

# Team notebook

November 6, 2017

## Contents

<b>1</b>	<b>Data Structures</b>	<b>1</b>
1.1	centroid decomposition tree . . . . .	1
1.2	heavy light decomposition . . . . .	1
1.3	hull optimizer . . . . .	2
1.4	hull trick optimization . . . . .	4
1.5	kd tree . . . . .	4
1.6	ordered set . . . . .	6
1.7	treap(explicit) . . . . .	6
1.8	treap(implicit) . . . . .	7
<b>2</b>	<b>Geometry</b>	<b>8</b>
2.1	geometry 2D . . . . .	8
2.2	geometry 3D . . . . .	13
2.3	pick theorem . . . . .	14
<b>3</b>	<b>Graphs</b>	<b>14</b>
3.1	2sat . . . . .	14
3.2	block cut tree . . . . .	15
3.3	tarjan bridges . . . . .	16
3.4	tarjan scc . . . . .	16
<b>4</b>	<b>Math</b>	<b>17</b>
4.1	fft . . . . .	17
4.2	formulas . . . . .	17
4.3	gauss jordan . . . . .	17
4.4	integral . . . . .	18
4.5	latitude longitude converter . . . . .	18
4.6	miller rabin . . . . .	19
4.7	modular arithmetic . . . . .	19
4.8	pollard rho . . . . .	20

4.9	primes . . . . .	21
4.10	simpson . . . . .	22
4.11	utilities . . . . .	22
<b>5</b>	<b>Misc</b>	<b>23</b>
5.1	cc template . . . . .	23
5.2	magic bits . . . . .	23
<b>6</b>	<b>Networks</b>	<b>23</b>
6.1	dilworth theorem . . . . .	23
6.2	dinic . . . . .	24
6.3	hopcroft karp . . . . .	24
6.4	konig theorem . . . . .	25
6.5	max bipartite matching . . . . .	25
6.6	maximum flows with edge demands . . . . .	26
6.7	minimum cost maximum flow . . . . .	26
6.8	minimum cut(bidirectional) . . . . .	27
6.9	minimum cut(directional) . . . . .	27
6.10	push relabel . . . . .	28
<b>7</b>	<b>Strings</b>	<b>29</b>
7.1	aho corasick . . . . .	29
7.2	hashing . . . . .	29
7.3	kmp automaton . . . . .	30
7.4	kmp . . . . .	30
7.5	manacher . . . . .	31
7.6	minimum expression . . . . .	31
7.7	suffix array . . . . .	31
7.8	z algorithm . . . . .	32

# 1 Data Structures

## 1.1 centroid decomposition tree

```
int pr[ MAXN ], sz[ MAXN ];
bool cent[ MAXN ];

void dfs_sz( int u, int p = -1 ) {
    sz[u] = 1;
    for( auto& v : graph[u] ) {
        if( v == p || cent[v] ) continue;
        dfs_sz( v, u );
        sz[u] += sz[v];
    }
}

int find_cent( int u ) {
    for( int t = sz[u]/2, p; ; ) {
        bool ok = true;
        for( auto& v : graph[u] ) {
            if( v == p || cent[v] ) continue;
            if( sz[v] > t ) {
                p = u; u = v;
                ok = false;
                break;
            }
        }
        if( ok ) return u;
    }
    return -1;
}

void decompose( int u, int p = -1 ) {
    dfs_sz( u );
    int c = find_cent( u );
    pr[c] = p;
    cent[c] = true;
    for( auto& v : graph[c] ) {
        if( cent[v] ) continue;
        decompose( v, c );
    }
}
```

## 1.2 heavy light decomposition

```
struct Edge { int v, c, idx; };

int n;
vector< Edge > graph[ MAXN ];
int base_array[ MAXN ], ptr;
int n_chain, chain_ind[ MAXN ], chain_head[ MAXN ], pos_base[ MAXN ];
int depth[ MAXN ], dp[ LOG2 ][ MAXN ], other_end[ MAXN ], subsize[ MAXN ];
int st[ MAXN*4 ];

void init( ) {
    n_chain = ptr = 0;
    for( int i = 0; i < n; ++i ) {
        graph[i].clear();
        chain_head[i] = -1;
        for( int j = 0; j < LOG2; ++j ) {
            dp[j][i] = -1;
        }
    }
}

void make_tree( int node, int s, int e );
void update_tree( int node, int s, int e, int x, int val );
int query_tree( int node, int s, int e, int l, int r );
void create_LCA( );
int LCA( int u, int v );

int query_up( int u, int v ) {
    if( u == v ) return 0;
    int uchain, vchain = chain_ind[v], ans = -1;
    while( true ) {
        uchain = chain_ind[u];
        if( uchain == vchain ) {
            if( u != v ) {
                ans = max( ans, query_tree( 1, 0, n-1, pos_base[v]+1, pos_base[u] ) );
            }
            return ans;
        }
        ans = max( ans, query_tree( 1, 0, n-1, pos_base[ chain_head[uchain] ], pos_base[u] ) );
        u = chain_head[uchain];
        u = dp[0][u];
    }
}
```

```

    return -1;
}

int query( int u, int v ) {
    int lca = LCA( u, v );
    return max( query_up( u, lca ), query_up( v, lca ) );
}

void change( int i, int val ) {
    int u = other_end[i];
    update_tree( 1, 0, n-1, pos_base[u], val );
}

void HLD( int u, int p = -1, int c = -1 ) {
    if( chain_head[n_chain] == -1 ) {
        chain_head[n_chain] = u;
    }
    chain_ind[u] = n_chain;
    pos_base[u] = ptr;
    base_array[ptr++] = c;
    int child = -1, ncost;
    for( int i = 0; i < SIZE(graph[u]); ++i ) {
        Edge& e = graph[u][i];
        if( e.v == p ) continue;
        if( child == -1 || subsize[child] < subsize[e.v] ) {
            child = e.v;
            ncost = e.c;
        }
    }
    if( child != -1 ) {
        HLD( child, u, ncost );
    }
    for( int i = 0; i < SIZE(graph[u]); ++i ) {
        Edge& e = graph[u][i];
        if( e.v == p || e.v == child ) continue;
        n_chain++;
        HLD( e.v, u, e.c );
    }
}

void dfs( int u, int p = -1 ) {
    dp[0][u] = p;
    depth[u] = ( p == -1 ? 0 : depth[p]+1 );
    subsize[u] = 1;
    for( int i = 0; i < SIZE(graph[u]); ++i ) {

```

```

        Edge& e = graph[u][i];
        if( e.v == p ) continue;
        other_end[ e.idx ] = e.v;
        dfs( e.v, u );
        subsize[u] += subsize[e.v];
    }
}

void create_HLD( ) {
    dfs( 0 );
    HLD( 0 );
    make_tree( 1, 0, n-1 );
    create_LCA( );
}

```

### 1.3 hull optimizer

```

/*
 * O( n ) where n = number of lines added
 * Given a set of lines of the form  $y = mx + b$ , find the minimum y-value
 * when any of the given lines are evaluated at the specified x.
 * To optimize for maximum y-value, call the constructor with query_max =
 * true.
 * Reference: https://github.com/alxli
 */

class hull_optimizer {
    struct line {
        ll m, b, val;
        lf xlo;
        bool is_query, query_max;
        line( ll m, ll b, ll val, bool is_query, bool query_max )
            : m(m), b(b), val(val), xlo(-oo),
              is_query(is_query), query_max(query_max) {}
        bool parallel( const line& l )const {
            return m == l.m;
        }
        lf intersect( const line& l )const {
            if( parallel( l ) ) {
                return oo;
            }
            return (lf)( l.b-b )/( m-l.m );
        }
    }
}

```

```

bool operator < ( const line &l )const {
    if( l.is_query ) {
        return query_max ? ( xlo < l.val ) : ( l.val < xlo );
    }
    return m < l.m;
}
};

set< line > hull;
bool query_max;
typedef set<line>::iterator hulliter;

bool has_prev( hulliter it )const {
    return it != hull.begin( );
}
bool has_next( hulliter it )const {
    return ( it != hull.end( ) ) && ( ++it != hull.end( ) );
}
bool irrelevant( hulliter it )const {
    if( !has_prev( it ) || !has_next( it ) ) {
        return false;
    }
    hulliter prev = it, next = it;
    --prev;
    ++next;
    return query_max ? (prev->intersect(*next) <= prev->intersect(*it))
        : (next->intersect(*prev) <= next->intersect(*it));
}
hulliter update_left_border( hulliter it ) {
    if( (query_max && !has_prev(it)) || (!query_max && !has_next(it)) ) {
        return it;
    }
    hulliter it2 = it;
    if val = it->intersect(query_max ? *--it2 : *++it2);
    line l(*it);
    l.xlo = val;
    hull.erase(it++);
    return hull.insert( it, l );
}

public:
hull_optimizer( bool query_max = false ) {
    this->query_max = query_max;
}
void add_line( ll m, ll b ) {

```

```

    line l( m, b, 0, false, query_max );
    hulliter it = hull.lower_bound( l );
    if( it != hull.end( ) && it->parallel( l ) ) {
        if( ( query_max && it->b < b ) || ( !query_max && b < it->b ) ) {
            hull.erase( it++ );
        } else {
            return ;
        }
    }
    it = hull.insert( it, l );
    if( irrelevant( it ) ) {
        hull.erase(it);
        return;
    }
    while( has_prev( it ) && irrelevant( --it ) ) {
        hull.erase( it++ );
    }
    while( has_next( it ) && irrelevant( ++it ) ) {
        hull.erase( it-- );
    }
    it = update_left_border( it );
    if( has_prev( it ) ) {
        update_left_border( --it );
    }
    if( has_next( ++it ) ) {
        update_left_border( ++it );
    }
}
ll get_best( ll x )const {
    line q( 0, 0, x, true, query_max );
    hulliter it = hull.lower_bound( q );
    if( query_max ) {
        --it;
    }
    return it->m*x + it->b;
}
};

