```
" Vim config file
set number
set autoindent
set ts=4
syntax on
map! ii <Esc>
set cindent shiftwidth=4

" Autocompletion with <tab>
function InsertTabWrapper()
    let col = col('.') - 1
    if !col || getline('.')[col - 1] !~ '\k'
        return "\<tab>"
    else
        return "\<c-p>"
    endif
endfunction
inoremap <tab> <c-r>=InsertTabWrapper()<cr>
```

```
//Different kinds of range trees and fenvik tree implementation
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <cassert>
#include <algorithm>

typedef int val_t;
#define size (1 << 17)
#define inf 0x3f3f3f3f

//RMQ without interval modification
struct rmq_t {
    val_t mas[size<<2];

    rmq_t() {}
    rmq_t(val_t* a) {
        memcpy(mas+size, a, sizeof(val_t)*size);
        for(int i = size-1; i > 0; i--)
            mas[i] = std::min(mas[i<<1], mas[(i<<1)+1]);
    }

    void modify(int ind, val_t val) {
        ind += size;
        mas[ind] += val;
        for(ind >>= 1; ind > 0; ind >>= 1)
            mas[ind] = std::min(mas[ind<<1], mas[(ind<<1)+1]);
    }

    val_t query(int l, int r) {
        if(l >= r) return inf;
        l += size; r += size-1;
        val_t ans = std::min(mas[l], mas[r]);
        for(; l < r; l >>= 1, r >>= 1) {
            if((l&1) == 0 && (l+1) < r)
                ans = std::min(ans, mas[l+1]);
            if((r&1) == 1 && (r-1) > l)
                ans = std::min(ans, mas[r-1]);
        }
        return ans;
    }
};
```

```
//RMQ with interval modification support
struct rmq_rm_t {
    val_t ans[size<<1], add[size<<1];

    rmq_rm_t() { }
    rmq_rm_t(val_t* a) {
        memset(add, 0, sizeof(add)); //! you may need something else here
        memcpy(ans+size, a, sizeof(val_t)*size);
        for(int i = size-1; i > 0; i--)
            ans[i] = std::min(ans[i<<1], ans[(i<<1)+1]);
    }

    void modify(int l, int r, val_t val) {
        if(l >= r) return;
        l += size; r += size-1;
        add[l] += val;
        if(l < r) add[r] += val;
        while(l > 0) {
            if((l&1) == 0 && (l+1) < r)
                add[l+1] += val;
            if((r&1) == 1 && (r-1) > l)
                add[r-1] += val;
            l >>= 1; r >>= 1;
            ans[l] = std::min(ans[l<<1]+add[l<<1], ans[(l<<1)+1]+add[(l<<1)+1]);
            ans[r] = std::min(ans[r<<1]+add[r<<1], ans[(r<<1)+1]+add[(r<<1)+1]);
        }
    }
    val_t query(int l, int r) {
        if(l >= r) return inf;
        l += size; r += size-1;
        val_t lans = ans[l]+add[l], rans = ans[r]+add[r];
        while(l > 0) {
            if((l&1) == 0 && (l+1) < r)
                lans = std::min(lans, ans[l+1]+add[l+1]);
            if((r&1) == 1 && (r-1) > l)
                rans = std::min(rans, ans[r-1]+add[r-1]);
            l >>= 1; r >>= 1;
            lans += add[l]; rans += add[r];
        }
        return std::min(lans, rans);
    }
};


// Fenvik tree
struct fenv_t {
    val_t mas[size];

    fenv_t() { }
    fenv_t(int sz, val_t* a) {
        int i;
        for(mas[0] = 0, i = 0; i < sz; i++) mas[i+1] = mas[i] + a[i];
        for(; i < size-1; i++) mas[i+1] = mas[i];
        for(i = size-1; i > 0; i--) mas[i] -= mas[i&(i-1)];
    }

    void modify(int ind, val_t val) {
        for(ind++; ind < size; ind = (ind<<1) - (ind&(ind-1)))
            mas[ind] += val;
    }
```

```
    val_t query(int r) {
        val_t ans = val_t();
        for(; r > 0; r = r&(r-1))
            ans += mas[r];
        return ans;
    }

    val_t query(int l, int r) {
        return query(r) - query(l);
    }
};
```

```
// Cartesian tree implementation

typedef int fkey_t;
typedef int skey_t;

struct node {
    fkey_t pk;
    skey_t sk;
    int sz;
    node *left, *right;

    int val, min_val, max_val;
    int64 sum_val;

    int sum, rev;

    void recalc();
    int size() {
        return this == NULL ? 0 : sz;
    }
};

void node::recalc() {
    sz = 1 + left->size() + right->size();
    sum_val = min_val = max_val = (val += sum);
    if(left != NULL) {
        left->sum += sum;
        left->rev ^= rev;
        min_val = std::min(min_val, left->min_val + left->sum);
        max_val = std::max(max_val, left->max_val + left->sum);
        sum_val += left->sum_val + int64(left->sum) * left->size();
    }
    if(right != NULL) {
        right->sum += sum;
        right->rev ^= rev;
        min_val = std::min(min_val, right->min_val + right->sum);
        max_val = std::max(max_val, right->max_val + right->sum);
        sum_val += right->sum_val + int64(right->sum) * right->size();
    }
    if(rev) std::swap(left, right);
    sum = rev = 0;
}

node *root = NULL;

node* new_node(const fkey_t& pk, const skey_t& sk) {
    node* ptr = new node();
    ptr->pk = pk; ptr->sk = sk;
```

```
    ptr->left = ptr->right = NULL;
    ptr->sz = 1;
    ptr->sum_val = ptr->val = ptr->max_val = ptr->min_val = 0;
    ptr->sum = ptr->rev = 0;
    return ptr;
}

void split(node* v, const fkey_t& pk, node* &left, node* &right) {
    if(v == NULL) {
        left = right = NULL;
        return;
    }
    v->recalc();
    if(v->pk <= pk) {
        split(v->right, pk, v->right, right);
        left = v;
    } else {
        split(v->left, pk, left, v->left);
        right = v;
    }
    v->recalc();
}
// Not intended to be used with delayed operations.
