

Team notebook

Universidad Tecnologica de Pereira

2 de abril de 2018

Índice

1. Algorithms	2		
1.1. Mo's algorithm on trees	2		
1.2. mo's algorithm	2		
2. Data Structures	3		
2.1. BIT	3		
2.2. persistent tree	3		
2.3. seg tree	4		
2.4. stl order statistic	4		
2.5. treap	4		
2.6. wavelet tree	6		
3. Dynamic Programming	6		
3.1. convex hull trick	6		
3.2. divide and conquer	7		
3.3. dp on trees	8		
4. Geometry	8		
4.1. all	8		
4.2. center 2 points + radius	10		
4.3. closest pair	11		
4.4. convex hull	11		
4.5. triangles	12		
5. Graphs	12		
5.1. bridges	12		
5.2. dinic	13		
5.3. euler formula	14		
5.4. eurlian	14		
5.5. heavy light decomposition	15		
		5.6. lca	15
		5.7. max flow min cost	16
		5.8. two sat	16
		6. Math	17
		6.1. FFT	17
		6.2. fibonacci	18
		6.3. lucas	18
		7. Number theory	18
		7.1. NTT	18
		7.2. all	19
		7.3. pollard rho	20
		7.4. totient	20
		8. Strings	21
		8.1. aho	21
		8.2. kmp	22
		8.3. suffix array	22
		8.4. z algorithm	24
		9. X - Misc	24
		9.1. equations	24
		9.2. Trigonometry	24
		9.3. Triangles	24
		9.4. Quadrilaterals	25
		9.5. Spherical coordinates	25
		9.6. Derivatives/Integrals	25
		9.7. Sums	25
		9.8. Series	25
		9.9. Geometric series	25

1. Algorithms

1.1. Mo's algorithm on trees

```

void flat(vector<vector<edge>> &g, vector<int> &a,
          vector<int> &le, vector<int> &ri, vector<int> &cost,
          int node, int pi, int &ts, int w) {

    cost[node] = w;
    le[node] = ts;
    a[ts] = node;
    ts++;
    for (auto e : g[node]) {
        if (e.to == pi) continue;
        flat(g, a, le, ri, cost, e.to, node, ts, e.w);
    }
    ri[node] = ts;
    a[ts] = node;
    ts++;
}

/** Case: cost in nodes: let P = LCA(u, v), le(u) <= le(v)
 * Case 1: P = u
 * In this case, our query range would be [le(u),le(v)].
 * Case 2: P != u
 * In this case, our query range would be [ri(u),le(v)] +
 * [le(P),le(P)].*/

// Case when the cost is in the edges.
void compute_queries(vector<vector<edge>> &g) {
    // g is undirected
    int n = g.size();
    lca_tree.init(g, 0);
    vector<int> a(2 * n), le(n), ri(n), cost(n);
    // a: nodes in the flatten array
    // le: left id of the given node
    // ri: right id of the given node
    // cost: cost of the edge from the node to the parent
    int ts = 0; // timestamp
    flat(g, a, le, ri, cost, 0, -1, ts, 0);
    int q; cin >> q;
    vector<query> queries(q);
    for (int i = 0; i < q; i++) {
        int u, v;

```

```

        cin >> u >> v;
        u--; v--;
        int lca = lca_tree.query(u, v);
        if (le[u] > le[v])
            swap(u, v);
        queries[i].id = i;
        queries[i].lca = lca;
        queries[i].u = u;
        queries[i].v = v;
        if (lca == u) {
            queries[i].a = le[u] + 1;
            queries[i].b = le[v];
        } else {
            queries[i].a = ri[u];
            queries[i].b = le[v];
        }
    }
    solve_mo(queries, a, le, cost); // this is the usual algorithm
}

```

1.2. mo's algorithm

```

const int MN = 5 * 100000 + 100;
const int SN = 708;
struct query {
    int a, b, id;
    query() {}
    query(int x, int y, int i) : a(x), b(y), id(i) {}
    bool operator < (const query &o) const {
        return b < o.b;
    }
};
vector<query> s[SN];
int ans[MN];
struct DS {
    void clear() {}
    void insert(int x) {}
    void erase(int x) {}
    long long query() {}
};
DS data;
int main() {
    int n, q;

```

```

while (cin >> n >> q) {
    for (int i = 0; i < SN; ++i)
        s[i].clear();
    vector<int> a(n);
    for (auto &i : a) cin >> i;
    for (int i = 0; i < q; ++i) {
        int b, e;
        cin >> b >> e;
        b--; e--;
        s[b / SN].emplace_back(b, e, i);
    }
    for (int i = 0; i < SN; ++i) {
        if (s[i].size()) sort(s[i].begin(), s[i].end());
    }
    for (int b = 0; b < SN; ++b) {
        if (s[b].size() == 0) continue;
        int i = s[b][0].a;
        int j = s[b][0].a - 1;
        data.clear();
        for (int k = 0; k < (int)s[b].size(); ++k) {
            int L = s[b][k].a;
            int R = s[b][k].b;
            while (j < R) { j++; data.insert(a[j]); }
            while (j > R) { data.erase(a[j]); j--; }
            while (i < L) { data.erase(a[i]); i++; }
            while (i > L) { i--; data.insert(a[i]); }
            ans[s[b][k].id] = data.query();
        }
    }
    for (int i = 0; i < q; ++i) {
        cout << ans[i] << endl;
    }
}
return 0;
};

```

2. Data Structures

2.1. BIT

```

int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);

```

```

// add v to value at x
void set(int x, int v) {
    while(x <= N) {
        tree[x] += v;
        x += (x & -x);
    }
}
// get cumulative sum up to and including x
int get(int x) {
    int res = 0;
    while(x) {
        res += tree[x];
        x -= (x & -x);
    }
    return res;
}
// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
    int idx = 0, mask = N;
    while(mask && idx < N) {
        int t = idx + mask;
        if(x >= tree[t]) {
            idx = t;
            x -= tree[t];
        }
        mask >>= 1;
    }
    return idx;
}

```

2.2. persistent tree

```

// Persistent binary trie (BST for integers)
const int MD = 31;

struct node_bin {
    node_bin *child[2];
    int val;

    node_bin() : val(0) {
        child[0] = child[1] = NULL;
    }
}

```

```

};

typedef node_bin* pnode_bin;

pnode_bin copy_node(pnode_bin cur) {
    pnode_bin ans = new node_bin();
    if (cur) *ans = *cur;
    return ans;
}

pnode_bin modify(pnode_bin cur, int key, int inc, int id = MD) {
    pnode_bin ans = copy_node(cur);
    ans->val += inc;
    if (id >= 0) {
        int to = (key >> id) & 1;
        ans->child[to] = modify(ans->child[to], key, inc, id - 1);
    }
    return ans;
}

int sum_smaller(pnode_bin cur, int key, int id = MD) {
    if (cur == NULL) return 0;
    if (id < 0) return 0; // strictly smaller
    // if (id == - 1) return cur->val; // smaller or equal

    int ans = 0;
    int to = (key >> id) & 1;
    if (to) {
        if (cur->child[0]) ans += cur->child[0]->val;
        ans += sum_smaller(cur->child[1], key, id - 1);
    } else {
        ans = sum_smaller(cur->child[0], key, id - 1);
    }
    return ans;
}