```

## 1.4 hull trick optimization

```

struct Line {
    ll m, b;
    Line( ) { }

```

```

Line( ll m, ll b ) : m(m), b(b) { }
ll solve( ll x ) {
    return m*x + b;
}
};

int sz;
Line hull[ MAXN ];
lf inters[ MAXN ];

lf find_intersection( const Line& l1, const Line& l2 ) {
    return lf( l1.b-l2.b )/lf( l2.m-l1.m );
}

void add_line( ll m, ll b ) {
    hull[ sz ] = Line( m, b );
    if( sz == 0 ) {
        inters[ sz ] = oo;
    } else {
        inters[ sz ] = find_intersection( hull[ sz ], hull[ sz-1 ] );
    }
    while( sz >= 2 && inters[ sz ] > inters[ sz-1 ] ) {
        hull[ sz-1 ] = hull[ sz ];
        inters[ sz-1 ] = find_intersection( hull[ sz-2 ], hull[ sz-1 ] );
        sz--;
    }
    sz++;
}

ll get_min( ll x ) {
    int lo = 0, hi = sz-1, mi;
    while( lo <= hi ) {
        mi = ( lo+hi )>>1;
        if( inters[ mi ] > x ) {
            lo = mi+1;
        } else {
            hi = mi-1;
        }
    }
    return hull[ hi ].solve( x );
}

```

## 1.5 kd tree

```

bool cmp_pt_d( const pt &a, const pt &b, int d ) {
    for( int i = 0; i < DIM; ++i ) {
        if( a.v[i] != b.v[ (d+i)%DIM ] ) {
            return a.v[i] < b.v[ (d+i)%DIM ];
        }
    }
    return true;
}

bool cmp_pt( const pt &a, const pt &b ) {
    return cmp_pt_d( a,b,0 );
}

struct Node {
    int dim;
    pt p;
    Node *l, *r;
    Node( int dim, pt &p, Node *l, Node *r ) : dim( dim ), p( p ), l( l ),
        r( r ) {}
};

typedef Node * pnode;

void k_sort( int f, int mi, int t ) {
    for( int i = f; i <= t; ++i ) {
        extra[ i ] = P[ 0 ][ i ];
    }
    for( int i = 1; i < DIM; ++i ) {
        for( int j = f, ii = f, jj = mi+1; j <= t; ++j ) {
            if( extra[ mi ].idx == P[i][j].idx ) continue;
            if( !cmp_pt_d( extra[ mi ], P[i][j], DIM-i ) ) {
                P[ i-1 ][ ii++ ] = P[ i ][ j ];
            } else {
                P[ i-1 ][ jj++ ] = P[ i ][ j ];
            }
        }
    }
    for( int i = f; i <= t; ++i ) {
        P[ DIM-1 ][ i ] = extra[ i ];
    }
}

void create_kd_tree( pnode &root, int f, int t, int d ) {
    if( t == f ) {

```

```

    root = new Node( d, points[P[0][f].idx], NULL, NULL );
    return;
}
int nd = (d+1)%DIM;
if( t-f == 1 ) {
    if( cmp_pt( P[0][f], P[0][t] ) ) {
        create_kd_tree( root, t, t, d );
        create_kd_tree( root->l, f, f, nd );
    } else {
        create_kd_tree( root, f, f, d );
        create_kd_tree( root->l, t, t, nd );
    }
    return;
}
int mi = (t+f+1)/2;
k_sort( f, mi, t );
root = new Node( d, points[ P[0][mi].idx ], NULL, NULL );
create_kd_tree( root->l, f, mi-1, nd );
create_kd_tree( root->r, mi+1, t, nd );
}

void kd_insert( pnode &root, pt &point, int d ) {
    if( root == NULL ) {
        root = new Node( d, point, NULL, NULL );
    } else if( root->p.v[d] <= point.v[d] ) {
        kd_insert( root->r, point, (d+1)%DIM );
    } else {
        kd_insert( root->l, point, (d+1)%DIM );
    }
}

pt min_pt( pt p, pt q, int d ) {
    if( p.v[d] < q.v[d] ) return p;
    if( p.v[d] > q.v[d] ) return q;
    if( samePt(p,q) ) return p;
    return min_pt( p, q, (d+1)%DIM );
}

pt find_min( pnode root, int d ) {
    if( root == NULL ) {
        return pt(oo,oo);
    }
    if( root->dim == d ) {
        if( root->l == NULL ) return root->p;
        return find_min( root->l, d );
    }

```

```

    }
    pt p = find_min( root->l, d );
    pt q = find_min( root->r, d );
    return min_pt( min_pt(p,q,d), root->p, d );
}

void kd_delete( pnode &root, pt point ) {
    if( root == NULL ) return;
    if( samePt(root->p, point) ) {
        if( root->r == NULL && root->l == NULL ) {
            root = NULL;
        } else {
            if( root->r == NULL ) swap( root->l, root->r );
            root->p = find_min( root->r, root->dim );
            kd_delete( root->r, root->p );
        }
        return;
    }
    if( root->p.v[ root->dim ] <= point.v[ root->dim ] ) {
        kd_delete( root->r, point );
    } else {
        kd_delete( root->l, point );
    }
}

void nearest_neighbor( pt &point, pnode &root, pt &r, lf &d ) {
    if( !root ) return;
    lf curd = dist( point, root->p );
    if( curd && d > curd ) {
        d = curd;
        r = root->p;
    }
    lf delta = abs( point.v[ root->dim ] - root->p.v[ root->dim ] );
    delta *= delta;
    if( point.v[ root->dim ] <= root->p.v[ root->dim ] ) {
        nearest_neighbor( point, root->l, r, d );
        if( d >= delta ) {
            nearest_neighbor( point, root->r, r, d );
        }
    } else {
        nearest_neighbor( point, root->r, r, d );
        if( d >= delta ) {
            nearest_neighbor( point, root->l, r, d );
        }
    }
}

```

```
}
```

## 1.6 ordered set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
```

```
using namespace __gnu_pbds;
```

```
typedef
tree<
    T,
    null_type,
    less< T >,
    rb_tree_tag,
    tree_order_statistics_node_update >
ordered_set;

// Get Kth element of type T [ 0, size )
*X.find_by_order( y )
// Count elements smaller than y
X.order_of_key( y )
```

## 1.7 treap(explicit)

```
struct node {
    ll k, p, mn;
    node *l, *r;
    node( ll k ) : k(k), p(rand()), mn(oo), l(nullptr), r(nullptr) { }
};
```

```
typedef node* pnode;
```

```
ll min_node( pnode t ) {
    if( t == nullptr ) {
        return oo;
    }
    return t->mn;
}
```

```
void upd_min( pnode t ) {
```

```
    if( t != nullptr ) {
        t->mn = min( t->k, min( min_node( t->l ), min_node( t->r ) ) );
    }
}
```

```
void merge( pnode &t, pnode left, pnode right ) {
    if( left == nullptr || right == nullptr ) {
        t = ( right == nullptr ) ? left : right;
    }
    else if( left->p > right->p ) {
        merge( left->r, left->r, right );
        t = left;
    }
    else {
        merge( right->l, left, right->l );
        t = right;
    }
    upd_min( t );
}
```

```
void split( pnode t, ll k, pnode &left, pnode &right ) {
    if( t == nullptr ) {
        left = right = nullptr;
    }
    else if( t->k < k ) {
        split( t->r, k, t->r, right );
        left = t;
    }
    else {
        split( t->l, k, left, t->l );
        right = t;
    }
    upd_min( t );
}
```

```
void insert( pnode &t, pnode new_node ) {
    if( t == nullptr ) {
        t = new_node;
    }
    else if( t->p < new_node->p ) {
        split( t, new_node->k, new_node->l, new_node->r );
        t = new_node;
    }
    else if( t->k < new_node->k ) {
        insert( t->r, new_node );
    }
}
```

```

}
else {
    insert( t->l, new_node );
}
upd_min( t );
}

void erase( pnode &t, ll k ) {
    if( t == nullptr ) {
        return;
    }
    if( t->k == k ) {
        merge( t, t->l, t->r );
    }
    else if( t->k < k ) {
        erase( t->r, k );
    }
    else {
        erase( t->l, k );
    }
    upd_min( t );
}

```

## 1.8 treap(implicit)

```

struct node {
    int k, p, cnt, mn;
    bool rvs;
    node *l, *r;
    node( int k ) : k( k ), p( rand() ), cnt( 1 ), rvs( false ), l( NULL ),
        r( NULL ) {}
};

typedef node* pnode;

int cnt( pnode &t ) {
    if( !t ) {
        return 0;
    }
    return t->cnt;
}

void upd_cnt( pnode &t ) {

```

```

    if( t ) {
        t->cnt = 1 + cnt( t->l ) + cnt( t->r );
    }
}

void push( pnode &t ) {
    if( t && t->rvs ) {
        t->rvs = false;
        swap( t->l, t->r );
        if( t->l != NULL ) t->l->rvs ^= true;
        if( t->r != NULL ) t->r->rvs ^= true;
    }
}

void merge( pnode &t, pnode left, pnode right ) {
    push( left ); push( right );
    if( !left || !right ) {
        t = left ? left : right ;
        return;
    }
    if( left->p > right->p ) {
        merge( left->r, left->r, right );
        t = left;
    }
    else {
        merge( right->l, left, right->l );
        t = right;
    }
    upd_cnt( t );
}

void split( pnode t, int k, pnode &left, pnode &right, int add = 0 ) {
    if( !t ) {
        left = right = NULL;
        return;
    }
    push( t );
    int cur_key = add + cnt( t->l );
    if( cur_key < k ) {
        split( t->r, k, t->r, right, add + 1 + cnt( t->l ) );
        left = t;
    }
    else {
        split( t->l, k, left, t->l, add );
        right = t;
    }
}

```