// Probably you just need to add v->recalc().
node* add(node* &v, const fkey_t& pk, const skey_t& sk) {
    if(v == NULL || v->sk >= sk) {
        node* cur = new_node(pk, sk);
        split(v, pk, cur->left, cur->right);
        cur->recalc();
        return v = cur;
    }
    node* ptr = (pk <= v->pk) ?
        add(v->left, pk, sk) : add(v->right, pk, sk);
    v->recalc();
    return ptr;
}
// You MUST manually call left->recalc() and right->recalc()
// while callilng this function !!!
node* merge(node* left, node* right) {
    if(left == NULL) return right;
    if(right == NULL) return left;
    if(left->sk <= right->sk) {
        if(left->right != NULL) left->right->recalc();
        left->right = merge(left->right, right);
        left->recalc();
        return left;
    } else {
        if(right->left != NULL) right->left->recalc();
        right->left = merge(left, right->left);
        right->recalc();
        return right;
    }
}

void erase(node* &v, const fkey_t& pk) {
    if(v == NULL) return;
    v->recalc();
    if(v->pk == pk) {
        node* tmp = merge(v->left, v->right);
        delete v;
        v = tmp;
        return;
```

```
    }
    if(pk <= v->pk) {
        erase(v->left, pk);
        v->recalc();
    }
    else {
        erase(v->right, pk);
        v->recalc();
    }
}

    node* search(node* v, const fkey_t& pk) {
        if(v == NULL)
            return NULL;
        v->recalc();
        if(v->pk == pk) return v;
        return (pk <= v->pk) ?
            search(v->left, pk) : search(v->right, pk);
    }

void clear(node* &v) {
    if(v == NULL) return;
    clear(v->left);
    clear(v->right);
    delete v;
    v = NULL;
}

node* begin() {
    if(root == NULL) return NULL;
    node* v = root;
    for(v->recalc(); v->left != NULL; v->recalc())
        v = v->left;
    return v;
}

int count_less(const fkey_t& pk) {
    int ans = 0;
    for(node* v = root; v != NULL;) {
        v->recalc();
        if(pk <= v->pk) v = v->left;
        else {
            ans += v->left->size() + 1;
            v = v->right;
        }
    }
    return ans;
}

node* get_kth(int k) { //zero based
    for(node* v = root; v != NULL ;) {
        v->recalc();
        if(v->left->size() == k) return v;
        else if(v->left->size() > k) v = v->left;
        else {
            k -= v->left->size()+1;
            v = v->right;
        }
    }
    return NULL;
}
```

```
void cut_k(node* v, int k, node* &left, node* &right) { // cut k nodes from left
    if(v == NULL) {
        left = right = NULL;
        return;
    }
    v->recalc();
    if(v->left->size() >= k) {
        right = v;
        cut_k(v->left, k, left, right->left);
    } else {
        left = v;
        cut_k(v->right, k - (v->left->size()+1), left->right, right);
    }
    v->recalc();
}
```

```
//Ukkonen algorithm

#define maxa (53)
#define maxl (1 << 17)
inline int numc(char c) { return c <= 'Z' ? c-'A' : c-'a' + 26; }
char s[maxl];

struct node {
    int k, p, suff;
    int next[maxa];
} tree[maxl*2];
int cnt, root, aux;

int new_node(int k, int p) {
    memset(tree+cnt, -1, sizeof(node));
    tree[cnt].k = k; tree[cnt].p = p;
    return cnt++;
}

bool test_and_split(int v, int k, int p, char c, int& r) {
    if(k >= p) { r = v; return tree[v].next[numc(c)] == -1; }
    int nx = tree[v].next[numc(s[k])];
    char wc = s[tree[nx].k + p-k];
    if(wc == c) { r = v; return false; }
    r = new_node(tree[nx].k, tree[nx].k + p-k);
    tree[v].next[numc(s[k])] = r;
    tree[r].next[numc(wc)] = nx;
    tree[nx].k += p-k;
    return true;
}

void canonize(int& v, int& k, int p) {
    while(k < p) {
        int nx = tree[v].next[numc(s[k])];
        if(tree[nx].p-tree[nx].k > p-k) break;
        k += tree[nx].p-tree[nx].k;
        v = nx;
    }
}

void make_tree() { // You MUST ensure to add '$' at the end of the string
    int oldr, v, k, p, i, len = strlen(s), r;
    cnt = 0;
    aux = new_node(-1, -1);
    v = root = new_node(-1, 0);
```

```
    for(i = 0; i < maxa; i++) tree[aux].next[i] = root;
    tree[root].suff = aux;
    for(i = k = 0; i < len; i++) {
        for(oldr = root; test_and_split(v, k, i, s[i], r); canonize(v, k, i)) {
            tree[r].next[numc(s[i])] = new_node(i, len);
            if(oldr != root) tree[oldr].suff = r;
            oldr = r;
            v = tree[v].suff;
        }
        if(oldr != root) tree[oldr].suff = r;
        canonize(v, k, i+1);
    }
}

bool find_str(const char* st) {
    for(int v = root, k = 0; *st; st++) {
        if(k >= tree[v].p) {
            if(tree[v].next[numc(*st)] == -1) return false;
            v = tree[v].next[numc(*st)];
            k = tree[v].k;
        }
        if(s[k++] != *st) return false;
    }
    return true;
}
```

```
//Aho-Corasik algorithm

#define maxa (53)
#define maxn (3 << 16)
inline int numc(char c) { return c <= 'Z' ? c-'A' : c-'a'+26; }

struct node {
    int end, suff, next[maxa];
} nodes[maxn];
int cnt, root;
int q[maxn];

int new_node() {
    memset(nodes+cnt, -1, sizeof(node));
    return cnt++;
}

std::vector<int> same_words[1 << 10];

void add_word(const char* st, int num) {