```

2.3. seg tree

```

/**
 * Important notes:
 * - When using lazy propagation remembert to create new versions for
   each push_down operation!!!
 * - remember to set left and right pointers to NULL

```

```

* */

struct node {
    long long ans, pending;
    node *left, *right;
    node() : ans(0), left(NULL), right(NULL), pending(0) {}
};

```

2.4. stl order statistic

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int,null_type,less<int>,//key,mapped type, comparator
            rb_tree_tag,tree_order_statistics_node_update> set_t;
//find_by_order(i) devuelve iterador al i-esimo elemento
//order_of_key(k): devuelve la pos del lower bound de k
//Ej: 12, 100, 505, 1000, 10000.
//order_of_key(10) == 0, order_of_key(100) == 1,
//order_of_key(707) == 3, order_of_key(9999999) == 5

```

2.5. treap

```

#define null NULL

struct node {
    int x, y, size;
    long long sum;
    node *l, *r;
    node(int k) : x(k), y(rand()), size(1),
                l(null), r(null), sum(0) { }
};

node* relax(node* p) {
    if (p) {
        p->size = 1;
        p->sum = p->x;
        if (p->l) {
            p->size += p->l->size;
            p->sum += p->l->sum;
        }
    }
}

```

```

    if (p->r) {
        p->size += p->r->size;
        p->sum += p->r->sum;
    }
}
return p;
}

// Puts all elements <= x in l and all elements > x in r.
void split(node* t, int x, node* &l, node* &r) {
    if (t == null) l = r = null; else {
        if (t->x <= x) {
            split(t->r, x, t->r, r);
            l = relax(t);
        } else {
            split(t->l, x, l, t->l);
            r = relax(t);
        }
    }
}

node* merge(node* l, node* r) {
    if (l == null) return relax(r);
    if (r == null) return relax(l);
    if (l->y > r->y) {
        l->r = merge(l->r, r);
        return relax(l);
    } else {
        r->l = merge(l, r->l);
        return relax(r);
    }
}

node* insert(node* t, node* m) {
    if (t == null || m->y > t->y) {
        split(t, m->x, m->l, m->r);
        return relax(m);
    }
    if (m->x < t->x) t->l = insert(t->l, m);
    else t->r = insert(t->r, m);
    return relax(t);
}

node* erase(node* t, int x) {
    if (t == null) return null;

```

```

    if (t->x == x) {
        node* q = merge(t->l, t->r);
        delete t;
        return relax(q);
    } else {
        if (x < t->x) t->l = erase(t->l, x);
        else t->r = erase(t->r, x);
        return relax(t);
    }
}

// Returns any node with the given x.
node* find(node* cur, int x) {
    while (cur != null and cur->x != x) {
        if (x < cur->x) cur = cur->l;
        else cur = cur->r;
    }
    return cur;
}

node* find_kth(node* cur, int k) {
    while (cur != null and k >= 0) {
        if (cur->l && cur->l->size > k) {
            cur = cur->l;
            continue;
        }
        if (cur->l)
            k -= cur->l->size;
        if (k == 0) return cur;
        k--;
        cur = cur->r;
    }
    return cur;
}

long long sum(node* p, int x) { // find the sum of elements <= x
    if (p == null) return 0LL;
    if (p->x > x) return sum(p->l, x);
    long long ans = (p->l ? p->l->sum : 0) + p->x + sum(p->r, x);
    assert(ans >= 0);
    return ans;
}

```

2.6. wavelet tree

```

struct wavelet {
    vector<int> values, ori;
    vector<int> map_left, map_right;
    int l, r, m;
    wavelet *left, *right;
    wavelet() : left(NULL), right(NULL) {}
    wavelet(int a, int b, int c) : l(a), r(b), m(c), left(NULL),
        right(NULL) {}
};

wavelet *init(vector<int> &data, vector<int> &ind, int lo, int hi) {
    if (lo > hi || (data.size() == 0)) return NULL;
    int mid = ((long long)(lo) + hi) / 2;
    if (lo + 1 == hi) mid = lo; // handle negative values

    wavelet *node = new wavelet(lo, hi, mid);

    vector<int> data_l, data_r, ind_l, ind_r;
    int ls = 0, rs = 0;
    for (int i = 0; i < int(data.size()); i++) {
        int value = data[i];
        if (value <= mid) {
            data_l.emplace_back(value);
            ind_l.emplace_back(ind[i]);
            ls++;
        } else {
            data_r.emplace_back(value);
            ind_r.emplace_back(ind[i]);
            rs++;
        }
        node->map_left.emplace_back(ls);
        node->map_right.emplace_back(rs);
        node->values.emplace_back(value);
        node->ori.emplace_back(ind[i]);
    }

    if (lo < hi) {
        node->left = init(data_l, ind_l, lo, mid);
        node->right = init(data_r, ind_r, mid + 1, hi);
    }
    return node;
}

```

```

int kth(wavelet *node, int to, int k) {
    // returns the kth element in the sorted version of (a[0], ..., a[to])
    if (node->l == node->r) return node->m;
    int c = node->map_left[to];
    if (k < c)
        return kth(node->left, c - 1, k);
    return kth(node->right, node->map_right[to] - 1, k - c);
}

int pos_kth_occurrence(wavelet *node, int val, int k) {
    // returns the position on the original array of the kth occurrence of
    // the value "val"
    if (!node) return -1;

    if (node->l == node->r) {
        if (int(node->ori.size()) <= k)
            return -1;
        return node->ori[k];
    }

    if (val <= node->m)
        return pos_kth_occurrence(node->left, val, k);
    return pos_kth_occurrence(node->right, val, k);
}

```

3. Dynamic Programming

3.1. convex hull trick

```

struct line {
    long long m, b;
    line (long long a, long long c) : m(a), b(c) {}
    long long eval(long long x) {
        return m * x + b;
    }
};

long double inter(line a, line b) {
    long double den = a.m - b.m;
    long double num = b.b - a.b;
    return num / den;
}

```

```

/**
 * min m_i * x_j + b_i, for all i.
 *   x_j <= x_{j + 1}
 *   m_i >= m_{j + 1}
 */
struct ordered_cht {
    vector<line> ch;
    int idx; // id of last "best" in query
    ordered_cht() {
        idx = 0;
    }

    void insert_line(long long m, long long b) {
        line cur(m, b);
        // new line's slope is less than all the previous
        while (ch.size() > 1 &&
            (inter(cur, ch[ch.size() - 2]) >= inter(cur, ch[ch.size() - 1]))) {
            // f(x) is better in interval [inter(ch.back(), cur), inf)
            ch.pop_back();
        }

        ch.push_back(cur);
    }

    long long eval(long long x) { // minimum
        // current x is greater than all the previous x,
        // if that is not the case we can make binary search.
        idx = min<int>(idx, ch.size() - 1);
        while (idx + 1 < (int)ch.size() && ch[idx + 1].eval(x) <=
            ch[idx].eval(x))
            idx++;

        return ch[idx].eval(x);
    }
};