```

    }
    upd_cnt( t );
}

void insert( pnode &t, int idx, int k ) {
    pnode new_node = new node( k );
    if( !t ) {
        t = new_node;
        return;
    }
    pnode left, right;
    split( t, idx, left, right );
    merge( left, left, new_node );
    merge( t, left, right );
    upd_cnt( t );
}

void erase( pnode &t, int k ) {
    if( !t ) {
        return;
    }
    push( t );
    if( t->k == k ) {
        merge( t, t->l, t->r );
    }
    else if( t->k < k ) {
        erase( t->r, k );
    }
    else {
        erase( t->l, k );
    }
    upd_cnt( t );
}

```

## 2 Geometry

### 2.1 geometry 2D

```

const int DIM = 2;
struct pt {
    lf v[DIM];
    pt( ) { }

```

```

    pt( lf x, lf y ) {
        v[0] = x;
        v[1] = y;
    }
};

inline lf x( pt P ) { return P.v[0]; }
inline lf y( pt P ) { return P.v[1]; }

istream& operator >> ( istream& in, pt& p ) {
    for( int i = 0; i < DIM; ++i ) {
        in >> p.v[i];
    }
    return in;
}

ostream& operator << ( ostream& out, const pt& p ) {
    for( int i = 0; i < DIM; ++i ) {
        out << double(p.v[i]) << " ";
    }
    return out;
}

pt operator + ( const pt& A, const pt& B ) { return pt( x(A)+x(B),
    y(A)+y(B) ); }
pt operator - ( const pt& A, const pt& B ) { return pt( x(A)-x(B),
    y(A)-y(B) ); }
pt operator * ( const lf& B, const pt& A ) { return pt( x(A)*B, y(A)*B ); }
pt operator * ( const pt& A, const lf& B ) { return pt( x(A)*B, y(A)*B ); }
pt operator * ( const pt& A, const pt& B ) { return pt(
    x(A)*x(B)-y(A)*y(B), x(A)*y(B)+y(A)*x(B) ); }
pt operator / ( const pt& A, const lf& B ) { return pt( x(A)/B, y(A)/B ); }

inline lf dot( pt A, pt B ) { return x(A)*x(B) + y(A)*y(B); }
inline lf cross( pt A, pt B ) { return x(A)*y(B) - y(A)*x(B); }
inline lf norm( pt A ) { return sqrt( norm(A) ); }
inline lf abs( pt A ) { return sqrt( norm(A) ); }
inline lf arg( pt A ) { return atan2( y(A), x(A) ); }
inline pt exp( pt A ) { return pt( exp( x(A) ) * cos( y(A) ), exp( x(A) )
    * sin( y(A) ) ); }
inline pt rot( pt P, lf ang ) { return P * exp( pt(0,1)*ang ); }
inline pt rotccw( pt P ) { return P * pt(0,1); }

```

```

inline pt rotcw( pt P ) { return P*pt(0,-1); }
inline bool same( lf a, lf b ) { return a+EPS > b && b+EPS > a; }
inline bool samePt( pt A, pt B ) { return same ( x(A), x(B) ) && same (
    y(A), y(B) ); }
inline lf angle( pt A, pt O, pt B ) { return (lf)acos( dot(A-O, B-O) /
    sqrt(norm(O-A) * norm(O-B)) ); }
inline bool parallel( pt A, pt B, pt C, pt D ) { return same ( 0, cross(
    B-A, D-C ) ); }
inline bool ortho( pt A, pt B, pt C, pt D ) { return same ( 0, dot( B-A,
    D-C ) ); }
inline lf dist( pt A, pt B ) { return abs( B - A ); }

pt inversion( lf r, pt A ) {
    return r*A / norm(A);
}

int get_points( pt p, pt q ) {
    return __gcd( abs(x(p)-x(q)), abs(y(p)-y(q)) );
}

// 0 for collinear points ( angle = 0 )
// 1 for angle BAX counter clockwise
// -1 for angle BAX clockwise
int ccw( pt X, pt A, pt B ) {
    lf c = cross( B-A, X-A );
    if( same( c, 0.0 ) ) { return 0; }
    if( c > EPS ) { return 1; }
    return -1;
}

lf distToLine( pt p, pt A, pt B, pt &c ) {
    lf u = dot( p-A , B-A ) / norm( B-A );
    c = A + u*( B-A );
    return dist( p , c );
}

pt refPoint( pt X, pt A, pt B ) {
    pt aux; distToLine( X, A, B, aux );
    return X + lf(2.0)*(aux-X);
}

pt linesIntersection( pt A, pt B, pt C, pt D ) {
    lf x = cross( C, D-C ) - cross( A, D-C );
    x /= cross( B-A, D-C );
    return A + x*(B-A);
}

```

```

inline bool lineContains( pt X, pt A, pt B ) { return fabs(cross( B-A ,
    X-A )) < EPS; }

inline bool segContains( pt X, pt A, pt B ) {
    if ( !same( 0, cross ( A-X, B-X ) ) ) return 0;
    return ( dot( A-X, B-X ) < EPS );
}

inline bool collinearSegsIntersects ( pt A, pt B, pt C, pt D ) {
    return segContains(A,C,D) || segContains(B,C,D)
        || segContains(C,A,B) || segContains(D,A,B);
}

bool segmentsIntersect( pt A, pt B, pt C, pt D ) {
    if( samePt(A,B) )
        return segContains( A, C, D );
    if( samePt(C,D) )
        return segContains( C, A, B );
    if( parallel(A,B,C,D) )
        return collinearSegsIntersects( A,B,C,D );
    pt aux = linesIntersection(A,B,C,D);
    return segContains(aux,A,B) && segContains(aux,C,D);
}

lf distToSegment( pt p, pt A, pt B, pt &c ) {
    lf u = dot( p-A , B-A ) / norm( B-A );
    if( u < -EPS ) { c = A; return dist( p , A ); }
    if( (u-1.0) > EPS ) { c = B; return dist( p , B ); }
    return distToLine(p,A,B,c);
}

inline bool insideCircle( pt p, pt c, lf r ) { return norm(c-p) <
    (r*r)+EPS; }

//From two Points and Radius, get center of the circle
//There are two possible centers, to get the other, reverse p1 p2
bool circle2Pt (pt p1, pt p2, lf r, pt& c) {
    lf d2 = x(p1-p2) * x(p1-p2) + y(p1-p2) * y(p1-p2);
    lf det = r*r / d2 - 0.25;
    if( det < -EPS ) return false;
    lf h = sqrt(det);
    c.v[0] = x(p1+p2)*0.5 + y(p1-p2)*h ;
    c.v[1] = y(p1+p2)*0.5 + x(p2-p1)*h ;
    return true;
}

```

```

}

pt circle3Pt(pt a, pt b, pt c) {
    b = (a+b)/lf(2.0); c = (a+c)/lf(2.0);
    return linesIntersection(b, b+rotcw(a-b), c, c+rotcw(a-c));
}

bool circleLineIntersection( pt c, lf r, pt A, pt B, pt &p1, pt &p2 ) {
    pt t;
    lf u = distToLine( c, A, B, t );
    if( u > r+EPS ) {
        return false;
    }
    pt v = (B-A)/abs(B-A);
    lf d = sqrt(r*r - u*u);
    p1 = t + d*v;
    p2 = t - d*v;
    return true;
}

// -1 for same circles
// 0 for no intersection
// 1 for tangent
// 2 for 2 points of intersection
int intersectionCircles( pt c1, lf r1, pt c2, lf r2, pt &p1, pt &p2 ) {
    if( samePt( c1, c2 ) && same(r1,r2) ) return -1;
    lf sr = (r1 + r2) * (r1 + r2);
    lf dr = (r1 - r2) * (r1 - r2);
    lf d = norm( c2-c1 );
    if( d+EPS < dr || d > sr+EPS ) return 0;
    if ( same(d,sr) || same(d,dr) ) {
        p1 = p2 = c1 + (c2-c1)/sqrt(d) * r1;
        return 1;
    }
    pt tmp;
    tmp.v[0] = (r1*r1 - r2*r2 + d) / (2.0*sqrt(d));
    tmp.v[1] = sqrt( r1*r1 - x(tmp)*x(tmp) );
    lf ang = arg( c2 - c1 );
    p1 = rot( tmp, ang ) + c1;
    p2 = refPoint( p1, c1, c2 );
    return 2;
}

// P[0] must be equal to P[n]

```

```

double perimeter(const vector<pt> &P) {
    double result = 0.0;
    for(int i = 0; i < (int)P.size()-1; i++) result += dist( P[i],P[i+1] );
    return result;
}

// P[0] must be equal to P[n]
// Area is positive if the polygon is ccw
double signedArea(const vector<pt> &P) {
    double result = 0.0;
    for(int i = 0; i < (int)P.size()-1; i++) result += cross( P[i],P[i+1] );
    return result / 2.0;
}

double area(const vector<pt> &P) { return fabs(signedArea(P)); }

// P[0] must be equal to P[n]
bool isConvex( const vector<pt> &P) {
    int sz = (int) P.size(); if(sz <= 3) return false;
    bool isL = ccw(P[0], P[1], P[2]) >= 0;
    for (int i = 1; i < sz-1; i++) {
        if( ( ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) >= 0 ) != isL )
            return false;
    }
    return true;
}

// P[0] must be equal to P[n]
pt computeCentroid(const vector<pt> &p) {
    pt c(0,0);
    lf scale = 6.0 * signedArea(p);
    for (int i = 0, j = 1; i < p.size()-1; i++, j++)
        c = c + (p[i]+p[j])*(x(p[i]) * y(p[j]) - x(p[j]) * y(p[i]));
    return c / scale;
}

// P[0] must be equal to P[n]
bool isSimple(const vector<pt> &p) {
    for (int i = 0, j, l; i < p.size()-1; i++) {
        for (int k = i+1; k < p.size()-1; k++) {
            j = (i+1); l = (k+1);
            if (i == l || j == k) continue;
            if (segmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
}

```