    int v;
    for(v = root; *st; st++) {
        if(nodes[v].next[numc(*st)] == -1)
            nodes[v].next[numc(*st)] = new_node();
        v = nodes[v].next[numc(*st)];
    }
    if(nodes[v].end == -1) nodes[v].end = num;
    same_words[nodes[v].end].push_back(num);
}

void aho_corasik() { //(only suffix links) automaton construction
    nodes[root].suff = root;
    int l, r = 0, i, u, nx;
    for(i = 0; i < maxa; i++)
```

```
        if((nx = nodes[root].next[i]) != -1) {
            nodes[nx].suff = root;
            q[r++] = nx;
        } else
            nodes[root].next[i] = root;
    for(l = 0; l < r; l++) {
        for(i = 0; i < maxa; i++)
            if((nx = nodes[q[l]].next[i]) != -1) {
                for(u = nodes[q[l]].suff; nodes[u].next[i] == -1; u = nodes[u].suff);
                nodes[nx].suff = nodes[u].next[i];
                q[r++] = nx;
            }
    }
}

int was[maxn], pres[1 << 10];
void search(const char* st) {
    memset(was, 0, sizeof(int)*cnt);
    for(int v = root; *st; st++) {
        for(; nodes[v].next[numc(*st)] == -1; v = nodes[v].suff);
        v = nodes[v].next[numc(*st)];
        for(int u = v; u != root; u = nodes[u].suff)
            if(was[u]++) break;
            else if(nodes[v].end != -1)
                for(int i = 0; i < same_words[nodes[v].end].size(); i++)
                    pres[same_words[nodes[v].end][i]] = 1;
    }
}
```

```
// Construction of suffix automata in O(n)

#define maxl (1 << 17)
#define maxn (1 << 18)
#define maxa 53

char s[maxl];
inline int char_num(char c) { return c <= 'Z' ? (c - 'A') : (c - 'a' + 26); }

struct node {
    int next[maxa], suff, len, end;
} nodes[maxn];
int root, cnt;

inline int new_node(int len) {
    memset(nodes + cnt, -1, sizeof(node));
    nodes[cnt].len = len;
    nodes[cnt].end = 0;
    return cnt++;
}

void extend(char c, int& last) {
    int nlast = new_node(nodes[last].len + 1), p;
    for(p = last; p >= 0 && nodes[p].next[char_num(c)] == -1; p = nodes[p].suff)
        nodes[p].next[char_num(c)] = nlast;
    last = nlast;
    if(p < 0) {
        nodes[nlast].suff = root;
        return;
    }
    int q = nodes[p].next[char_num(c)];
    if(nodes[q].len == nodes[p].len + 1) {
```

```
        nodes[nlast].suff = q;
        return;
    }
    int nq = new_node(nodes[p].len + 1);
    memcpy(nodes[nq].next, nodes[q].next, sizeof(nodes[q].next));
    nodes[nlast].suff = nq;
    nodes[nq].suff = nodes[q].suff;
    nodes[q].suff = nq;
    for(; p >= 0 && nodes[p].next[char_num(c)] == q; p = nodes[p].suff)
        nodes[p].next[char_num(c)] = nq;
}

void create_automata() {
    int l = strlen(s), i, last;
    cnt = 0;
    last = root = new_node(0);
    for(i = 0; i < l; i++)
        extend(s[i], last);
    for(i = last; i >= 0; i = nodes[i].suff)
        nodes[i].end = 1;
}
```

```
//Caclulation of z-funcion
void calc_z(const char* st, int* z) {
    int i, j=0, r = 0, l;
    z[0] = l = strlen(st);
    for(i = 1; i < l; i++) {
        z[i] = std::max(std::min(r-i, z[i-j]), 0);
        for(; i+z[i] < l && st[z[i]] == st[i+z[i]]; z[i]++);
        if(i+z[i] > r) {
            r = i+z[i];
            j = i;
        }
    }
}

//Lyndon decomposition and minimal cyclical shift search in O(n)
std::string min_cyclic_shift (std::string s) {
    s += s;
    int n = (int) s.length();
    int i=0, ans=0;
    while (i < n/2) {
        ans = i;
        int j=i+1, k=i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                ++k;
            ++j;
        }
        while (i <= k) i += j - k;
    }
    return s.substr (ans, n/2);
}
```

```
//Suffix massive construction

char st[maxl];

int ind[maxl], nind[maxl], val[20][maxl], cnt[maxl], cc, log2, l;
```

```
void build_mass() {
    int i, j;
    l = strlen(st);
    for(log2 = 0; (1 << log2) < l; log2++);
    memset(cnt, 0, sizeof(cnt));
    memset(val, 0, sizeof(val));
    for(i = 0; i < l; i++) cnt[st[i]+1]++;
    for(i = 0; i < 128; i++) cnt[i+1] += cnt[i];
    for(i = 0; i < l; i++)
        ind[cnt[st[i]]++] = i;
    for(i = 1; i < l; i++)
        val[0][ind[i]] = val[0][ind[i-1]] + (st[ind[i]] != st[ind[i-1]]);

    for(j = 0; j < log2; j++) {
        cnt[0] = 0; cnt[cc = 1] = 1;
        for(i = 1; i < l; i++)
            if(val[j][ind[i]] == val[j][ind[i-1]]) cnt[cc]++;
            else cnt[++cc] = 1;
        for(i = 0; i < cc; i++) cnt[i+1] += cnt[i];
        int sz = (1 << j);
        for(i = 0; i < l; i++)
            nind[cnt[val[j][(ind[i]-sz+l) % l]]++] = (ind[i]-sz+l) % l;
        memcpy(ind, nind, sizeof(int)*l);
        for(i = 1; i < l; i++)
            val[j+1][ind[i]] = val[j+1][ind[i-1]] +
                (val[j][ind[i]] != val[j][ind[i-1]] || val[j][(ind[i]+sz)%l] != val[j][(ind[i-1]+sz)%l]);
    }
}

int lcp(int i, int j) {
    int ans = 0, k;
    for(k = log2-1; k >= 0; k--)
        if(val[k][i] == val[k][j]) {
            i = (i + (1 << k)) % l;
            j = (j + (1 << k)) % l;
            ans += (1 << k);
        }
    return ans;
}
```

```
//Min-cost max-flow alogrithm

#define maxn (1 << 7)
#define maxe (1 << 14)
#define inf 0x3f3f3f3f