/**
 * Dynammic convex hull trick
 */

typedef long long int64;
typedef long double float128;

```

```

const int64 is_query = -(1LL<<62), inf = 1e18;

struct Line {
    int64 m, b;
    mutable function<const Line*> succ;
    bool operator<(const Line& rhs) const {
        if (rhs.b != is_query) return m < rhs.m;
        const Line* s = succ();
        if (!s) return 0;
        int64 x = rhs.m;
        return b - s->b < (s->m - m) * x;
    }
};

struct HullDynamic : public multiset<Line> { // will maintain upper hull
    for maximum
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;
        return (float128)(x->b - y->b)*(z->m - y->m) >= (float128)(y->b -
            z->b)*(y->m - x->m);
    }

    void insert_line(int64 m, int64 b) {
        auto y = insert({ m, b });
        y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
        if (bad(y)) { erase(y); return; }
        while (next(y) != end() && bad(next(y))) erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }

    int64 eval(int64 x) {
        auto l = *lower_bound((Line) { x, is_query });
        return l.m * x + l.b;
    }
};

```

3.2. divide and conquer

```

/**
 * recurrence:
 *   dp[k][i] = min dp[k-1][j] + c[i][j - 1], for all j > i;
 *
 * "comp" computes dp[k][i] for all i in O(n log n) (k is fixed)
 * */

void comp(int l, int r, int le, int re) {
    if (l > r) return;

    int mid = (l + r) >> 1;

    int best = max(mid + 1, le);
    dp[cur][mid] = dp[cur ^ 1][best] + cost(mid, best - 1);
    for (int i = best; i <= re; i++) {
        if (dp[cur][mid] > dp[cur ^ 1][i] + cost(mid, i - 1)) {
            best = i;
            dp[cur][mid] = dp[cur ^ 1][i] + cost(mid, i - 1);
        }
    }

    comp(l, mid - 1, le, best);
    comp(mid + 1, r, best, re);
}

```

3.3. dp on trees

```

/**
 * for any node, save the total answer and the answer of every children.
 * for the query(node, pi) the answer is ans[node] - partial[node][pi]
 * cases:
 *   - all children missing
 *   - no child is missing
 *   - missing child is current pi
 * */

void add_edge(int u, int v) {
    int id_u_v = g[u].size();
    int id_v_u = g[v].size();
    g[u].emplace_back(v, id_v_u); // id of the parent in the child's list
    (g[v][id] -> u)
    g[v].emplace_back(u, id_u_v); // id of the parent in the child's list
    (g[u][id] -> v)
}

```

4. Geometry

4.1. all

```

double INF = 1e100;
double EPS = 1e-12;

struct PT {
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c); }
    PT operator / (double c) const { return PT(x/c, y/c); }
};

double dot(PT p, PT q) { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
    return os << "(" << p.x << ", " << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p) { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
    return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
    return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
    double r = dot(b-a,b-a);
    if (fabs(r) < EPS) return a;
    r = dot(c-a, b-a)/r;
    if (r < 0) return a;
}

```



```

    if (r > 1) return b;
    return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
    double a, double b, double c, double d) {
    return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
    return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a, b, c, d)
        && fabs(cross(a-b, a-c)) < EPS
        && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if (LinesCollinear(a, b, c, d)) {
        if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
            dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
        if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
            return false;
        return true;
    }
    if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
    if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
    return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first

```

```

PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
    b=b-a; d=c-d; c=c-a;
    assert(dot(b, b) > EPS && dot(d, d) > EPS);
    return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
    b=(a+b)/2;
    c=(a+c)/2;
    return ComputeLineIntersection(b, b+RotateCW90(a-b), c,
        c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
    bool c = 0;
    for (int i = 0; i < p.size(); i++){
        int j = (i+1)%p.size();
        if ((p[i].y <= q.y && q.y < p[j].y ||
            p[j].y <= q.y && q.y < p[i].y) &&
            q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y -
            p[i].y))
            c = !c;
    }
    return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
    for (int i = 0; i < p.size(); i++)
        if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
            return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {

```

```

vector<PT> ret;
b = b-a;
a = a-c;
double A = dot(b, b);
double B = dot(a, b);
double C = dot(a, a) - r*r;
double D = B*B - A*C;
if (D < -EPS) return ret;
ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(Pt a, Pt b, double r, double R) {
    vector<PT> ret;
    double d = sqrt(dist2(a, b));
    if (d > r+R || d+min(r, R) < max(r, R)) return ret;
    double x = (d*d-R*R+r*r)/(2*d);
    double y = sqrt(r*r-x*x);
    Pt v = (b-a)/d;
    ret.push_back(a+v*x + RotateCCW90(v)*y);
    if (y > 0)
        ret.push_back(a+v*x - RotateCCW90(v)*y);
    return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion. Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
    double area = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        area += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
    return fabs(ComputeSignedArea(p));
}

```

```

PT ComputeCentroid(const vector<PT> &p) {
    PT c(0,0);
    double scale = 6.0 * ComputeSignedArea(p);
    for (int i = 0; i < p.size(); i++){
        int j = (i+1) % p.size();
        c = c + (p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
    for (int i = 0; i < p.size(); i++) {
        for (int k = i+1; k < p.size(); k++) {
            int j = (i+1) % p.size();
            int l = (k+1) % p.size();
            if (i == l || j == k) continue;
            if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

```

4.2. center 2 points + radius

```

vector<point> find_center(point a, point b, long double r) {
    point d = (a - b) * 0.5;
    if (d.dot(d) > r * r) {
        return vector<point> ();
    }
    point e = b + d;
    long double fac = sqrt(r * r - d.dot(d));
    vector<point> ans;
    point x = point(-d.y, d.x);
    long double l = sqrt(x.dot(x));
    x = x * (fac / l);
    ans.push_back(e + x);
    x = point(d.y, -d.x);
    x = x * (fac / l);
    ans.push_back(e + x);
    return ans;
}

```

}

4.3. closest pair

```
struct point {
    double x, y;
    int id;
    point() {}
    point (double a, double b) : x(a), y(b) {}
};

double dist(const point &o, const point &p) {
    double a = p.x - o.x, b = p.y - o.y;
    return sqrt(a * a + b * b);
}

double cp(vector<point> &p, vector<point> &x, vector<point> &y) {
    if (p.size() < 4) {
        double best = 1e100;
        for (int i = 0; i < p.size(); ++i)
            for (int j = i + 1; j < p.size(); ++j)
                best = min(best, dist(p[i], p[j]));
        return best;
    }

    int ls = (p.size() + 1) >> 1;
    double l = (p[ls - 1].x + p[ls].x) * 0.5;
    vector<point> xl(ls), xr(p.size() - ls);
    unordered_set<int> left;
    for (int i = 0; i < ls; ++i) {
        xl[i] = x[i];
        left.insert(x[i].id);
    }
    for (int i = ls; i < p.size(); ++i) {
        xr[i - ls] = x[i];
    }

    vector<point> yl, yr;
    vector<point> pl, pr;
    yl.reserve(ls); yr.reserve(p.size() - ls);
    pl.reserve(ls); pr.reserve(p.size() - ls);
    for (int i = 0; i < p.size(); ++i) {
        if (left.count(y[i].id))
```