```

    }
    return true;
}

// P[0] must be equal to P[n]
// Return 1 for interior, 0 for boundary and -1 for exterior
// O( N )
int inPolygon(pt X, const vector<pt> &P) {
    const int n = P.size(); int cnt = 0;
    for (int i = 0; i < n-1; i++) {
        if( segContains(X, P[i], P[i+1]) ) return 0;
        if( y(P[i]) <= y(X) ) {
            if( y(P[i+1]) > y(X) )
                if( !(ccw( X, P[i], P[i+1]) >= 0) ) cnt++;
        }
        else if ( y(P[i+1]) <= y(X) ) {
            if( ccw( X, P[i], P[i+1]) >= 0 ) cnt--;
        }
    }
    if(cnt == 0) return -1;
    else return 1;
}

// P[ 0 ] must be the left most (down) point
// 0 for collinear, 1 for inside, -1 for outside
// O( Log N )
int inConvexPolygon( pt X, lf mnx, lf mxx, vector<pt> &P ) {
    if( x(X) < mnx || x(X) > mxx )
        return -1;
    int lo = 1, hi = int( P.size() )-1, mi;
    while( lo <= hi ) {
        mi = (lo+hi)/2;
        if( cross( P[mi]-P[0], X-P[0] ) < -EPS ) {
            lo = mi+1;
        }
        else {
            hi = mi-1;
        }
    }
    lo = hi;
    if( hi == -1 ) return -1;
    lf c = cross( X-P[lo], X-P[lo+1] );
    if( same( c, 0.0 ) )
        return ( segContains( X, P[lo], P[lo+1] ) ? 0 : -1 );
    if( c > EPS )

```

```

        return -1;
    return 1;
}

// O( N )
lf diameterOfConvexPolygon( const vector<pt> &P, pt &A, pt &B ) {
    lf ans = -oo, d;
    int lo = 0, hi = 0;
    int sz = int(P.size());
    for( int i = 0, j = 0; i < sz; ++i ) {
        while( dist( P[i], P[j] )+EPS < dist( P[i], P[ (j+1)%sz ] ) ) {
            j = (j+1)%sz;
        }
        d = dist( P[i], P[j] );
        if( ans+EPS < d ) {
            ans = d;
            lo = i; hi = j;
        }
    }
    A = P[lo]; B = P[hi];
    return ans;
}

//Returns the Polygon to the left of AB (counter clockwise)
// O( N )
vector<pt> cutPolygon( pt A, pt B, const vector<pt> &P ) {
    vector<pt> Q;
    for (int i = 0; i < (int)P.size(); i++) {
        double left1 = cross( B-A , P[i]-A ), left2 = 0;
        if( i != (int)P.size()-1 ) left2 = cross( B-A , P[i+1]-A );
        if(left1 > -EPS) Q.push_back(P[i]);
        if( left1 * left2 < -EPS ) Q.push_back( linesIntersection(P[i],
            P[i+1], A, B ) );
    }
    if (!Q.empty() && !samePt(Q.back(), Q.front()) ) Q.push_back(Q.front());
    return Q;
}

// Returns Polygon in clockwise and with leftmost (down) point at P[0]
// O( N )
vector<pt> reorganize( vector<pt> &P ) {
    int n = int(P.size());
    vector<pt> R( n );
    if( P.size() == 1 ) {
        R[ 0 ] = P[ 0 ];
    }

```

```

    return R;
}
//Check if is counterclockwise
if ( signedArea( P ) > EPS ) { reverse( P.begin(), P.end() ); }
int s = 0;
for( int i = 1; i < n; ++i ) {
    if( x(P[s]) > x(P[i]) || ( x(P[s]) == x(P[i]) && y(P[s]) > y(P[i]) ) ) {
        s = i;
    }
}
R[ 0 ] = P[ s ];
for( int i = (s+1)%n, j = 1; i != s; i = (i+1)%n, ++j ) {
    if( samePt( P[i], P[(i-1+n)%n] ) ) {
        j--;
        continue;
    }
    R[ j ] = P[ i ];
}
R[ n-1 ] = R[ 0 ];
return R;
}

// P and Q must P[0] = P[n]
// Be careful with polygons of just one point
// O( N + M )
vector<pt> convexPolygonSum( vector<pt> &P, vector<pt> &Q ) {
    P = reorganize( P );
    Q = reorganize( Q );
    int n = int( P.size() ), m = int( Q.size() );
    vector<pt> R( n+m-1 );
    R[ 0 ] = (P[ 0 ] + Q[ 0 ]);
    int i = 1, j = 1, k = 1;
    for( ; i < n && j < m; ++k ) {
        if( cross( P[i]-P[i-1], Q[j]-Q[j-1] ) < -EPS ) {
            R[ k ] = R[ k-1 ] + ( P[ i ]-P[ i-1 ] );
            ++i;
        }
        else {
            R[ k ] = R[ k-1 ] + ( Q[ j ]-Q[ j-1 ] );
            ++j;
        }
    }
    while( i < n ) {
        R[ k ] = R[ k-1 ] + ( P[ i ]-P[ i-1 ] );

```

```

        ++i;
        ++k;
    }
    while( j < m ) {
        R[ k ] = R[ k-1 ] + ( Q[ j ]-Q[ j-1 ] );
        ++j;
        ++k;
    }
    vector<pt> T;
    T.PB( R[ 0 ] );
    for( int i = 1; i+1 < int(R.size()); ++i ) {
        if( same( cross( R[i]-R[i-1], R[i+1]-R[i-1] ), 0.0 ) )
            continue;
        T.PB( R[i] );
    }
    T.PB( T[ 0 ] );

    return T;
}

// Monotone Chain O( N Log N )
bool cmpPt( pt A, pt B ) {
    if( !same( x(A), x(B) ) ) return x(A) < x(B);
    return y(A) < y(B);
}

int turn(pt A, pt B, pt C) {
    lf r = cross(B-A, C-A);
    if( same( r, 0.0 ) ) return 0;
    if( r > EPS ) return 1;
    return -1;
}

// Return CH in ccw order starting at leftmost - downmost x
// Doesn't return P[ n ] = P[ 0 ]
vector<pt> CH( vector<pt> &P ) {
    if ( P.size() == 1 ) return P;
    const int n = P.size();
    sort ( P.begin(), P.end(), cmpPt );
    vector<pt> up;
    up.push_back(P[0]); up.push_back(P[1]);
    vector<pt> dn;
    dn.push_back(P[0]); dn.push_back(P[1]);
    for ( int i = 2; i < n; ++i ) {
        // If collinear points are needed is > and <, otherwise >= and <=

```

```

while ( up.size() > 1 && turn(up[up.size()-2],up.back(),P[i]) >= 0 )
    up.pop_back();
while ( dn.size() > 1 && turn(dn[dn.size()-2],dn.back(),P[i]) <= 0 )
    dn.pop_back();
up.push_back(P[i]);
dn.push_back(P[i]);
}
for (int i = (int) up.size() - 2; i >= 1; i--) dn.push_back(up[i]);
return dn;
}

```

## 2.2 geometry 3D

```

struct pt {
    lf x, y, z;
    pt() {}
    pt( lf x, lf y, lf z ) : x( x ), y ( y ), z ( z ) {}
};

const lf EPS = 1e-9;
const lf PI = acos( -1.0 );
const pt o = pt( 0.0, 0.0, 0.0 );

inline lf x( pt P ) { return P.x; }
inline lf y( pt P ) { return P.y; }
inline lf z( pt P ) { return P.z; }

istream& operator >> ( istream& in, pt& p ) {
    lf x,y,z; in >> x >> y >> z;
    p = pt(x,y,z); return in;
}

ostream& operator << ( ostream& out, const pt& p ) {
    out << "(" << p.x << ", " << p.y << ", " << p.z << ")";
    return out;
}

pt operator + ( const pt& A, const pt& B ) { return { x(A)+x(B),
    y(A)+y(B), z(A)+z(B) }; }
pt operator - ( const pt& A, const pt& B ) { return { x(A)-x(B),
    y(A)-y(B), z(A)-z(B) }; }
pt operator * ( const pt& A, const lf& B ) { return { x(A)*B, y(A)*B,
    z(A)*B }; }

```

```

pt operator * ( const lf& B, const pt& A ) { return { x(A)*B, y(A)*B,
    z(A)*B }; }
pt operator / ( const pt& A, const lf& B ) { return { x(A)/B, y(A)/B,
    z(A)/B }; }

inline pt cross( pt A, pt B ) { return pt( y(A)*z(B)-z(A)*y(B),
    z(A)*x(B)-x(A)*z(B), x(A)*y(B)-y(A)*x(B) ); }
inline lf dot( pt A, pt B ) { return x(A)*x(B) + y(A)*y(B) + z(A)*z(B); }
inline lf norm( pt A ) { return x(A)*x(A) + y(A)*y(A) + z(A)*z(A); }
inline lf abs( pt A ) { return sqrt( norm(A) ); }
inline bool same ( lf a, lf b ) { return a+EPS > b && b+EPS > a; }
inline bool samePt ( pt A, pt B ) { return same ( x(A), x(B) ) && same (
    y(A), y(B) ) && same ( z(A), z(B) ); }
inline bool zero( lf d ) { return d >= -EPS && d <= EPS; }

bool is_plane( pt A, pt B, pt C ) {
    return !samePt( cross( B-A, C-A ), o );
}

// 1 for intersect, 0 for inside, -1 for parallel
int linePlane( pt S, pt T, pt A, pt B, pt C, pt& r ) {
    pt n = cross( B-A, C-A );
    pt u = T-S;
    lf d = dot( n, u );
    if( !zero( d ) ) {
        d = dot( n, A-S ) / d;
        r = S + u*d;
        return 1;
    }
    d = dot( n, A-S );
    if( zero( d ) ) return 0;
    return -1;
}

bool lineLineIntersection( pt A, pt B, pt C, pt D, pt& S ) {
    pt e = B-A, f = D-C, g = C-A;
    pt fg = cross( f, g ), fe = cross( f, e );
    lf h = abs( fg ), k = abs( fe );
    if( zero( h ) || zero( k ) ) return false;
    if( samePt( cross( fg, fe ), o ) )
        S = A + e*h/k;
    else
        S = A - e*h/k;
    return true;
}

```