typedef int dist_t;

struct edge {
    int dest, back, f, c;
    dist_t w;
} edges[maxe];
int cnt, n;

std::vector<int> g[maxn];

void add_edge(int u, int v, int c, dist_t w) {
    edges[cnt].dest = v;
    edges[cnt].back = cnt+1;
    edges[cnt].f = 0;
```

```
    edges[cnt].c = c;
    edges[cnt].w = w;
    g[u].push_back(cnt);
    edges[cnt+1].dest = u;
    edges[cnt+1].back = cnt;
    edges[cnt+1].f = 0;
    edges[cnt+1].c = 0;
    edges[cnt+1].w = -w;
    g[v].push_back(cnt+1);
    cnt += 2;
}

int was[maxn], prev[maxn], r[maxn];
dist_t dist[maxn], phi[maxn];

bool dijk(int s, int t) {
    int i, j;
    for(i = 0; i < n; i++) dist[i] = inf;
    memset(was, 0, sizeof(int)*n);
    memset(r, 0, sizeof(int)*n);
    dist[s] = 0;
    r[s] = inf;
    for(i = 0; i < n; i++) {
        int mv = -1;
        for(j = 0; j < n; j++)
            if(!was[j] && (mv == -1 || dist[mv] > dist[j])) mv = j;
        if(mv == -1 || dist[mv] == inf) break;
        was[mv] = 1;
        for(j = 0; j < g[mv].size(); j++) {
            edge& e = edges[g[mv][j]];
            if(!was[e.dest] && e.f < e.c && dist[e.dest] > dist[mv] + e.w + phi[mv]-phi[e.dest]) {
                dist[e.dest] = dist[mv] + e.w + phi[mv]-phi[e.dest];
                r[e.dest] = std::min(r[mv], e.c-e.f);
                prev[e.dest] = e.back;
            }
        }
    }
    return r[t] > 0;
}

dist_t aug(int s, int t) {
    int rr = r[t];
    dist_t ans = 0;
    while(s != t) {
        edge& e = edges[prev[t]];
        ans -= e.w*rr;
        e.f -= rr;
        edges[e.back].f += rr;
        t = e.dest;
    }
    return ans;
}

int flow;
dist_t cost;
void min_cost_max_flow(int s, int t) {
    int i;
    for(i = 0; i < n; i++) phi[i] = 0;
    cost = flow = 0;
    while(dijk(s, t)) {
        flow += r[t];
        cost += aug(s, t);
```

```
        for(i = 0; i < n; i++) phi[i] += dist[i];
    }
}
```

// Hungarian algorithm

```
#define inf 0x3f3f3f3f
int u[maxn], v[maxn], p[maxn], prev[maxn], minv[maxn], nw[maxn];
int c[maxn][maxn];
int n;

int min_cost_assignement() {
    memset(u, 0, sizeof(int)*(n+1));
    memset(v, 0, sizeof(int)*(n+1));
    memset(p, 0, sizeof(int)*(n+1));
    for(int i = 1; i <= n; i++) {
        int j0 = 0, j1 = -1;
        p[0] = i;
        memset(minv, 0x3f, sizeof(int)*(n+1));
        minv[0] = 0;
        memset(nw, 0, sizeof(int)*(n+1));
        do {
            nw[j0] = 1;
            int i0 = p[j0], del = inf;
            for(int j = 1; j <= n; j++)
                if(nw[j] == 0) {
                    int cc = c[i0-1][j-1] - u[i0] - v[j];
                    if(cc < minv[j]) {
                        minv[j] = cc;
                        prev[j] = j0;
                    }
                    if(minv[j] < del) {
                        del = minv[j];
                        j1 = j;
                    }
                }
            for(int j = 0; j <= n; j++)
                if(nw[j] == 1) {
                    u[p[j]] += del;
                    v[j] = v[i] - del;
                } else
                    minv[j] -= del;
            j0 = j1;
        } while(p[j0] != 0);
        do {
            j1 = prev[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while(j0 != 0);
    }
    int ans = 0;
    for(int i = 1; i <= n; i++) ans += c[p[i]-1][i-1];
    return ans;
}
```

//Bridges, articulations points and blocks

```
#define inf 0x3f3f3f3f
#define maxn (1 << 10)
#define maxe (maxn*maxn)
```

```
std::vector<int> g[maxn];
int low[maxn], was[maxn], num[maxn], cur;
int n;

std::vector<int> is_bridge[maxn], back[maxn], comp[maxn];
std::pair<int, int> stck[maxe];
int ssize, comp_cnt, is_art[maxn];

void add_biedge(int u, int v) {
    back[u].push_back(g[v].size());
    back[v].push_back(g[u].size());
    g[u].push_back(v);
    g[v].push_back(u);
    comp[u].push_back(-1);
    comp[v].push_back(-1);
    is_bridge[u].push_back(0);
    is_bridge[v].push_back(0);
}
void clear() {
    for(int i = 0; i < n; i++)
        g[i].clear(), is_bridge[i].clear(), back[i].clear(), comp[i].clear();
    memset(num, -1, sizeof(num));
    memset(was, 0, sizeof(was));
    cur = comp_cnt = ssize = 0;
}
//Searching for articulation points
void dfs(int v, int par) {
    was[v] = 1;
    num[v] = low[v] = cur++;
    int d = 0, maxlo = -1;
    for(int i = 0; i < g[v].size(); i++) {
        if(g[v][i] == par) continue;
        if(was[g[v][i]]) {
            low[v] = std::min(low[v], num[g[v][i]]);
            continue;
        }
        dfs(g[v][i], v); d++;
        maxlo = std::max(maxlo, low[g[v][i]]);
        low[v] = std::min(low[v], low[g[v][i]]);
    }
    if(par >= 0) is_art[v] = (maxlo >= num[v]);
    else is_art[v] = (d > 1);
}
//Searching for bridges
void dfs(int v, int par) {
    was[v] = 1;
    num[v] = low[v] = cur++;
    for(int i = 0; i < g[v].size(); i++) {
        is_bridge[v][i] = 0;
        if(g[v][i] == par) continue;
        if(was[g[v][i]]) {
            low[v] = std::min(low[v], num[g[v][i]]);
            continue;
        }
        dfs(g[v][i], v);
        low[v] = std::min(low[v], low[g[v][i]]);
        if(low[g[v][i]] > num[v]) is_bridge[v][i] = 1;