```
        yl.push_back(y[i]);
    else
        yr.push_back(y[i]);

    if (left.count(p[i].id))
        pl.push_back(p[i]);
    else
        pr.push_back(p[i]);
}

double dl = cp(pl, xl, yl);
double dr = cp(pr, xr, yr);
double d = min(dl, dr);
vector<point> yp; yp.reserve(p.size());
for (int i = 0; i < p.size(); ++i) {
    if (fabs(y[i].x - l) < d)
        yp.push_back(y[i]);
}
for (int i = 0; i < yp.size(); ++i) {
    for (int j = i + 1; j < yp.size() && j < i + 7; ++j) {
        d = min(d, dist(yp[i], yp[j]));
    }
}
return d;
}

double closest_pair(vector<point> &p) {
    vector<point> x(p.begin(), p.end());
    sort(x.begin(), x.end(), [](const point &a, const point &b) {
        return a.x < b.x;
    });
    vector<point> y(p.begin(), p.end());
    sort(y.begin(), y.end(), [](const point &a, const point &b) {
        return a.y < b.y;
    });
    return cp(p, x, y);
}
```

4.4. convex hull

```
#define REMOVE_REDUNDANT
```

```
typedef double T;
```

```

const T EPS = 1e-7;
struct PT {
    T x, y;
    PT() {}
    PT(T x, T y) : x(x), y(y) {}
    bool operator<(const PT &rhs) const { return make_pair(y,x) <
        make_pair(rhs.y,rhs.x); }
    bool operator==(const PT &rhs) const { return make_pair(y,x) ==
        make_pair(rhs.y,rhs.x); }
};

T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
    return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 &&
        (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
    sort(pts.begin(), pts.end());
    pts.erase(unique(pts.begin(), pts.end()), pts.end());
    vector<PT> up, dn;
    for (int i = 0; i < pts.size(); i++) {
        while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >=
            0) up.pop_back();
        while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <=
            0) dn.pop_back();
        up.push_back(pts[i]);
        dn.push_back(pts[i]);
    }
    pts = dn;
    for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
    if (pts.size() <= 2) return;
    dn.clear();
    dn.push_back(pts[0]);
    dn.push_back(pts[1]);
    for (int i = 2; i < pts.size(); i++) {
        if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
        dn.push_back(pts[i]);
    }
}

```

```

if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
}
pts = dn;
#endif
}

```

4.5. triangles

Let a, b, c be length of the three sides of a triangle.

$$p = (a + b + c) * 0.5$$

The inradius is defined by:

$$iR = \sqrt{\frac{(p-a)(p-b)(p-c)}{p}}$$

The radius of its circumcircle is given by the formula:

$$cR = \frac{abc}{\sqrt{(a+b+c)(a+b-c)(a+c-b)(b+c-a)}}$$

5. Graphs

5.1. bridges

```

struct edge{
    int to, id;
    edge(int a, int b) : to(a), id(b) {}
};

struct graph {
    vector<vector<edge>> > g;
    vector<int> vi, low, d, pi, is_b;

    int ticks, edges;

    graph(int n, int m) {
        g.assign(n, vector<edge>());
        is_b.assign(m, 0);
        vi.resize(n);
    }
}

```

```

low.resize(n);
d.resize(n);
pi.resize(n);
edges = 0;
}

void add_edge(int u, int v) {
    g[u].push_back(edge(v, edges));
    g[v].push_back(edge(u, edges));
    edges++;
}

void dfs(int u) {
    vi[u] = true;
    d[u] = low[u] = ticks++;
    for (int i = 0; i < g[u].size(); ++i) {
        int v = g[u][i].to;
        if (v == pi[u]) continue;
        if (!vi[v]) {
            pi[v] = u;
            dfs(v);
            if (d[u] < low[v])
                is_b[g[u][i].id] = true;

            low[u] = min(low[u], low[v]);
        } else {
            low[u] = min(low[u], d[v]);
        }
    }
}

// Multiple edges from a to b are not allowed.
// (they could be detected as a bridge).
// If you need to handle this, just count
// how many edges there are from a to b.
void comp_bridges() {
    fill(pi.begin(), pi.end(), -1);
    fill(vi.begin(), vi.end(), 0);
    fill(low.begin(), low.end(), 0);
    fill(d.begin(), d.end(), 0);
    ticks = 0;
    for (int i = 0; i < g.size(); ++i)
        if (!vi[i]) dfs(i);
}

```

```
};
```

5.2. dinic

```

// taken from
// https://github.com/jaehyunp/stanfordacm/blob/master/code/MinCostMaxFlow.cc
typedef long long LL;

struct edge {
    int u, v;
    LL cap, flow;
    edge() {}
    edge(int u, int v, LL cap): u(u), v(v), cap(cap), flow(0) {}
};

struct dinic {
    int N;
    vector<edge> E;
    vector<vector<int>>> g;
    vector<int> d, pt;

    dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

    void add_edge(int u, int v, LL cap) {
        if (u != v) {
            E.emplace_back(edge(u, v, cap));
            g[u].emplace_back(E.size() - 1);
            E.emplace_back(edge(v, u, 0));
            g[v].emplace_back(E.size() - 1);
        }
    }

    bool bfs(int S, int T) {
        queue<int> q({S});
        fill(d.begin(), d.end(), N + 1);
        d[S] = 0;
        while(!q.empty()) {
            int u = q.front(); q.pop();
            if (u == T) break;
            for (int k: g[u]) {
                edge &e = E[k];
                if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
                    d[e.v] = d[e.u] + 1;

```

```

        q.emplace(e.v);
    }
}
}
return d[T] != N + 1;
}

LL dfs(int u, int T, LL flow = -1) {
    if (u == T || flow == 0) return flow;
    for (int &i = pt[u]; i < int(g[u].size()); ++i) {
        edge &e = E[g[u][i]];
        edge &oe = E[g[u][i]^1];
        if (d[e.v] == d[e.u] + 1) {
            LL amt = e.cap - e.flow;
            if (flow != -1 && amt > flow) amt = flow;
            if (LL pushed = dfs(e.v, T, amt)) {
                e.flow += pushed;
                oe.flow -= pushed;
                return pushed;
            }
        }
    }
    return 0;
}

LL max_flow(int S, int T) {
    LL total = 0;
    while (bfs(S, T)) {
        fill(pt.begin(), pt.end(), 0);
        while (LL flow = dfs(S, T))
            total += flow;
    }
    return total;
}
};

```

5.3. euler formula

Euler's formula states that if a finite, connected, planar graph is drawn in the plane without any edge intersections, and v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer, infinitely large region), then:

$$f + v = e + 2$$

It can be extended to non connected planar graphs with c connected components:

$$f + v = e + c + 1$$

5.4. eurelian

```

struct edge;
typedef list<edge>::iterator iter;

struct edge {
    int next_vertex;
    iter reverse_edge;

    edge(int next_vertex) :next_vertex(next_vertex) {}
};

const int max_vertices = 6666;
int num_vertices;
list<edge> adj[max_vertices]; // adjacency list
vector<int> path;

void find_path(int v) {
    while(adj[v].size() > 0) {
        int vn = adj[v].front().next_vertex;
        adj[vn].erase(adj[v].front().reverse_edge);
        adj[v].pop_front();
        find_path(vn);
    }
    path.push_back(v);
}

void add_edge(int a, int b) {
    adj[a].push_front(edge(b));
    iter ita = adj[a].begin();
    adj[b].push_front(edge(a));
    iter itb = adj[b].begin();
    ita->reverse_edge = itb;
    itb->reverse_edge = ita;
}

```

5.5. heavy light decomposition