```

bool planesIntersection( pt A, pt B, pt C, pt D, pt E, pt F, pt& S, pt& T
) {
    pt n1 = cross( B-A, C-A );
    pt n2 = cross( D-E, F-E );
    pt u = cross( n1, n2 );
    if( samePt( u, o ) ) return false;
    lineLineIntersection( A, B, D, E, S );
    T = S + u;
}

```

---

## 2.3 pick theorem

$$A = I + \frac{B}{2} - 1$$

- A: Area
- I: Points inside the polygon
- B: Points in the boundary of the polygon

# 3 Graphs

## 3.1 2sat

```

/*
 * Equivalences
 * (s1^a2)v(a1^s2) = (s1vs2)^(a1va2)^(s1va1)^(s2va2)
 */
struct SAT {
    int n;
    vector< vector< vi > > graph;
    vi tag;
    vb seen, value;
    stack< int > st;
    SAT( int n ) : n( n ), graph( 2, vector< vi >( 2*n ) ), tag( 2*n ),
        seen( 2*n ), value( 2*n ) { }
    int neg( int x ) {
        return 2*n-x-1;
    }
    ///We give u v v and it makes u -> v and v -> u
    void make_implication( int u, int v ) {

```

```

        implication( neg(u), v );
        implication( neg(v), u );
    }
    void make_true( int u ) {
        add_edge( neg(u), u );
    }
    void make_false( int u ) {
        make_true( neg(u) );
    }
    void eq( int u, int v ) {
        implication( u, v );
        implication( v, u );
    }
    void diff( int u, int v ) {
        eq( u, neg(v) );
    }
    void implication( int u, int v ) {
        add_edge( u, v );
        add_edge( neg(v), neg(u) );
    }
    void add_edge( int u, int v ) {
        graph[ 0 ][ u ].push_back( v );
        graph[ 1 ][ v ].push_back( u );
    }
    void dfs( int id, int u, int t = 0 ) {
        seen[ u ] = true;
        for( auto& v : graph[ id ][ u ] )
            if( !seen[ v ] )
                dfs( id, v, t );
        if( id == 0 )
            st.push( u );
        else
            tag[ u ] = t;
    }
    void kosaraju( ) {
        for( int u = 0; u < n; u++ ) {
            if( !seen[ u ] )
                dfs( 0, u );
            if( !seen[ neg(u) ] )
                dfs( 0, neg(u) );
        }
        fill( seen.begin( ), seen.end( ), false );
        int t = 0;
        while( !st.empty( ) ) {
            int u = st.top( ); st.pop( );

```

```

        if( !seen[ u ] )
            dfs( 1, u, t++ );
    }
}
bool satisfiable( ) {
    kosaraju();
    for( int i = 0; i < n; i++ ) {
        if( tag[ i ] == tag[ neg(i) ] ) return false;
        value[ i ] = tag[ i ] > tag[ neg(i) ];
    }
    return true;
}
};

```

---

## 3.2 block cut tree

```

namespace BlockCutTree {

    int t, rootCh, typeCnt;
    int low[ MAX ], dfn[ MAX ], type[ MAX ];
    vi graph[ MAX ];
    bool cut[ MAX ];
    map< pii, int > bridges;
    stack< int > s;

    void init( ) {
        t = rootCh = typeCnt = 0;
        bridges.clear( );
        for( int i = 0; i < MAX; i++ ) {
            dfn[ i ] = 0;
            cut[ i ] = false;
            graph[ i ].clear( );
        }
    }

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

    void tarjan( int u, int fu ) {
        low[ u ] = dfn[ u ] = ++t;
        for( auto& v : graph[ u ] ) {
            if( v == fu ) continue;

```

```

            if( !dfn[ v ] ){
                if( u == 1 ) rootCh++;
                s.push( v );
                tarjan( v, u );
                low[ u ] = min( low[ u ], low[ v ] );
                if( low[ v ] >= dfn[ u ] ) {
                    int w;
                    typeCnt++;
                    do {
                        w = s.top( ); s.pop( );
                        if( cut[ w ] )
                            LowestCommonAncestor::add_edge( typeCnt, type[ w ] );
                        else type[ w ] = typeCnt;
                    } while( w != v );
                    if( low[ v ] > dfn[ u ] )
                        bridges[ make_pair( min( u, v ), max( u, v ) ) ] = typeCnt;
                    if( !cut[ u ] ) {
                        cut[ u ] = true;
                        type[ u ] = ++typeCnt;
                        LowestCommonAncestor::add_edge( typeCnt, typeCnt-1 );
                    }
                    else
                        LowestCommonAncestor::add_edge( type[ u ], typeCnt );
                }
            }
            else low[ u ] = min( low[ u ], dfn[ v ] );
        }
    }

    void create_block_cut_tree( ) {
        LowestCommonAncestor::init( );
        tarjan( 1, 1 );
        if( rootCh == 1 ){
            cut[ 1 ] = false;
            type[ 1 ] = --typeCnt;
        }
        LowestCommonAncestor::dfs( type[ 1 ], type[ 1 ] );
        LowestCommonAncestor::build_sparse_table( );
    }
}

```

---

## 3.3 tarjan bridges



---

```

void dfs( int u, int p = -1 ) {
    dfn[ u ] = low[ u ] = ++t;
    int children = 0;
    for( int i = 0; i < SIZE( graph[u] ); ++i ) {
        int v = graph[ u ][ i ];
        if( !dfn[ v ] == -1 ) {
            children++;
            dfs( v, u );
            low[ u ] = min( low[u], low[v] );
            ///Bridges
            if( low[v] > dfn[u] ) {
                cout << u << " " << v << endl;
            }
            ///Articulation points
            if( p == -1 && children > 1 ) {
                ap[ u ] = true;
            }
            if( p != -1 && low[v] >= dfn[u] ) {
                ap[ u ] = true;
            }
        }
        else if( v != p ) {
            low[ u ] = min( low[u], dfn[v] );
        }
    }
}

```

---

### 3.4 tarjan scc

---

```

void dfs( int u ) {
    dfn[ u ] = low[ u ] = ++t;
    st.push( u );
    in_stack[ u ] = true;
    for( int i = 0; i < SIZE( graph[u] ); ++i ) {
        int v = graph[ u ][ i ];
        if( dfn[ v ] == -1 ) {
            dfs( v );
            low[ u ] = min( low[ u ], low[ v ] );
        }
        else if( in_stack[v] == true ) {
            low[ u ] = min( low[u], dfn[v] );
        }
    }
}

```

---

```

}
if( low[ u ] == dfn[ u ] ) {
    int w;
    while( st.top() != u ) {
        w = st.top();
        cout << w << " ";
        in_stack[ w ] = false;
        st.pop();
    }
    w = st.top();
    cout << w << "\n";
    in_stack[ w ] = false;
    st.pop();
}
}

```

---

## 4 Math

### 4.1 fft

---

```

const lf PI = acos( -1.0 );

struct cp { lf r, i; };

cp operator + ( const cp& a, const cp& b ) { return { a.r+b.r, a.i+b.i }; }
cp operator - ( const cp& a, const cp& b ) { return { a.r-b.r, a.i-b.i }; }
cp operator * ( const cp& a, const cp& b ) { return { a.r*b.r-a.i*b.i,
    a.r*b.i+a.i*b.r }; }
cp operator * ( const cp& a, lf x ) { return { a.r*x, a.i*x }; }
cp operator * ( lf x, const cp& a ) { return { a.r*x, a.i*x }; }
cp operator / ( const cp& a, lf x ) { return { a.r/x, a.i/x }; }

ostream& operator << ( ostream& out, const cp& c ) {
    out << c.r;
    return out;
}

void rev( cp* a, int n ) {
    int i, j, k;
    for( i = 1, j = n>>1; i < n-1; ++i ) {

```

---

```

    if( i < j ) swap( a[ i ], a[ j ] );
    for( k = n>>1; j >= k; j -= k, k >>= 1 );
    j += k;
}
}

void dft( cp* a, int n, int flag = 1 ) {
    rev( a, n );
    for( int m = 2; m <= n; m <<= 1 ) {
        cp wm = { cos( flag*2.0*PI/m ), sin( flag*2.0*PI/m ) };
        for( int k = 0; k < n; k += m ) {
            cp w = { 1.0, 0.0 };
            for( int j = k; j < k+(m>>1); ++j, w = w*wm ) {
                cp u = a[ j ], v = a[ j+(m>>1) ]*w;
                a[ j ] = u+v;
                a[ j+(m>>1) ] = u-v;
            }
        }
    }
}

void mul( int na, cp* a, int nb, cp* b ) {
    int n = 1;
    while( n <= na+nb+1 ) n <<= 1;
    dft( a, n ); dft( b, n );
    for( int i = 0; i < n; ++i ) {
        a[ i ] = a[ i ]*b[ i ];
    }
    dft( a, n, -1 );
    for( int i = 0; i < n; ++i ) {
        a[ i ].r = round( a[ i ].r/lf(n) );
    }
}

```

## 4.2 formulas

- $a^T(\text{mod } m) = a^{T \bmod n}(\text{mod } m)$   
iff  $a$  and  $m$  are coprime, then  $n = \text{phi}(m)$
- Remember that the highest power of a prime  $p$  dividing  $n!$  is given by the procedure:
  1. Greatest integer less than or equal to  $\frac{n}{p}$
  2. Greatest integer less than or equal to  $\frac{n}{p^2}$

3. Greatest integer less than or equal to  $\frac{n}{p^3}$
4. Repeat until the greatest integer less than or equal to  $\frac{n}{p^k}$  is 0
5. Add all of your numbers up.

## 4.3 gauss jordan