    }
}
//Searching for biconnected components
void release(int u, int v) {
```

```
    while(ssize-- > 0) {
        int i = stck[ssize].first, j = stck[ssize].second;
        comp[i][j] = comp[g[i][j]][back[i][j]] = comp_cnt;
        if((i == u && g[i][j] == v) || (i == v && g[i][j] == u)) break;
    }
    comp_cnt++;
}
void dfs(int v, int par) {
    was[v] = 1;
    num[v] = low[v] = cur++;
    for(int i = 0; i < g[v].size(); i++) {
        if(g[v][i] == par) continue;
        if(num[g[v][i]] < num[v]) stck[ssize++] = std::make_pair(v, i);
        if(was[g[v][i]]) {
            low[v] = std::min(low[v], num[g[v][i]]);
            continue;
        }
        dfs(g[v][i], v);
        low[v] = std::min(low[v], low[g[v][i]]);
        if(low[g[v][i]] >= num[v])
            release(v, g[v][i]);
    }
}
```

//Searching for minmal cut

```
#define inf 0x3f3f3f3f
#define maxn (1 << 7)

int n;
int w[maxn][maxn];
int v[maxn], A[maxn], col[maxn];

int mincut()
{
    int res = inf, i, j, k, u, s, t, last;
    for (i=0; i<n; i++) v[i] = i;
    while (n >= 2 && res > 0)
    {
        for (i=0; i<n; i++) A[v[i]] = 0;
        for (i=0; i<n; i++) col[v[i]] = 0;
        u = v[0]; col[u] = 1;
        for (i=0; i<n; i++) A[v[i]] = w[u][v[i]];
        s = t = u;
        for (k=1; k<n; k++)
        {
            s = t; t = -1;
            for (i=0; i<n; i++) if (!col[v[i]] &&
                (t == -1 || A[t] < A[v[i]])) t = v[i];
            last = A[t];
            col[t] = 1;
            for (i=0; i<n; i++) A[v[i]] += w[t][v[i]];
        }
        for (i=0; i<n; i++)
        {
            w[s][v[i]] += w[t][v[i]];
            w[v[i]][s] += w[t][v[i]];
        }
        for (i=0; i<n; i++) if (v[i] == t) break;
        for (; i<n; i++) v[i] = v[i+1]; n--;
        if (last < res) res = last;
```

```
    }
    return res;
}
```

```
//Setting the line up in the bitset
inline const bool test(dword *arr, const int &x) {
    return (arr[x>>5] >> (x&31)) & 1;
}
inline void setBit(dword *arr, const int &x) {
    arr[x>>5] |= (1<<(x&31));
}

inline const dword getMask(const int &l, const int &r) {
    return (r<32 ? (dword(1)<<r) : dword(0)) - (dword(1)<<l);
}
inline void setLine(dword *arr, int l, int r) {
    if (l>r) return;
    int li = (l>>5);
    int ri = (r>>5);
    if (li==ri) {
        arr[li] |= getMask(l - (li<<5), r+1 - (ri<<5));
        return;
    }
    arr[li] |= getMask(l - (li<<5), 32);
    arr[ri] |= getMask(0, r+1 - (ri<<5));
    for (int i = li+1; i<=ri-1; i++) arr[i] = dword(-1);
}

//Geometry
//--------------------------------

//integer point
struct Point {
    int x, y;
    inline Point() : x(0), y(0) {}
    inline Point(int _x, int _y) : x(_x), y(_y) {}

    inline int operator! () const { return x*x + y*y; }
    inline const Point operator+ (const Point &b) const {
        return Point(x+b.x, y+b.y);
    }
    inline const Point operator- (const Point &b) const {
        return Point(x-b.x, y-b.y);
    }
    inline const Point operator* (int b) const {
        return Point(x*b, y*b);
    }
    inline bool operator== (const Point &b) const {
        return x==b.x && y==b.y;
    }
    inline bool half() const {
        return (y<0 || (y==0 && x<0));
    }
    inline bool operator< (const Point &b) const;
};
inline int vect (const Point &a, const Point &b) {
    return a.x*b.y - a.y*b.x;
}
inline int scal (const Point &a, const Point &b) {
    return a.x*b.x + a.y*b.y;
```

```
}
// comparing by polar angle
// <*this> and <b> MUST BE NONZERO
inline bool Point::operator< (const Point &b) const {
    int th = half();
    int bh = b.half();
    if (th ^ bh) return th < bh;
    int pv = vect(*this, b);
    return pv > 0;
}
// decides whether <p> is inside the oriented angle (<a>-<b>) (including bounds)
// both <a> and <b> are nonzero
bool inSector(const Point &a, const Point &b, const Point &p) {
    int vab = vect(a, b);
    if (!vab && (scal(a, b) > 0)) if (scal(a, p) < 0) return false;
    if (vab >= 0) return (vect(a, p) >= 0 && vect(p, b) >= 0);
    else          return (vect(a, p) >= 0 || vect(p, b) >= 0);
}
inline bool CrossSegs(int l1, int r1, int l2, int r2) {
    if (l1 > r1) std::swap(l1, r1);
    if (l2 > r2) std::swap(l2, r2);
    return !(l1>r2 || r1<l2);
}
// crosses two closed line segments <p1>-<p2> and {p3>-<p4>
bool CrossLineLine(const Point &p1, const Point &p2, const Point &p3, const Point &p4) {
    int a11 = p2.x - p1.x;
    int a12 = p3.x - p4.x;
    int a21 = p2.y - p1.y;
    int a22 = p3.y - p4.y;
    int b1 = p3.x - p1.x;
    int b2 = p3.y - p1.y;
    int det = a11*a22 - a12*a21;
    int detu = b1 *a22 - a12*b2 ;
    int detv = a11*b2 - b1 *a21;
    if (det == 0) {
        if (detu || detv) return false;
        if (p1.x != p2.x || p3.x != p4.x) return CrossSegs(p1.x, p2.x, p3.x, p4.x);
        if (p1.y != p2.y || p3.y != p4.y) return CrossSegs(p1.y, p2.y, p3.y, p4.y);
        return (p1 == p3);
    }
    if (det < 0) {
        det = -det;
        detu = -detu;
        detv = -detv;
    }
    return (detu>=0 && detu<=det && detv>=0 && detv<=det);
}
//determines whether point <p> lies on the closed segment <a>-<b>