```
// Heavy-Light Decomposition
struct TreeDecomposition {
    vector<int> g[MAXN], c[MAXN];
    int s[MAXN]; // subtree size
    int p[MAXN]; // parent id
    int r[MAXN]; // chain root id
    int t[MAXN]; // index used in segtree/bit/...
    int d[MAXN]; // depth
    int ts;

    void dfs(int v, int f) {
        p[v] = f;
        s[v] = 1;
        if (f != -1) d[v] = d[f] + 1;
        else d[v] = 0;

        for (int i = 0; i < g[v].size(); ++i) {
            int w = g[v][i];
            if (w != f) {
                dfs(w, v);
                s[v] += s[w];
            }
        }
    }

    void hld(int v, int f, int k) {
        t[v] = ts++;
        c[k].push_back(v);
        r[v] = k;

        int x = 0, y = -1;
        for (int i = 0; i < g[v].size(); ++i) {
            int w = g[v][i];
            if (w != f) {
                if (s[w] > x) {
                    x = s[w];
                    y = w;
                }
            }
        }
        if (y != -1) {
            hld(y, v, k);
        }
    }
}
```

```
for (int i = 0; i < g[v].size(); ++i) {
    int w = g[v][i];
    if (w != f && w != y) {
        hld(w, v, w);
    }
}

void init(int n) {
    for (int i = 0; i < n; ++i) {
        g[i].clear();
    }
}

void add(int a, int b) {
    g[a].push_back(b);
    g[b].push_back(a);
}

void build() {
    ts = 0;
    dfs(0, -1);
    hld(0, 0, 0);
}
};
```

5.6. lca

```
void init(vector<vector<edge> > &g, int root) {
    // g is undirected
    dfs(g, root);
    int N = g.size(), i, j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            P[i][j] = -1;
            MI[i][j] = inf;
        }
    }

    for (i = 0; i < N; i++) {
        P[i][0] = T[i];
    }
}
```

```

    MI[i][0] = W[i];
}

for (j = 1; 1 << j < N; j++)
    for (i = 0; i < N; i++)
        if (P[i][j - 1] != -1) {
            P[i][j] = P[P[i][j - 1]][j - 1];
            MI[i][j] = min(MI[i][j-1], MI[ P[i][j - 1] ][j - 1]);
        }
}

```

5.7. max flow min cost

```

struct MCMF {
    typedef int ctype;
    enum { MAXN = 1000, INF = INT_MAX };
    struct Edge { int x, y; ctype cap, cost; };
    vector<Edge> E;    vector<int> adj[MAXN];
    int N, prev[MAXN]; ctype dist[MAXN], phi[MAXN];

    MCMF(int NN) : N(NN) {}

    void add(int x, int y, ctype cap, ctype cost) { // cost >= 0
        Edge e1={x,y,cap,cost}, e2={y,x,0,-cost};
        adj[e1.x].push_back(E.size()); E.push_back(e1);
        adj[e2.x].push_back(E.size()); E.push_back(e2);
    }

    void mcmf(int s, int t, ctype &flowVal, ctype &flowCost) {
        int x;
        flowVal = flowCost = 0; memset(phi, 0, sizeof(phi));
        while (true) {
            for (x = 0; x < N; x++) prev[x] = -1;
            for (x = 0; x < N; x++) dist[x] = INF;
            dist[s] = prev[s] = 0;

            set< pair<ctype, int> > Q;
            Q.insert(make_pair(dist[s], s));
            while (!Q.empty()) {
                x = Q.begin()->second; Q.erase(Q.begin());
                FOREACH(it, adj[x]) {
                    const Edge &e = E[*it];
                    if (e.cap <= 0) continue;

```

```

                ctype cc = e.cost + phi[x] - phi[e.y]; // ***
                if (dist[x] + cc < dist[e.y]) {
                    Q.erase(make_pair(dist[e.y], e.y));
                    dist[e.y] = dist[x] + cc;
                    prev[e.y] = *it;
                    Q.insert(make_pair(dist[e.y], e.y));
                }
            }
            if (prev[t] == -1) break;

            ctype z = INF;
            for (x = t; x != s; x = E[prev[x]].x)
                { z = min(z, E[prev[x]].cap); }
            for (x = t; x != s; x = E[prev[x]].x)
                { E[prev[x]].cap -= z; E[prev[x]^1].cap += z; }
            flowVal += z;
            flowCost += z * (dist[t] - phi[s] + phi[t]);
            for (x = 0; x < N; x++)
                { if (prev[x] != -1) phi[x] += dist[x]; } // ***
        }
    }
};

```

5.8. two sat

```

vector<int> G[MAX];
vector<int> GT[MAX];
vector<int> Ftime;
vector<vector<int> > SCC;
bool visited[MAX];
int n;

void dfs1(int n){
    visited[n] = 1;

    for (int i = 0; i < G[n].size(); ++i) {
        int curr = G[n][i];
        if (visited[curr]) continue;
        dfs1(curr);
    }
}

```



```

    Ftime.push_back(n);
}

void dfs2(int n, vector<int> &scc) {
    visited[n] = 1;
    scc.push_back(n);

    for (int i = 0; i < GT[n].size(); ++i) {
        int curr = GT[n][i];
        if (visited[curr]) continue;
        dfs2(curr, scc);
    }
}

void kosaraju() {
    memset(visited, 0, sizeof visited);

    for (int i = 0; i < 2 * n; ++i) {
        if (!visited[i]) dfs1(i);
    }

    memset(visited, 0, sizeof visited);
    for (int i = Ftime.size() - 1; i >= 0; i--) {
        if (visited[Ftime[i]]) continue;
        vector<int> _scc;
        dfs2(Ftime[i], _scc);
        SCC.push_back(_scc);
    }
}

/**
 * After having the SCC, we must traverse each scc, if in one SCC are -b
 * y b, there is not a solution.
 * Otherwise we build a solution, making the first "node" that we find
 * truth and its complement false.
 */

bool two_sat(vector<int> &val) {
    kosaraju();
    for (int i = 0; i < SCC.size(); ++i) {
        vector<bool> tmpvisited(2 * n, false);
        for (int j = 0; j < SCC[i].size(); ++j) {
            if (tmpvisited[SCC[i][j] ^ 1]) return 0;

```

```

            if (val[SCC[i][j]] != -1) continue;
            else {
                val[SCC[i][j]] = 0;
                val[SCC[i][j] ^ 1] = 1;
            }
            tmpvisited[SCC[i][j]] = 1;
        }
    }
    return 1;
}

```

6. Math

6.1. FFT

```

typedef long double T;

const T pi = acos(-1);

struct cpx {
    T real, image;
    cpx(T _real, T _image) {
        real = _real;
        image = _image;
    }
    cpx() {}
};

cpx operator + (const cpx &c1, const cpx &c2) {
    return cpx(c1.real + c2.real, c1.image + c2.image);
}

cpx operator - (const cpx &c1, const cpx &c2) {
    return cpx(c1.real - c2.real, c1.image - c2.image);
}

cpx operator * (const cpx &c1, const cpx &c2) {
    return cpx(c1.real * c2.real - c1.image * c2.image, c1.real
        * c2.image + c1.image * c2.real);
}

int rev(int id, int len) {
    int ret = 0;
    for (int i = 0; (1 << i) < len; i++) {

```