```

const double EPS = 1e-10;

double Gauss_Jordan( vvd& a, vvd& b ) {
    const int n = int( a.size( ) );
    const int m = int( a[ 0 ].size( ) );
    vi irow( n ), icol( n ), ipiv( n );
    double det = 1;
    for( int i = 0; i < n; i++ ) {
        int pj = -1, pk = -1;
        for( int j = 0; j < n; j++ ) {
            if( !ipiv[ j ] ) {
                for( int k = 0; k < n; k++ ) {
                    if( !ipiv[ k ] ) {
                        if( pj == -1 || abs( a[ j ][ k ] ) > abs( a[ pj ][ pk ] ) ) {
                            pj = j;
                            pk = k;
                        }
                    }
                }
            }
        }
        if( abs( a[ pj ][ pk ] ) < EPS ) {
            cerr << "Matrix is singular." << endl;
            exit( 0 );
        }
        ipiv[ pk ]++;
        swap( a[ pj ], a[ pk ] );
        swap( b[ pj ], b[ pk ] );
        if( pj != pk ) {
            det *= -1;
        }
        irow[ i ] = pj;
        icol[ i ] = pk;
        double c = 1.0/a[ pk ][ pk ];
        det *= a[ pk ][ pk ];
        a[ pk ][ pk ] = 1.0;
        for( int p = 0; p < n; p++ ) {

```

---

```

    a[ pk ][ p ] *= c;
}
for( int p = 0; p < m; p++ ) {
    b[ pk ][ p ] *= c;
}
for( int p = 0; p < n; p++ ) {
    if( p != pk ) {
        c = a[ p ][ pk ];
        a[ p ][ pk ] = 0;
        for( int q = 0; q < n; q++ ) {
            a[ p ][ q ] -= a[ pk ][ q ]*c;
        }
        for( int q = 0; q < m; q++ ) {
            b[ p ][ q ] -= b[ pk ][ q ]*c;
        }
    }
}
for( int p = n-1; p >= 0; p-- ) {
    if( irow[ p ] != icol[ p ] ) {
        for( int k = 0; k < n; k++ ) {
            swap( a[ k ][ irow[ p ] ], a[ k ][ icol[ p ] ] );
        }
    }
}
return det;
}

```

---

## 4.4 integral

- Simpsons rule:  $\int_a^b f(x)dx \approx \frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)]$
- Arc length:  $s = \int_a^b \sqrt{1 + [f'(x)]^2}dx$
- Area of a surface of revolution:  $A = 2\pi \int_a^b f(x)\sqrt{1 + [f'(x)]^2}dx$
- Volume of a solid of revolution:  $V = \pi \int_a^b f(x)^2 dx$
- Note: In case of multiple functions such as  $g(x)$   $h(x)$  for a solid of revolution then  $f(x) = g(x) - h(x)$

## 4.5 latitude longitude converter

---

```

struct Lat_Lon {
    double r, lat, lon;
};

struct Rect {
    double x, y, z;
};

Lat_Lon convert( Rect& p ) {
    Lat_Lon q;
    q.r = sqrt( p.x*p.x + p.y*p.y + p.z*p.z );
    q.lat = 180.0/PI*asin( p.z/q.r );
    q.lon = 180.0/PI*acos( p.x/sqrt( p.x*p.x + p.y*p.y ) );
    return q;
}

Rect convert( Lat_Lon& q ) {
    Rect p;
    p.x = q.r*cos( q.lon*PI/180.0 )*cos( q.lat*PI/180.0 );
    p.y = q.r*sin( q.lon*PI/180.0 )*cos( q.lat*PI/180.0 );
    p.z = q.r*sin( q.lat*PI/180.0 );
    return p;
}

```

---

## 4.6 miller rabin

---

```

bool check( ll a, ll n ) {
    ll u = n-1;
    int t = 0;
    while( u%2LL == 0 ) {
        t++;
        u /= 2LL;
    }
    ll nxt = mod_pow( a, u, n );
    if( nxt == 1 )
        return false;
    ll lst;
    for( int i = 0; i < t; i++ ) {
        lst = nxt;
        nxt = mod_mul( lst, lst, n );
        if( nxt == 1 ) {
            return ( lst != n-1 );
        }
    }
}

```

---

```

    }
}
return ( nxt != 1 );
}

bool miller_rabin( ll n, int it = 20 ) {
    if( n <= 1 ) {
        return false;
    }
    if( n == 2 ) {
        return true;
    }
    if( n%2LL == 0 ) {
        return false;
    }
    for( int i = 0; i < it; i++ ) {
        ll a = rand()%( n-1 ) + 1;
        if( check( a, n ) ) {
            return false;
        }
    }
    return true;
}

```

## 4.7 modular arithmetic

```

/*
 * Modular Arithmetic
 */
int mod_of( int n, int mod ) {
    return ( ( n%mod )+mod )%mod;
}

/// returns d = gcd( a, b ); finds x, y such that d = a*x + b*y
int extended_euclid( int a, int b, int &x, int &y ) {
    int xx = 0; y = 0;
    int yy = 1; x = 1;
    while( b ) {
        int q = a/b;
        int t = b; b = a%b; a = t;
        t = xx; xx = x-q*xx; x = t;
        t = yy; yy = y-q*yy; y = t;
    }
}

```

```

    return a;
}

int mod_inverse( int a, int n ) {
    int x, y;
    int d = extended_euclid( a, n, x, y );
    if( d > 1 ) {
        return -1;
    }
    return x%n;
}

/// computes x and y such that ax + by = c
bool linear_diophantine( int a, int b, int c, int &x, int &y ) {
    int d = __gcd( a, b );
    if( c%d ) {
        return false;
    }
    x = c/d*mod_inverse( a/d, b/d );
    y = ( c-a*x )/b;
    return true;
}

/// finds all solutions to a*x = b ( mod n )
vi modular_linear_equation_solver( int a, int b, int n ) {
    a = mod_of( a, n );
    b = mod_of( b, n );
    vi ret;
    int x, y;
    int d = extended_euclid( a, n, x, y );
    if( b%d == 0 ) {
        x = mod_of( x*( b/d ), n );
        for( int i = 0; i < d; i++ ) {
            ret.pb( mod_of( x+i*( n/d ), n ) );
        }
    }
    return ret;
}

/// Chinese remainder theorem (special case): find z such that
/// z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).
/// Return (z,M). On failure, M = -1.
pii chinese_remainder_theorem( int x, int a, int y, int b ) {
    int s, t;
    int d = extended_euclid( x, y, s, t );
}

```

```

    if( a%d != b%d ) {
        return MP( 0, -1 );
    }
    return MP( mod_of( s*b*x+t*a*y, x*y )/d, x*y/d );
}

/// Chinese remainder theorem: find z such that
/// z % x[i] = a[i] for all i. Note that the solution is
/// unique modulo M = lcm_i (x[i]). Return (z,M). On
/// failure, M = -1. Note that we do not require the a[i]'s
/// to be relatively prime.
pii chinese_remainder_theorem( const vi& x, const vi& a ) {
    pii ret = MP( a[ 0 ], x[ 0 ] );
    for( int i = 1; i < int( x.size( ) ); i++ ) {
        ret = chinese_remainder_theorem( ret.SE, ret.FI, x[ i ], a[ i ] );
        if( ret.SE == -1 ) {
            break;
        }
    }
    return ret;
}

```

---

## 4.8 pollard rho

```

ll pollard_rho( ll n ) {
    ll x, y, i = 1, k = 2, d;
    x = y = rand( )%n;
    while( true ) {
        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;
        }
    }
}

```

---

```

        k *= 2LL;
    }
}
return 1;
}

vll factorize( ll n ) {
    vll ans;
    if( n == 1 ) {
        return ans;
    }
    if( miller_rabin( n ) ) {
        ans.PB( n );
    }
    else {
        ll d = 1;
        while( d == 1 ) {
            d = pollard_rho( n );
        }
        vll dd = factorize( d );
        ans = factorize( n/d );
        for( int i = 0; i < dd.size( ); i++ ) {
            ans.PB( dd[ i ] );
        }
    }
    return ans;
}

```

---

## 4.9 primes

```

vi count_divisors_sieve( ) {
    const int mx = int( 1e7 )+1;
    bitset< mx > is_prime; is_prime.set( );
    vi cnt( mx, 1 );
    is_prime[ 0 ] = is_prime[ 1 ] = 0;
    for( int i = 2; i < mx; i++ ) {
        if( is_prime[ i ] ) {
            cnt[ i ]++;
            for( int j = i+i; j < mx; j += i ) {
                int n = j, c = 1;
                while( n%i == 0 ) {
                    n /= i;
                    c++;
                }
            }
        }
    }
}

```

```

    }
    cnt[ j ] = cnt[ j ]*c;
    is_prime[ j ] = 0;
}
}
return cnt;
}

vi euler_phi_sieve( ) {
    const int mx = int( 1e7 )+1;
    bitset< mx > is_prime; is_prime.set( );
    vi phi( mx );
    for( int i = 1; i < mx; i++ ) {
        phi[ i ] = i;
    }
    is_prime[ 0 ] = is_prime[ 1 ] = 0;
    for( int i = 2; i < mx; i++ ) {
        if( is_prime[ i ] ) {
            for( int j = i; j < mx; j += i ) {
                phi[ j ] = phi[ j ]-( phi[ j ]/i );
                is_prime[ j ] = 0;
            }
        }
    }
    return phi;
}

ll count_divisors( vll& prime, ll n ) {
    int total_primes = int( prime.size( ) );
    ll r = 1;
    for( int i = 0; prime[ i ]*prime[ i ] <= n && i < total_primes; i++ ) {
        ll p = 1;
        while( n%prime[ i ] == 0 ) {
            n /= prime[ i ];
            p++;
        }
        r = r*p;
    }
    if( n != 1 ) {
        r = r*2LL;
    }
    return r;
}

```