inline bool onLine(const Point &p, const Point &a, const Point &b) {
    if (vect(p-a, b-a)) return false;
    if (a==b) return p==a;
    return (scal(p-a, b-a)>=0 && scal(p-b, a-b)>=0);
}
// determines whether a point <p> is inside the triangle <a>-<b>-<c>
// does not work for triangles with zero area
inline int uabs(int a) { return (a<0 ? -a : a); }
inline bool inTriangle(const Point &p, const Point &a, const Point &b, const Point &c, bool strict = false) {
    int t, tv;

    tv = uabs(vect(c-a, b-a));

    t = uabs(vect(p-a, b-a));
```

```
    if (strict && !t) return false;
    tv -= t;

    t = uabs(vect(p-b, c-b));
    if (strict && !t) return false;
    tv -= t;

    t = uabs(vect(p-c, a-c));
    if (strict && !t) return false;
    tv -= t;

    return tv >= 0;
}
// determines whether point <p> lies inside polygon <arr[0], ..., arr[n]>
// no self-crossings!  may have equal points arr[0] = arr[n]
bool inPolygon(const Point &p, int n, const Point *arr, bool strict = false) {
    // if lies on the border
    for (int i = 0; i<n; i++)
        if (onLine(p, arr[i], arr[i+1]))
            return !strict;
    // BIG PRIMES: 1061109589 / 1061109601
    Point spot = p + Point(15013, 15017);
    int cnt = 0;
    for (int i = 0; i<n; i++)
        if (CrossLineLine(spot, p, arr[i], arr[i+1]))
            cnt ^= 1;
    return bool(cnt);
}
// Graham's convex hull
// changes order of points!  no equal points allowed!  must have nonzero area
Point hctr;
bool cmpHull(const Point &a, const Point &b) {
    Point ad = a - hctr;
    Point bd = b - hctr;
    int tv = vect(ad, bd);
    if (tv) return tv > 0;
    return !ad < !bd;
}
void ConvexHull(int n, Point *arr, int &k, Point *res, bool strict = true) {
    int i;
    int best = 0;
    for (i = 1; i<n; i++) {
        if (arr[i].x < arr[best].x) best = i;
        if (arr[i].x == arr[best].x && arr[i].y < arr[best].y) best = i;
    }

    std::swap(arr[best], arr[0]);
    hctr = arr[0];
    std::sort(arr+1, arr+n, cmpHull);

    k = 0;
    res[k++] = arr[0];
    res[k++] = arr[1];

    for (i = 2; i<n; i++) {
        if (strict) while (k>=2 && vect(res[k-1]-res[k-2], arr[i]-res[k-2]) <= 0) k--;
        if (!strict) while (k>=2 && vect(res[k-1]-res[k-2], arr[i]-res[k-2]) < 0) k--;
        res[k++] = arr[i];
    }
    if (!strict) {
        k--;
        for (i = n-1; i>0; i--) {
```

```
            res[k++] = arr[i];
            if (vect(arr[i]−arr[0], arr[i−1]−arr[0]) != 0) break;
        }
    }
}
//real point

//Line crosses Line (infinite)
bool CrossLineLine(const Point &a1, const Point &b1, const Point &a2, const Point &b2, Point &res) {
    real a11 = b1.x − a1.x;
    real a12 = a2.x − b2.x;
    real a21 = b1.y − a1.y;
    real a22 = a2.y − b2.y;
    real xb1 = a2.x − a1.x;
    real xb2 = a2.y − a1.y;
    real det  = a11*a22 − a12*a21;
    real detu = xb1*a22 − a12*xb2;
    real detv = a11*xb2 − xb1*a21;
    if (fabs(det) < EPS) return false;
    detu /= det;
    detv /= det;
    Point c = a1 + (a2−a1)*detu;
    res = c;
    return true;
}
//Line crosses Circle (infinite)
bool CrossLineCircle(const Point &la, const Point &lb, const Point &cc, real cr, Point &res1, Point &res2) {
    Point st = la − cc;
    Point dir = lb − la;

    real qa = scal(dir, dir);
    real qb = 2.0 * scal(st, dir);
    real qc = scal(st, st) − cr*cr;
    real qd = qb*qb − 4.0*qa*qc;
    if (qd < −EPS) return false;
    if (qd < 0.0) qd = 0.0;
    qd = sqrt(qd);

    real x1 = (−qb − qd) / (2.0 * qa);
    real x2 = (−qb + qd) / (2.0 * qa);

    res1 = la + dir*x1;
    res2 = la + dir*x2;
    return true;
}
//Circle crosses Circle
bool CrossCircleCircle(const Point &c1, real r1, const Point &c2, real r2, Point &res1, Point &res2) {
    real la = 2.0 * (c2.x − c1.x);
    real lb = 2.0 * (c2.y − c1.y);
    real lc = sqr(c1.x)−sqr(c2.x) + sqr(c1.y)−sqr(c2.y) + sqr(r2)−sqr(r1);

    if (la*la + lb*lb < EPS) return false;

    Point a, b;
    if (fabs(la) > fabs(lb)) {
        a = Point(−lc/la, 0.0);
        b = Point(−(lb+lc)/la, 1.0);
    }
    else {
        a = Point(0.0, −lc/lb);
        b = Point(1.0, −(lc+la)/lb);
    }
```

```
    return CrossLineCircle(a, b, c1, r1, res1, res2);
}
//Gauss
//--------------------
int n, m, r;
real matr[SIZE][SIZE];
int adr[SIZE];
bool used[SIZE];
real sol[SIZE];
//Gauss algorithm
void Gauss() {
    int i, j, u;
    r = 0;
    memset(used, 0, sizeof(used));
    for (i = 0; i<=n; i++) {
        int best = -1;
        for (j = r; j<m; j++) if (best<0 || fabs(matr[j][i]) > fabs(matr[best][i])) best = j;
        if (best < 0) break;

        for (u = 0; u<=n; u++) std::swap(matr[best][u], matr[r][u]);
        if (fabs(matr[r][i]) < EPS) continue;

        for (u = n; u>=i; u--) matr[r][u] /= matr[r][i];

        for (j = 0; j<m; j++) if (j != r) {
            real coef = matr[j][i];
            for (u = i; u<=n; u++) matr[j][u] -= coef*matr[r][u];
        }

        used[i] = true;
        adr[r++] = i;
    }
}
//Getting solution
bool GetSolution() {
    int i, j;
    memset(sol, 0, sizeof(sol));
    if (used[n]) return false;
    sol[n] = -1.0; //MUST BE SO!