```

    ret <= 1;
    if (id & (1 << i)) ret |= 1;
}
return ret;
}

void fft(cpx *a, int len, int dir) {
    for (int i = 0; i < len; i++) {
        A[rev(i, len)] = a[i];
    }

    for (int s = 1; (1 << s) <= len; s++) {
        int m = (1 << s);

        cpx wm = cpx(cos(dir * 2 * pi / m), sin(dir * 2 * pi / m));
        for (int k = 0; k < len; k += m) {
            cpx w = cpx(1, 0);
            for (int j = 0; j < (m >> 1); j++) {
                cpx t = w * A[k + j + (m >> 1)];
                cpx u = A[k + j];
                A[k + j] = u + t;
                A[k + j + (m >> 1)] = u - t;
                w = w * wm;
            }
        }
    }
    if (dir == -1) for (int i = 0; i < len; i++) A[i].real /= len,
        A[i].image /= len;
    for (int i = 0; i < len; i++) a[i] = A[i];
}

```

6.2. fibonacci

Let A , B and n be integer numbers.

$$k = A - B \quad (1)$$

$$F_A F_B = F_{k+1} F_A^2 + F_k F_A F_{A-1} \quad (2)$$

$$\sum_{i=0}^n F_i^2 = F_{n+1} F_n \quad (3)$$

$ev(n)$ = returns 1 if n is even.

$$\sum_{i=0}^n F_i F_{i+1} = F_{n+1}^2 - ev(n) \quad (4)$$

$$\sum_{i=0}^n F_i F_{i-1} = \sum_{i=0}^{n-1} F_i F_{i+1} \quad (5)$$

6.3. lucas

For non-negative integers m and n and a prime p , the following congruence relation holds :

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{n_i} \pmod{p},$$

where :

$$m = m_k p^k + m_{k-1} p^{k-1} + \dots + m_1 p + m_0,$$

and :

$$n = n_k p^k + n_{k-1} p^{k-1} + \dots + n_1 p + n_0$$

are the base p expansions of m and n respectively. This uses the convention that $\binom{m}{n} = 0$ if $m \leq n$.

7. Number theory

7.1. NTT

```

typedef long long int LL;
typedef pair<LL, LL> PLL;

/* The following vector of pairs contains pairs (prime, generator)
 * where the prime has an Nth root of unity for N being a power of two.
 * The generator is a number g s.t g^(p-1)=1 (mod p)
 * but is different from 1 for all smaller powers */
vector<PLL> nth_roots_unity {
    {1224736769,330732430},{1711276033,927759239},{167772161,167489322},
    {469762049,343261969},{754974721,643797295},{1107296257,883865065}};

PLL ext_euclid(LL a, LL b) {
    if (b == 0)
        return make_pair(1,0);
    pair<LL,LL> rc = ext_euclid(b, a % b);
    return make_pair(rc.second, rc.first - (a / b) * rc.second);
}

```

```

}

//returns -1 if there is no unique modular inverse
LL mod_inv(LL x, LL modulo) {
    PLL p = ext_euclid(x, modulo);
    if ( (p.first * x + p.second * modulo) != 1 )
        return -1;
    return (p.first+modulo) % modulo;
}

//Number theory fft. The size of a must be a power of 2
void ntfft(vector<LL> &a, int dir, const PLL &root_unity) {
    int n = a.size();
    LL prime = root_unity.first;
    LL basew = mod_pow(root_unity.second, (prime-1) / n, prime);
    if (dir < 0) basew = mod_inv(basew, prime);
    for (int m = n; m >= 2; m >>= 1) {
        int mh = m >> 1;
        LL w = 1;
        for (int i = 0; i < mh; i++) {
            for (int j = i; j < n; j += m) {
                int k = j + mh;
                LL x = (a[j] - a[k] + prime) % prime;
                a[j] = (a[j] + a[k]) % prime;
                a[k] = (w * x) % prime;
            }
            w = (w * basew) % prime;
        }
        basew = (basew * basew) % prime;
    }
    int i = 0;
    for (int j = 1; j < n - 1; j++) {
        for (int k = n >> 1; k > (i ^ k); k >>= 1);
        if (j < i) swap(a[i], a[j]);
    }
}

```

7.2. all

```

// Discrete logarithm
// Computes x which a ^ x = b mod n.
long long d_log(long long a, long long b, long long n) {
    long long m = ceil(sqrt(n));

```

```

    long long aj = 1;
    map<long long, long long> M;
    for (int i = 0; i < m; ++i) {
        if (!M.count(aj))
            M[aj] = i;
        aj = (aj * a) % n;
    }

    long long coef = mod_pow(a, n - 2, n);
    coef = mod_pow(coef, m, n);
    // coef = a ^ (-m)
    long long gamma = b;
    for (int i = 0; i < m; ++i) {
        if (M.count(gamma)) {
            return i * m + M[gamma];
        } else {
            gamma = (gamma * coef) % n;
        }
    }
    return -1;
}

void ext_euclid(long long a, long long b, long long &x, long long &y,
               long long &g) {
    x = 0, y = 1, g = b;
    long long m, n, q, r;
    for (long long u = 1, v = 0; a != 0; g = a, a = r) {
        q = g / a, r = g % a;
        m = x - u * q, n = y - v * q;
        x = u, y = v, u = m, v = n;
    }
}

/**
 * Chinese remainder theorem.
 * Find z such that z % x[i] = a[i] for all i.
 */
long long crt(vector<long long> &a, vector<long long> &x) {
    long long z = 0;
    long long n = 1;
    for (int i = 0; i < x.size(); ++i)
        n *= x[i];

    for (int i = 0; i < a.size(); ++i) {
        long long tmp = (a[i] * (n / x[i])) % n;

```

```

    tmp = (tmp * mod_inv(n / x[i], x[i])) % n;
    z = (z + tmp) % n;
}

return (z + n) % n;
}

```

7.3. pollard rho

```

const int rounds = 20;
// checks whether a is a witness that n is not prime, 1 < a < n
bool witness(long long a, long long n) {
    // check as in Miller Rabin Primality Test described
    long long u = n - 1;
    int t = 0;
    while (u % 2 == 0) {
        t++;
        u >>= 1;
    }
    long long next = mod_pow(a, u, n);
    if (next == 1) return false;
    long long last;
    for (int i = 0; i < t; ++i) {
        last = next;
        next = mod_mul(last, last, n);
        if (next == 1) {
            return last != n - 1;
        }
    }
    return next != 1;
}
// Checks if a number is prime with prob 1 - 1 / (2 ^ it)
bool miller_rabin(long long n, int it = rounds) {
    if (n <= 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;
    for (int i = 0; i < it; ++i) {
        long long a = rand() % (n - 1) + 1;
        if (witness(a, n)) {
            return false;
        }
    }
    return true;
}

```

```

}

long long pollard_rho(long long n) {
    long long x, y, i = 1, k = 2, d;
    x = y = rand() % n;
    while (1) {
        ++i;
        x = mod_mul(x, x, n);
        x += 2;
        if (x >= n) x -= n;
        if (x == y) return 1;
        d = __gcd(abs(x - y), n);
        if (d != 1) return d;
        if (i == k) {
            y = x;
            k *= 2;
        }
    }
    return 1;
}