```

ll highest_exponent( ll n, ll p ) {
    ll r = 0, t = p;
    while( t <= n ) {
        r = r+( n/t );
        t = t*p;
    }
    return r;
}

ll count_divisors_factorial( vll& prime, ll n ) {
    int total_primes = int( prime.size( ) );
    ll r = 0;
    for( int i = 0; prime[ i ] <= n && i < total_primes; i++ ) {
        r = r+( highest_exponent( n, prime[ i ] )+1 );
    }
    return r;
}

ll sum_divisors( vll& prime, ll n ) {
    int total_primes = int( prime.size( ) );
    ll r = 1;
    for( int i = 0; prime[ i ]*prime[ i ] <= n && i < total_primes; i++ ) {
        ll p = 1;
        while( n%prime[ i ] == 0 ) {
            n /= prime[ i ];
            p++;
        }
        r = r*( ( bin_pow( prime[ i ], p )-1 )/( prime[ i ]-1 ) );
    }
    if( n != 1 ) {
        r = r*( ( bin_pow( n, 2 )-1 )/( n-1 ) );
    }
    return r;
}

ll euler_phi( vll& prime, ll n ) {
    int total_primes = int( prime.size( ) );
    ll r = n;
    for( int i = 0; prime[ i ]*prime[ i ] <= n && i < total_primes; i++ ) {
        if( n%prime[ i ] == 0 ) {
            r = r-( r/prime[ i ] );
        }
        while( n%prime[ i ] == 0 ) {
            n /= prime[ i ];
        }
    }
}

```

```

}
if( n != 1 ) {
    r = r-( r/n );
}
return r;
}

```

---

## 4.10 simpson

---

```

lf f( lf x );

```

```

inline double simpson(double fl,double fr,double fmid,double l,double r)
{ return (fl+fr+4.0*fmid)*(r-l)/6.0; }

double rsimpson(double slr,double fl,double fr,double fmid,double
l,double r) {
    double mid = (l+r)*0.5;
    double fml = f((l+mid)*0.5);
    double fmr = f((mid+r)*0.5);
    double slm = simpson(fl,fmid,fml,l,mid);
    double smr = simpson(fmid,fr,fmr,mid,r);
    if(fabs(slr-slm-smr) < EPS) return slm+smr;
    return
        rsimpson(slm,fl,fmid,fml,l,mid)+rsimpson(smr,fmid,fr,fmr,mid,r);
}

double integrate(double l,double r) {
    double mid = (l+r)*0.5;
    double fl = f(l);
    double fr = f(r);
    double fmid = f(mid);
    return rsimpson(simpson(fl,fr,fmid,l,r),fl,fr,fmid,l,r);
}

```

---

## 4.11 utilities

---

```

ll mod_mul( ll a, ll b, ll mod ) {
    ll x = 0, y = a%mod;
    while( b ) {
        if( b&1 ) {
            x = ( x+y )%mod;

```

```

    }
    y = ( y+y )%mod;
    b >>= 1;
}
return x;
}

ll mod_pow( ll b, ll e, ll mod ) {
    ll r = 1;
    while( e > 0 ) {
        if( e&1 ) {
            r = mod_mul( r, b, mod );
        }
        b = mod_mul( b, b, mod );
        e >>= 1;
    }
    return r;
}

ll bin_pow( ll b, ll e ) {
    ll r = 1;
    while( e > 0 ) {
        if( e&1 ) {
            r = r*b;
        }
        b = b*b;
        e >>= 1;
    }
    return r;
}

```

---

## 5 Misc

### 5.1 cc template

---

```

#include <bits/stdc++.h>

#define PB      push_back
#define PF      push_front
#define MP      make_pair
#define FI      first
#define SE      second

```

```

#define SIZE( A ) int( ( A ).size( ) )
#define ALL( A ) ( A ).begin( ), ( A ).end( )
#define ALLR( A ) ( A ).rbegin( ), ( A ).rend( )

```

```

using namespace std;

```

```

typedef long long      ll;
typedef unsigned long long ull;
typedef long double    lf;
typedef pair< int, int > pii;
typedef pair< ll, ll >  pll;
typedef vector< bool >  vb;
typedef vector< lf >    vd;
typedef vector< ll >    vll;
typedef vector< int >   vi;
typedef vector< pii >   vpil;

```

```

const int MAXN = int( 1e5 )+10;
const int MOD = int( 1e9 )+7;
const int oo = INT_MAX;

```

```

int main( ) {

```

```

#ifdef LOCAL
    freopen( "input", "r", stdin );
#else
    ios_base::sync_with_stdio( 0 );
    cin.tie( 0 );
#endif

```

```

    return 0;
}

```

## 5.2 magic bits

Value	Binary Sample	Meaning
x	00101100	the original x value
x & -x	00000100	extract lowest bit set
x   -x	11111100	create mask for lowest-set-bit & bits to its left
x ^ -x	11111000	create mask bits to left of lowest bit set
x & (x-1)	00101000	strip off lowest bit set --> useful to process words in 0(bits set)

	instead of 0(nbits in a word)
x   (x-1) 00101111	fill in all bits below lowest bit set
x ^ (x-1) 00000111	create mask for lowest-set-bit & bits to its right
~x & (x-1) 00000011	create mask for bits to right of lowest bit set
x   (x+1) 00101101	toggle lowest zero bit
x / (x&-x) 00001011	shift number right so lowest set bit is at bit 0

## 6 Networks

### 6.1 dilworth theorem

Chain: Set of elements in which every two are comparable.

Antichain: Set of elements in which every two are NOT comparable.

The graph is built by making an edge between U and V if U comparable to V (transitivity applies).

- The width of a finite partially ordered set S is the minimum number of chains needed to cover S, i.e. the minimum number of chains such that any element of S is in at least one of the chains.
- The width of a finite partially ordered set S is the maximum size of an antichain in S.
- The maximum size of an antichain is ( Number of nodes - Maximum Bipartite Matching)

### 6.2 dinic

```

/*
 * O( |v|^2*|e| )
 */
struct Edge {
    int from, to, cap, flow;
    Edge( int from, int to, int cap, int flow ) :
        from(from), to(to), cap(cap), flow(flow) { }
};

struct Network {
    int n;
    vector< Edge > edges;
    vector< vi > graph;

```



```

vi dist, ptr;

Network( int n ) : n(n), graph(n), dist(n), ptr(n) { }

void add_edge( int from, int to, int cap ) {
    graph[ from ].PB( SIZE(edges) );
    edges.PB( Edge( from, to, cap, 0 ) );
    graph[ to ].PB( SIZE(edges) );
    edges.PB( Edge( to, from, 0, 0 ) );
}

bool bfs( int s, int t ) {
    fill( ALL(dist), -1 );
    queue< int > q;
    q.push( s );
    dist[ s ] = 0;
    while( !q.empty() && dist[ t ] == -1 ) {
        int u = q.front(); q.pop();
        for( int i = 0; i < SIZE( graph[u] ); ++i ) {
            int id = graph[u][i], v = edges[id].to;
            if( dist[ v ] == -1 && edges[id].flow < edges[id].cap ) {
                q.push( v );
                dist[ v ] = dist[u]+1;
            }
        }
    }
    return ( dist[ t ] != -1 );
}

int dfs( int u, int t, int flow ) {
    if( !flow ) return 0;
    if( u == t ) return flow;
    for( ; ptr[u] < SIZE( graph[u] ); ++ptr[u] ) {
        int id = graph[u][ ptr[u] ], v = edges[id].to;
        if( dist[ v ] != dist[ u ]+1 ) continue;
        int pushed = dfs( v, t, min( flow, edges[id].cap-edges[id].flow ) );
        if( pushed ) {
            edges[ id ].flow += pushed;
            edges[ id^1 ].flow -= pushed;
            return pushed;
        }
    }
    return 0;
}

ll max_flow( int s, int t ) {
    ll flow = 0;
    while( bfs( s, t ) ) {

```

```

        fill( ALL(ptr), 0 );
        while( int pushed = dfs( s, t, oo ) ) {
            flow += pushed;
        }
    }
    return flow;
}
};

```

---

### 6.3 hopcroft karp

```

/*
 * O( |e|*sqrt(|v|) )
 */
struct MBM {
    int n1, n2, edges;
    vi last, prev, head, matching, dist;
    vb used, seen;

    MBM( ) :
        last(MAXN1), prev(MAXM), head(MAXM), matching(MAXN2),
        dist(MAXN1), used(MAXN1), seen(MAXN1) { }

    void init( int n1, int n2 ) {
        this->n1 = n1; this->n2 = n2;
        edges = 0;
        fill( last.begin(), last.begin()+n1, -1 );
    }

    void add_edge( int u, int v ) {
        head[ edges ] = v;
        prev[ edges ] = last[ u ];
        last[ u ] = edges++;
    }

    void bfs( ) {
        fill( dist.begin(), dist.begin()+n1, -1 );
        queue< int > q;
        for( int u = 0; u < n1; u++ ) {
            if( !used[u] ) {
                q.push( u );
                dist[ u ] = 0;
            }
        }
        while( !q.empty() ) {

```

```

int u1 = q.front(); q.pop();
for( int e = last[u1]; e >= 0; e = prev[e] ) {
    int u2 = matching[ head[e] ];
    if( u2 >= 0 && dist[u2] < 0 ) {
        dist[ u2 ] = dist[u1]+1;
        q.push( u2 );
    }
}
}
}
bool dfs( int u1 ) {
    seen[ u1 ] = true;
    for( int e = last[u1]; e >= 0; e = prev[e] ) {
        int v = head[ e ];
        int u2 = matching[ v ];
        if( u2 < 0 || ( !seen[u2] && dist[u2] == dist[u1]+1 && dfs(u2) ) ) {
            matching[ v ] = u1;
            used[ u1 ] = true;
            return true;
        }
    }
    return false;
}
int max_matching( ) {
    fill( used.begin(), used.begin()+n1, false );
    fill( matching.begin(), matching.begin()+n2, -1 );
    int ans = 0;
    while( true ) {
        bfs( );
        fill( seen.begin(), seen.begin()+n1, false );
        int f = 0;
        for( int u = 0; u < n1; u++ ) {
            if( !used[ u ] && dfs( u ) ) {
                f++;
            }
        }
        if( f == 0 ) {
            return ans;
        }
        ans += f;
    }
    return 0;
}
};

```

## 6.4 konig theorem

- In any bipartite graph, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover.
- The complement of a vertex cover in any graph is an independent set, so a minimum vertex cover is complementary to a maximum independent set.

## 6.5 max bipartite matching