    for (i = 0; i<n; i++) if (!used[i]) sol[i] = rand() / 32768.0; //free variables
    for (i = r-1; i>=0; i--)
        for (j = adr[i]+1; j<=n; j++)
            sol[adr[i]] -= sol[j]*matr[i][j];
    return true;
}
//Integer polynom dividing
//--------------------
struct Poly {
    int deg;
    int arr[SIZE];
    //Divides poly by poly
    Poly Divide(const Poly &b) {
        Poly ost = *this;
        Poly bad; bad.deg = -1; //Bad poly with negative degree
        Poly res; res.deg = deg - b.deg;
        int i, j;
        for (i = deg-b.deg; i>=0; i--) {
            if (ost.arr[i+b.deg] % b.arr[b.deg]) return bad;
            int coef = ost.arr[i+b.deg] / b.arr[b.deg];
            res.arr[i] = coef;
            for (j = 0; j<=b.deg; j++)
                ost.arr[i+j] -= coef*b.arr[j];
```

```
        }
        //checking for even division; delete if reminder is needed
        for (i = 0; i<b.deg; i++) if (ost.arr[i]) return bad;
        return res;
    }
};
//LCA
//--------------------
//calculating LCA of two nodes: <a> and <b>
//needs (N*logN)*sizeof(int) memory
int n;
int hgt[SIZE];
int father[LOGS][SIZE];
void LCAInit() {
    //init array <hgt> - heights of nodes
    //init array <father[0]> - fathers of nodes (-1 if none)
    int i, j;
    for (i = 1; i<LOGS; i++) {
        for (j = 0; j<n; j++) {
            if (father[i-1][j] < 0) father[i][j] = -1;
            else father[i][j] = father[i-1][father[i-1][j]];
        }
    }
}
int LCA(int a, int b) {
    if (hgt[a] < hgt[b]) std::swap(a, b);
    int i;
    for (i = LOGS-1; i>=0; i--)
        if (father[i][a] >= 0 && hgt[father[i][a]] >= hgt[b])
            a = father[i][a];
    for (i = LOGS-1; i>=0; i--)
        if (father[i][a] != father[i][b]) {
            a = father[i][a];
            b = father[i][b];
        }
    if (a != b) a = father[0][a];
    return a;
}
```

```
//Some useful stuff for integration

#define inf 0x3f3f3f3f

typedef long long int64;
typedef double real;

#define CUT 1000000
#define MINP 357
#define MAXP 200000
#define MULT 24
#define DEG 8

int inarr[5];
real func(real x) {
    real res = 0.0;
    for (int i = 4; i>=0; i--) {
        res *= x;
        res += inarr[i];
    }
    return 1.0 / res;
}
```

```
int ql;
real quadx[128];
real quadc[128];

int sign;

struct Line {
    real l, r;
    real integ;

    inline void calc() {
        integ = 0.0;
        for (int i = 0; i<ql; i++) integ += quadc[i] * func(l*(1.0−quadx[i]) + r*quadx[i]);
        integ *= r−l;
    }

    inline Line(real _l = 0.0, real _r = 1.0) : l(_l), r(_r) {
        calc();
    }

    inline void Divide(Line &a, Line &b) const {
        real m = (l+r)/2.0;
        a.l = l;
        a.r = m;
        b.l = m;
        b.r = r;
        a.calc();
        b.calc();
    }
    inline bool operator< (const Line &b) const {
        return integ > b.integ;
    }
};
typedef std::multiset<Line> lSet;

lSet lines;

real ans;

int64 arr[128];
int64 narr[128];
int64 coef;
void BuildCool(int deg) {
    ql = deg+1;
    int i, j, u;
    for (i = 0; i<=deg; i++) {
        memset(arr, 0, sizeof(arr));
        arr[0] = 1;
        for (j = 0; j<=deg; j++) if (i != j) {
            memset(narr, 0, sizeof(narr));
            for (u = 0; u<=deg; u++) narr[u] = (u ? arr[u−1] : 0) − j*arr[u];
            memcpy(arr, narr, sizeof(arr));
        }

        coef = 0;
        for (j = deg; j>=0; j−−) {
            coef *= i;
            coef += arr[j];
        }

        quadx[i] = i;
```

```
        quadc[i] = 0;
        for (j = 0; j<=deg; j++)
            quadc[i] += arr[j] * pow(real(deg), j+1) / real(j+1);
        quadc[i] /= coef;
    }
    for (i = 0; i<ql; i++) {
        quadx[i] /= deg;
        quadc[i] /= deg;
    }
}

int main() {
    freopen("definite.in", "r", stdin);
    freopen("definite.out", "w", stdout);
    int i;
    for (i = 0; i<5; i++) scanf("%d", inarr+i);

    sign = 1;
    if (inarr[4] < 0) {
        sign = -sign;
        for (i = 0; i<5; i++) inarr[i] = -inarr[i];
    }


    ql = 3;
    quadx[0] = 0.0;
    quadx[1] = 0.5;
    quadx[2] = 1.0;

    quadc[0] = 1/6.0;
    quadc[1] = 4/6.0;
    quadc[2] = 1/6.0;

    for (i = 0; i<MINP; i++) {
        real tl = CUT * (2.0 * real(i)/MINP - 1.0);
        real tr = CUT * (2.0 * real(i+1)/MINP - 1.0);
        lines.insert(Line(tl, tr));
    }

    lSet::iterator it;
    for (i = 0; i<MAXP-MINP; i++) {
        it = lines.begin();
        Line left, right;

        it->Divide(left, right);
        lines.erase(it);

        lines.insert(left);
        lines.insert(right);
    }

    BuildCool(DEG);
    for (it = lines.end(), it--; ; it--) {
        real l = it->l;
        real r = it->r;
        for (i = 0; i<MULT; i++) {
            real tl = (l*real(MULT-i) + r*real(i)) / MULT;
            real tr = (l*real(MULT-i-1) + r*real(i+1)) / MULT;
            ans += Line(tl, tr).integ;
        }
        if (it == lines.begin()) break;
    }
```

```
    printf("%0.15lf\n", sign*ans);

    return 0;
}

//————————————————————————