// Returns a list with the prime divisors of n
vector<long long> factorize(long long n) {
    vector<long long> ans;
    if (n == 1)
        return ans;
    if (miller_rabin(n)) {
        ans.push_back(n);
    } else {
        long long d = 1;
        while (d == 1)
            d = pollard_rho(n);
        vector<long long> dd = factorize(d);
        ans = factorize(n / d);
        for (int i = 0; i < dd.size(); ++i)
            ans.push_back(dd[i]);
    }
    return ans;
}

```

7.4. totient

```

for (int i = 1; i < MN; i++)

```

```

    phi[i] = i;

for (int i = 1; i < MN; i++)
    if (!sieve[i]) // is prime
        for (int j = i; j < MN; j += i)
            phi[j] -= phi[j] / i;

long long totient(long long n) {
    if (n == 1) return 0;
    long long ans = n;
    for (int i = 0; primes[i] * primes[i] <= n; ++i) {
        if ((n % primes[i]) == 0) {
            while ((n % primes[i]) == 0) n /= primes[i];
            ans -= ans / primes[i];
        }
    }
    if (n > 1) {
        ans -= ans / n;
    }
    return ans;
}

```

8. Strings

8.1. aho

```

// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 6 * 50 + 10;
const int MAXC = 26;
// Bit i in this mask is on if the keyword with index i
// appears when the machine enters this state.
int out[MAXS];

int f[MAXS]; // Failure function
int g[MAXS][MAXC]; // Goto function, or -1 if fail.

int buildMatchingMachine(const vector<string> &words,
    char lowestChar = 'a', char highestChar = 'z') {
    memset(out, 0, sizeof out);
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);
}

```

```

int states = 1; // Initially, we just have the 0 state

for (int i = 0; i < words.size(); ++i) {
    const string &keyword = words[i];
    int currentState = 0;
    for (int j = 0; j < keyword.size(); ++j) {
        int c = keyword[j] - lowestChar;
        if (g[currentState][c] == -1) {
            g[currentState][c] = states++;
        }
        currentState = g[currentState][c];
    }
    // There's a match of keywords[i] at node currentState.
    out[currentState] |= (1 << i);
}

// State 0 should have an outgoing edge for all characters.
for (int c = 0; c < MAXC; ++c) {
    if (g[0][c] == -1) {
        g[0][c] = 0;
    }
}

queue<int> q;
// Iterate over every possible input
for (int c = 0; c <= highestChar - lowestChar; ++c) {
    // All nodes s of depth 1 have f[s] = 0
    if (g[0][c] != -1 and g[0][c] != 0) {
        f[g[0][c]] = 0;
        q.push(g[0][c]);
    }
}

while (q.size()) {
    int state = q.front();
    q.pop();
    for (int c = 0; c <= highestChar - lowestChar; ++c) {
        if (g[state][c] != -1) {
            int failure = f[state];
            while (g[failure][c] == -1) {
                failure = f[failure];
            }
            failure = g[failure][c];
            f[g[state][c]] = failure;
        }
    }
}

```

```

        // Merge out values
        out[g[state][c]] |= out[failure];
        q.push(g[state][c]);
    }
}
}
return states;
}
int findNextState(int currentState, char nextInput,
    char lowestChar = 'a') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while (g[answer][c] == -1) answer = f[answer];
    return g[answer][c];
}

```

8.2. kmp

```

void kmp(const string &needle, const string &haystack) {
    // Precompute border function
    int m = needle.size();
    vector<int> border(m);
    border[0] = 0;
    for (int i = 1; i < m; ++i) {
        border[i] = border[i - 1];
        while (border[i] > 0 and needle[i] != needle[border[i]]) {
            border[i] = border[border[i] - 1];
        }
        if (needle[i] == needle[border[i]]) border[i]++;
    }

    // Now the actual matching
    int n = haystack.size();
    int seen = 0;
    for (int i = 0; i < n; ++i){
        while (seen > 0 and haystack[i] != needle[seen]) {
            seen = border[seen - 1];
        }
        if (haystack[i] == needle[seen]) seen++;
        if (seen == m) {
            printf("Needle occurs from %d to %d\n", i - m + 1, i);
            seen = border[m - 1];
        }
    }
}

```

```

    }
}

```

8.3. suffix array

```

// Complexity: O(n log n)
// * * * IMPORTANT: The last character of s must compare less
// than any other character (for example, do s = s + '\1';
// before calling this function).
//Output:
// sa = The suffix array. Contains the n suffixes of s sorted
// in lexicographical order. Each suffix is represented
// as a single integer (the position in the string
// where it starts).
// rank = The inverse of the suffix array. rank[i] = the index
// of the suffix s[i..n) in the pos array. (In other
// words, sa[i] = k <=> rank[k] = i).
// With this array, you can compare two suffixes in O(1):
// Suffix s[i..n) is smaller than s[j..n) if and
// only if rank[i] < rank[j].
namespace SuffixArray {
    int t, rank[MAXN], sa[MAXN], lcp[MAXN];
    bool compare(int i, int j){
        return rank[i + t] < rank[j + t];
    }
    void build(const string &s){
        int n = s.size();
        int bc[256];
        for (int i = 0; i < 256; ++i) bc[i] = 0;
        for (int i = 0; i < n; ++i) ++bc[s[i]];
        for (int i = 1; i < 256; ++i) bc[i] += bc[i-1];
        for (int i = 0; i < n; ++i) sa[--bc[s[i]]] = i;
        for (int i = 0; i < n; ++i) rank[i] = bc[s[i]];
        for (t = 1; t < n; t <= 1){
            for (int i = 0, j = 1; j < n; i = j++){
                while (j < n && rank[sa[j]] == rank[sa[i]]) j++;
                if (j - i == 1) continue;
                int *start = sa + i, *end = sa + j;
                sort(start, end, compare);
                int first = rank[*start + t], num = i, k;
                for(; start < end; rank[*start++] = num){
                    k = rank[*start + t];
                    if (k != first and (i > first or k >= j))

```

```

        first = k, num = start - sa;
    }
}
// Remove this part if you don't need the LCP
int size = 0, i, j;
for(i = 0; i < n; i++) if (rank[i] > 0) {
    j = sa[rank[i] - 1];
    while(s[i + size] == s[j + size]) ++size;
    lcp[rank[i]] = size;
    if (size > 0) --size;
}
lcp[0] = 0;
}
};
// Applications:
// lcp(x,y) = min(lcp(x,x+1), lcp(x+1, x+2), ... , lcp(y-1, y))
void number_of_different_substrings(){
    // If you have the i-th smaller suffix, Si,
    // it's length will be |Si| = n - sa[i]
    // Now, lcp[i] stores the number of
    // common letters between Si and Si-1
    // (s.substr(sa[i]) and s.substr(sa[i-1]))
    // so, you have |Si| - lcp[i] different strings
    // from these two suffixes => n - lcp[i] - sa[i]
    for(int i = 0; i < n; ++i) ans += n - sa[i] - lcp[i];
}
void number_of_repeated_substrings(){
    // Number of substrings that appear at least twice in the text.
    // The trick is that all 'spare' substrings that can give us
    // Lcp(i - 1, i) can be obtained by Lcp(i - 2, i - 1)
    // due to the ordered nature of our array.
    // And the overall answer is
    // Lcp(0, 1) +
    // Sum(max[0, Lcp(i, i - 1) - Lcp(i - 2, i - 1)])
    // for 2 <= i < n
    // File Recover
    int cnt = lcp[1];
    for(int i=2; i < n; ++i){
        cnt += max(0, lcp[i] - lcp[i-1]);
    }
}
void repeated_m_times(int m){
    // Given a string s and an int m, find the size
    // of the biggest substring repeated m times (find the rightmost pos)

```

