(int _s,int _t) {
    pfc ret(0,0), got;
    bellman_ford(_s);
    for (int a = 0; a < V; ++a) pot[a] = dist[a];
    while( (got = dijkstra(_s,_t)).first != FINF ) ret += got;
    return ret;
}

// Clears mcmf structure
inline void mcmf_clear() {
    edge.clear();
    for (int a = 0; a < V; ++a) adj[a].clear();
}
```

Tarjan (Strongly Connected Components)

```
namespace Tarjan {
    int cmp[MAXN]; // component of [i]
    int cnt;       // number of components
    int ind[MAXN], low[MAXN], pre;
    bool instack[MAXN];
    stack<int> st;

    void tarjan(int v) {
        ind[v] = low[v] = pre++;
        st.push(v);
        instack[v] = 1;
        for (int nxt: adj[v]) {
            if (ind[nxt] == -1) {
                tarjan(nxt);
                low[v] = min(low[v], low[nxt]);
            }
            else if (instack[nxt]) low[v] = min(low[v], ind[nxt]);
        }
        if (ind[v] == low[v]) {
            int vv;
            do {
                vv = st.top();
                st.pop();
                instack[vv] = 0;
                cmp[vv] = cnt;
            } while (vv != v);
            ++cnt;
        }
    }

    inline void init() {
        memset(ind, -1, sizeof(ind));
        pre = 0; cnt = 0;
    }
}
```

3 Strings

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

```
typedef char HType;

const int P1 = 31, P2 = 37, MOD = (int)1e9 + 7;

struct Hash {
    ll h1, h2;
    Hash(ll a = 0, ll b = 0) { h1 = a; h2 = b; }
    void append(HType c) {
        h1 = (P1*h1 + c) % MOD;
        h2 = (P2*h2 + c) % MOD;
    }
    bool operator== (Hash that) const { return h1 == that.h1 && h2 == that.h2; }
    bool operator!= (Hash that) const { return h1 != that.h1 || h2 != that.h2; }
    Hash operator*  (Hash that) const {
        return Hash((h1*that.h1)%MOD, (h2*that.h2)%MOD);
    }
    Hash operator-  (Hash that) const {
        return Hash( (h1 - that.h1 + MOD)%MOD, (h2 - that.h2 + MOD)%MOD);
    }
};

Hash pot[MAXN];

vector<Hash> build_hash(int n, HType *v) {
    pot[0] = Hash(1,1);
    vector<Hash> ret;
    Hash acc;
    for (int i = 0; i < n; i++) {
        acc.append(v[i]);
        ret.push_back(acc);
        if (i > 0) pot[i] = pot[i-1] * Hash(P1, P2);
    }
    return ret;
}

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

KMP

```
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

```
// 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;
}
```

4 Data structures

Convex trick optimization (constant version)

```
// 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 trick optimization

```
// 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);
    }
};
```

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);
}
```

Constant RMQ

```
int rmq[LOG][MAXN];
int v[MAXN];
int n;

// Builds RMQ structure for array v of size n in O(n*log(n))
void build_rmq() {
    for (int i = 0; i < n; i++)
        rmq[0][i] = v[i];
    for (int log = 1; log < LOG; ++log) {
        for (int i = 0; i < n; i++) {
            rmq[log][i] = min(rmq[log-1][i], rmq[log-1][min(n-1, i + (1<<(log-1)))]);
        }
    }
}
// l e r inclusives
int get_rmq(int l, int r) {
    int len = r - l + 1;
    int bit = 31 - __builtin_clz(len);
    return min(rmq[bit][l], rmq[bit][r - (1<<bit) + 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;
    }
}
```

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 s = 0, m = 2; s<logn; s++, m <<= 1) {
        Complex wm(cos(2.L * type * PI / m), sin(2.L * type * PI / m));
        for (int k = 0; k < n; k += m) {
            Complex w = 1;
            for (int j = 0; 2 * j < m; j++) {
                Complex t = w * v2[k + j + (m >> 1)], u = v2[k + j];
                v2[k + j] = u + t; v2[k + j + (m >> 1)] = u - t;
                w *= wm;
            }
        }
    }
    if (type == -1) for (Complex &c: v2) c /= n;
    return v2;
}
```

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;
}
```

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;
}
```

6 Geometry

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);
}
```