//Coefficients for Gauss quadrature

real arrx[21] = {0.0031239146898052496, 0.0163865807168468540, 0.0399503329247995820,
                 0.0733183177083413520, 0.1157800182621610600, 0.1664305979012938300,
                 0.2241905820563900900, 0.2878289398962806100, 0.3559893415987994700,
                 0.4272190729195524700, 0.5000000000000000000, 0.5727809270804474800,
                 0.6440106584012005300, 0.7121710601037193300, 0.7758094179436099100,
                 0.8335694020987062200, 0.8842199817378390000, 0.9266816822916587000,
                 0.9600496670752003400, 0.9836134192831531600, 0.9968760853101947800};
real arrc[21] = {0.0080086141288871673, 0.0184768948854262470, 0.0285672127134286060,
                 0.0380500568141896520, 0.0467222117280169310, 0.0543986495835741890,
                 0.0609157080268642670, 0.0661344693166687340, 0.0699436973955365750,
                 0.0722622019949850230, 0.0730405668248452210, 0.0722622019949850230,
                 0.0699436973955365750, 0.0661344693166687340, 0.0609157080268642670,
                 0.0543986495835741890, 0.0467222117280169310, 0.0380500568141896520,
                 0.0285672127134286060, 0.0184768948854262470, 0.0080086141288871673};

---

//The great FFT
typedef std::complex<double> Complex;

void FFT(int size, const Complex *a, Complex *r, int sign) {
    int i, j, k, logn;
    for (logn = 0; size>1; logn++) size >>= 1;
    size = (1<<logn);
    int curr = 0;
    for (i = 0; i<size; i++) {
        r[i] = a[curr];
        for (j = logn-1; j>=0; j--) {
            curr ^= (1<<j);
            if (curr & (1<<j)) break;
        }
    }
    for (i = 0; i<logn; i++) {
        int m = (1<<i);
        int n = (m<<1);
        Complex w1, t, w, f;
        w = 1.0;
        w1 = Complex(cos(sign*2*PI/n), sin(sign*2*PI/n));
        for (j = 0; j<m; j++) {
            for (k = j; k<size; k += n) {
                t = r[k+m]; t *= w;
                f = r[k];
                r[k] += t;
                r[k+m] = f;
                r[k+m] -= t;
            }
            w *= w1;
        }
    }
};

typedef std::vector<int> iVect;
typedef std::vector<Complex> cVect;
```

```
void multPolynoms(cVect &a, cVect &b, cVect &res) {
    int i;
    int alen = a.size();
    int blen = b.size();
    int clen = std::max(alen, blen);
    for (i = 0; (1<<i)<clen; i++);
    clen = (1<<(i+1));
    a.resize(clen);
    b.resize(clen);
    res.resize(clen);
    cVect ar(clen), br(clen);
    FFT(clen, &a[0], &ar[0], 1);
    FFT(clen, &b[0], &br[0], 1);
    for (i = 0; i<clen; i++) ar[i] *= br[i];
    FFT(clen, &ar[0], &res[0], -1);
    for (i = 0; i<clen; i++) res[i] /= clen;
    res.resize(alen + blen - 1);
    a.resize(alen);
    b.resize(blen);
}
//Miller-Rabin primality test
int md;
inline int mult(int a, int b) {
  return (int64(a)*b)%md;
}
inline bool testPrime(int n, int k) {
  md = n;
  int pow = n-1;
  int sq = k % n;
  if (sq == 0) return true;
  int res = 1;
  bool test;
  while (pow > 0) {
    if (pow & 1)
      res = mult(res, sq);
    test = ((sq != 1) && (sq != n-1));
    sq = mult(sq, sq);
    if (test && (sq == 1))
      return false;
    pow >>= 1;
  }
  return (res == 1);
}
//Pollard heuristics

//use any number for c except 0 and 2
inline int func(int x, int n, int c) {
    int t = (int64(x)*x - c) % n;
    if (t < 0) t += n;
    return t;
}
#define TESTS 3
#define ITERS 500 //300/T ~= average iterations for T tests
int GetDivisor(int n) {
    int i, k, d, t;
    int x[TESTS], y[TESTS], c[TESTS];
    i = 1;
    k = 2;
    for (t = 0; t<TESTS; t++) {
        y[t] = x[t] = rand()%n;
        c[t] = rand()%32;
        if (c[t]==0 || c[t]==2) c[t]++; //bad values
```

```
    }
    while (i < ITERS) {
        i++;
        for (t = 0; t<TESTS; t++) {
            x[t] = func(x[t], n, c[t]);
            d = gcd(y[t]-x[t], n);
            if (d!=1 && d!=n) return d;
        }
        if (i==k) {
            for (t = 0; t<TESTS; t++) y[t] = x[t];
            k <<= 1;
        }
    }
    return -1;
}
#define SIZE (1<<20)
bool npr[SIZE];
int primes[SIZE];
int pnum;
#define SIMPLE 100
void PollardToPrimes(int n, int *res, int &k) {
    if (n==1) return;
    if (n <= primes[SIMPLE]*primes[SIMPLE]) {
        res[k++] = n;
        return;
    }
    int t = GetDivisor(n);
    if (t<0) {
        res[k++] = n;
        return;
    }
    PollardToPrimes(t, res, k);
    PollardToPrimes(n/t, res, k);
}
void ToPrimes(int n, int *res, int &k) {
    k = 0;
    int i;
    for (i = 0; i<=SIMPLE && i*i<=n; i++) {
        const int &p = primes[i];
        if (n%p != 0) continue;
        do {
            n /= p;
            res[k++] = p;
        } while (n%p == 0);
    }
    PollardToPrimes(n, res, k);
}
int n, k;
int divs[64];
int main() {
    srand(666);
    int i, j;
    npr[0] = npr[1] = true;
    for (i = 2; i*i<SIZE; i++) if (!npr[i])
        for (j = i*i; j<SIZE; j += i)
            npr[j] = true;
    for (i = 0; i<SIZE; i++) if (!npr[i]) primes[pnum++] = i;
    return 0;
}
```