```

    // if a string is repeated m+1 times, then it's repeated m times too
    // The answer is the maximum, over i, of the longest common prefix
    // between suffix i+m-1 in the sorted array.
    int length = 0, position = -1, t;
    for (int i = 0; i <= n-m; ++i){
        if ((t = getLcp(i, i+m-1, n)) > length){
            length = t;
            position = sa[i];
        } else if (t == length) { position = max(position, sa[i]); }
    }
    // Here you'll get the rightmost position
    // (that means, the last time the substring appears)
    for (int i = 0; i < n; ){
        if (sa[i] + length > n) { ++i; continue; }
        int ps = 0, j = i+1;
        while (j < n && lcp[j] >= length){
            ps = max(ps, sa[j]);
            j++;
        }
        if(j - i >= m) position = max(position, ps);
        i = j;
    }
    if(length != 0)
        printf("%d %d\n", length, position);
    else
        puts("none");
}
void smallest_rotation(){
    // Reads a string of length k. Then just double it (s = s+s)
    // and find the suffix array.
    // The answer is the smallest i for which s.size() - sa[i] >= k
    // If you want the first appearance (and not the string)
    // you'll need the second cycle
    int best = 0;
    for (int i=0; i < n; ++i){
        if (n - sa[i] >= k){
            //Find the first appearance of the string
            while (n - sa[i] >= k){
                if(sa[i] < sa[best] && sa[i] != 0) best = i;
                i++;
            }
            break;
        }
    }
    if (sa[best] == k) puts("0");
}

```

```

    else printf("%d\n", sa[best]);
}

```

8.4. z algorithm

```

vector<int> compute_z(const string &s){
    int n = s.size();
    vector<int> z(n,0);
    int l,r;
    r = l = 0;
    for(int i = 1; i < n; ++i){
        if(i > r) {
            l = r = i;
            while(r < n and s[r - 1] == s[r])r++;
            z[i] = r - l;r--;
        }else{
            int k = i-l;
            if(z[k] < r - i +1) z[i] = z[k];
            else {
                l = i;
                while(r < n and s[r - l] == s[r])r++;
                z[i] = r - l;r--;
            }
        }
    }
    return z;
}

```

9. X - Misc

9.1. equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{aligned} ax + by = e \\ cx + dy = f \end{aligned} \Rightarrow \begin{aligned} x &= \frac{ed - bf}{ad - bc} \\ y &= \frac{af - ec}{ad - bc} \end{aligned}$$

In general, given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A'_i}{\det A}$$

where A'_i is A with the i 'th column replaced by b .

9.2. Trigonometry

$$\sin(v + w) = \sin v \cos w + \cos v \sin w$$

$$\cos(v + w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v + w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$

$$\sin v + \sin w = 2 \sin \frac{v + w}{2} \cos \frac{v - w}{2}$$

$$\cos v + \cos w = 2 \cos \frac{v + w}{2} \cos \frac{v - w}{2}$$

$$(V + W) \tan(v - w)/2 = (V - W) \tan(v + w)/2$$

where V, W are lengths of sides opposite angles v, w .

$$a \cos x + b \sin x = r \cos(x - \phi)$$

$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2 + b^2}, \phi = \text{atan2}(b, a)$.

9.3. Triangles

Side lengths: a, b, c

Semiperimeter: $p = \frac{a + b + c}{2}$

Area: $A = \sqrt{p(p - a)(p - b)(p - c)}$

Circumradius: $R = \frac{abc}{4A}$

Inradius: $r = \frac{A}{p}$

Length of median (divides triangle into two equal-area triangles):
 $m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$

Length of bisector (divides angles in two): $s_a = \sqrt{bc \left[1 - \left(\frac{a}{b+c} \right)^2 \right]}$

Law of sines: $\frac{\sin \alpha}{a} = \frac{\sin \beta}{b} = \frac{\sin \gamma}{c} = \frac{1}{2R}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc \cos \alpha$

Law of tangents: $\frac{a+b}{a-b} = \frac{\tan \frac{\alpha+\beta}{2}}{\tan \frac{\alpha-\beta}{2}}$

9.4. Quadrilaterals

With side lengths a, b, c, d , diagonals e, f , diagonals angle θ , area A and magic flux $F = b^2 + d^2 - a^2 - c^2$:

$$4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2 f^2 - F^2}$$

For cyclic quadrilaterals the sum of opposite angles is 180° , $ef = ac + bd$, and $A = \sqrt{(p-a)(p-b)(p-c)(p-d)}$.

9.5. Spherical coordinates

$$\begin{aligned} x &= r \sin \theta \cos \phi & r &= \sqrt{x^2 + y^2 + z^2} \\ y &= r \sin \theta \sin \phi & \theta &= \arccos(z/\sqrt{x^2 + y^2 + z^2}) \\ z &= r \cos \theta & \phi &= \operatorname{atan2}(y, x) \end{aligned}$$

9.6. Derivatives/Integrals

$$\begin{aligned} \frac{d}{dx} \arcsin x &= \frac{1}{\sqrt{1-x^2}} & \frac{d}{dx} \arccos x &= -\frac{1}{\sqrt{1-x^2}} \\ \frac{d}{dx} \tan x &= 1 + \tan^2 x & \frac{d}{dx} \arctan x &= \frac{1}{1+x^2} \\ \int \tan ax &= -\frac{\ln |\cos ax|}{a} & \int x \sin ax &= \frac{\sin ax - ax \cos ax}{a^2} \\ \int e^{-x^2} &= \frac{\sqrt{\pi}}{2} \operatorname{erf}(x) & \int x e^{ax} dx &= \frac{e^{ax}}{a^2} (ax - 1) \end{aligned}$$

Integration by parts:

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

9.7. Sums

$$c^a + c^{a+1} + \dots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1$$

$$\begin{aligned} 1 + 2 + 3 + \dots + n &= \frac{n(n+1)}{2} \\ 1^2 + 2^2 + 3^2 + \dots + n^2 &= \frac{n(2n+1)(n+1)}{6} \\ 1^3 + 2^3 + 3^3 + \dots + n^3 &= \frac{n^2(n+1)^2}{4} \\ 1^4 + 2^4 + 3^4 + \dots + n^4 &= \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \end{aligned}$$

9.8. Series

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, (-\infty < x < \infty) \\ \ln(1+x) &= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots, (-1 < x \leq 1) \\ \sqrt{1+x} &= 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \dots, (-1 \leq x \leq 1) \\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, (-\infty < x < \infty) \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, (-\infty < x < \infty) \end{aligned}$$

9.9. Geometric series

$$r \neq 1$$

$$a + ar + ar^2 + ar^3 + \dots + ar^{n-1} = \sum_{k=0}^{n-1} ar^k = a \left(\frac{1-r^n}{1-r} \right)$$