```

/*
 * O( v*e ) where v = # nodes, e = # edges
 */
int n, m;
vi graph[ MAXN ];
int match[ MAXM ];
bool seen[ MAXM ];
bool dfs( int u ) {
    for( int i = 0; i < SIZE( graph[u] ); ++i ){
        int v = graph[ u ][ i ];
        if( seen[ v ] ) continue;
        seen[ v ] = true;
        if( match[ v ] == -1 || dfs( match[v] ) ) {
            match[ v ] = u;
            return true;
        }
    }
    return false;
}
int mbm( ) {
    int r = 0;
    memset( match, -1, sizeof(match) );
    for( int u = 0; u < n; ++u ) {
        memset( seen, false, sizeof(seen) );
        r += dfs( u );
    }
    return r;
}

```

## 6.6 maximum flows with edge demands

We construct a new graph  $G'=(V',E')$  from  $G$  by adding new source and target vertices  $s'$  and  $t'$ , adding edges from  $s'$  to each vertex in  $V$ , adding edges from

each vertex in  $V$  to  $t'$ , and finally adding an edge from  $t$  to  $s$ . As follows:

- $D = \sum_{u \rightarrow v \in E} d(u \rightarrow v)$
- For each vertex  $v \in V$ , we set  $c'(s' \rightarrow v) = \sum_{u \in V} d(u \rightarrow v)$  and  $c'(v \rightarrow t') = \sum_{w \in V} d(v \rightarrow w)$
- For each edge  $u \rightarrow v \in E$ , we set  $c'(u \rightarrow v) = c(u \rightarrow v) - d(u \rightarrow v)$
- Finally, we set  $c'(t \rightarrow s) = \infty$
- Note: When there is no  $s, t$  you can work without them.

In  $G'$ , the total capacity out of  $s'$  and the total capacity into  $t'$  are both equal to  $D$ . We call a flow with value exactly  $D$  a saturating flow, since it saturates all the edges leaving  $s'$  or entering  $t'$ . If  $G'$  has a saturating flow, it must be a maximum flow, so we can find it using any max-flow algorithm.

Once we've found a feasible  $(s, t)$ -flow in  $G$ , we can transform it into a maximum flow using an augmenting-path algorithm, but with one small change. To ensure that every flow we consider is feasible, we must redefine the residual capacity of an edge as follows:

$$\begin{aligned} c(u \rightarrow v) - f(u \rightarrow v), & \text{ for original edges} \\ f(v \rightarrow u) - d(v \rightarrow u), & \text{ for residual edges} \\ 0, & \text{ otherwise} \end{aligned}$$

## 6.7 minimum cost maximum flow

```

/*
 * 0( ? )
 */
struct Edge {
    int from, to, cap, cost, flow;
    Edge() {}
    Edge( int from, int to, int cap, int cost, int flow ) :
        from(from), to(to), cap(cap), cost(cost), flow(flow) {}
};

struct Network {
    int n;
    vector< Edge > edge;
    vector< vi > graph;
    vi pred, dist, phi;

```

```

Network( int n ) : n(n), graph(n), pred(n), dist(n), phi(n) {}

void add_edge( int from, int to, int cap, int cost ) {
    graph[ from ].PB( SIZE( edge ) );
    edge.PB( Edge( from, to, cap, cost, 0 ) );
    graph[ to ].PB( SIZE( edge ) );
    edge.PB( Edge( to, from, 0, -cost, 0 ) );
}

bool dijkstra( int s, int t ) {
    fill( ALL(dist), oo );
    fill( ALL(pred), -1 );
    set< pii > pq;
    dist[ s ] = 0;
    for( pq.insert( MP( dist[s], s ) ); !pq.empty(); ) {
        int u = ( *pq.begin() ).SE; pq.erase( pq.begin() );
        for( int i = 0; i < SIZE( graph[u] ); i++ ) {
            Edge& e = edge[ graph[u][i] ];
            int ndist = dist[e.from] + e.cost + phi[e.from] - phi[e.to];
            if( e.cap - e.flow > 0 && ndist < dist[e.to] ) {
                pq.erase( MP( dist[e.to], e.to ) );
                dist[ e.to ] = ndist;
                pred[ e.to ] = graph[ u ][ i ];
                pq.insert( MP( dist[e.to], e.to ) );
            }
        }
    }
    for( int i = 0; i < n; i++ ) {
        phi[ i ] = min( oo, phi[i] + dist[i] );
    }
    return ( dist[t] != oo );
}

pair< ll, ll > max_flow( int s, int t ) {
    ll mincost = 0, maxflow = 0;
    fill( ALL(phi), 0 );
    while( dijkstra( s, t ) ) {
        int flow = oo;
        for( int v = pred[t]; v != -1; v = pred[ edge[v].from ] ) {
            flow = min( flow, edge[v].cap - edge[v].flow );
        }
        for( int v = pred[t]; v != -1; v = pred[ edge[v].from ] ) {
            Edge& e1 = edge[ v ];
            Edge& e2 = edge[ v^1 ];
            mincost += e1.cost * flow;
            e1.flow += flow;

```

```

        e2.flow -= flow;
    }
    maxflow += flow;
}
return MP( maxflow, mincost );
}
};

```

## 6.8 minimum cut(bidirectional)

```

/*
 * O( |v|^3 )
 */
int n;
pair< int, vi > min_cut( vector< vi >& graph ) {
    vi used( n );
    vi cut, best_cut;
    int best_weight = -1;
    for( int phase = n-1; phase >= 0; --phase ) {
        vi w = graph[ 0 ];
        vi added = used;
        int prev, last = 0;
        for( int i = 0; i < phase; ++i ) {
            prev = last; last = -1;
            for( int j = 1; j < n; ++j ) {
                if( !added[j] && ( last == -1 || w[j] > w[last] ) ) {
                    last = j;
                }
            }
        }
        if( i == phase-1 ) {
            for( int j = 0; j < n; j++ ) {
                graph[ prev ][ j ] += graph[ last ][ j ];
            }
            for( int j = 0; j < n; j++ ) {
                graph[ j ][ prev ] = graph[ prev ][ j ];
            }
            used[ last ] = true;
            cut.PB( last );
            if( best_weight == -1 || w[last] < best_weight ) {
                best_cut = cut;
                best_weight = w[last];
            }
        }
    }
}

```

```

    else {
        for( int j = 0; j < n; j++ ) {
            w[ j ] += graph[ last ][ j ];
        }
        added[ last ] = true;
    }
}
}
return MP( best_weight, best_cut );
}

```

## 6.9 minimum cut(directional)

```

/*
 * O( |e|*flow_complexity )
 */
bool cmp_edge( const Edge &e1, const Edge &e2 ) {
    if( e1.cap != e2.cap ) return e1.cap > e2.cap;
    return e1.index < e2.index;
}
bool ok[ MAXN ];
ll get_flow( int s, int t ) {
    Network netw( n );
    for( int i = 0; i < m; ++i ) {
        if( !ok[ edges[i].index ] ) {
            netw.add_edge( edges[i].from, edges[i].to, edges[i].cap );
        }
    }
    return netw.max_flow( s, t );
}
vi min_cut( int s, int t ) {
    sort( ALL(edges), cmp_edge );
    ll flow = get_flow( s, t );
    vi ans;
    for( int i = 0; flow; ++i ) {
        ok[ edges[i].index ] = true;
        ll cur_flow = get_flow( s, t );
        ok[ edges[i].index ] = (flow-cur_flow == edges[i].cap);
        if( ok[ edges[i].index ] ) {
            ans.PB( edges[i].index );
            flow = cur_flow;
        }
    }
}

```

```
}
```

## 6.10 push relabel

```
/*
 * O( |V|^3 )
 */
struct Edge {
    int from, to, cap, flow, index;
    Edge( int from, int to, int cap, int flow, int index ) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Network {
    int n;
    vector< vector<Edge> > graph;
    vll excess;
    vi dist, active, count;
    queue< int > q;

    Network( int n ) : n(n), graph(n), excess(n), dist(n), active(n),
        count(2*n) {}

    void add_edge( int from, int to, int cap ) {
        graph[ from ].PB( Edge( from, to, cap, 0, SIZE( graph[to] ) ) );
        if( from == to ) graph[ from ].back( ).index++;
        graph[ to ].PB( Edge( to, from, 0, 0, SIZE( graph[from] )-1 ) );
    }

    void enqueue( int v ) {
        if( !active[ v ] && excess[v] > 0 ) {
            active[ v ] = true;
            q.push( v );
        }
    }

    void push( Edge &e ) {
        int amt = int( min( excess[e.from], ll(e.cap-e.flow) ) );
        if( dist[e.from] <= dist[e.to] || amt == 0 ) return ;
        e.flow += amt;
        graph[ e.to ][ e.index ].flow -= amt;
        excess[ e.to ] += amt;
        excess[ e.from ] -= amt;
        enqueue( e.to );
    }
}
```

```
void gap( int k ) {
    for( int v = 0; v < n; v++ ) {
        if( dist[v] < k ) continue;
        count[ dist[v] ]--;
        dist[ v ] = max( dist[v], n+1 );
        count[ dist[v] ]++;
        enqueue( v );
    }
}

void relabel( int v ) {
    count[ dist[ v ] ]--;
    dist[ v ] = 2*n;
    for( int i = 0; i < SIZE( graph[v] ); i++ )
        if( graph[v][i].cap-graph[v][i].flow > 0 )
            dist[ v ] = min( dist[v], dist[ graph[v][i].to ]+1 );
    count[ dist[v] ]++;
    enqueue( v );
}

void discharge( int v ) {
    for( int i = 0; excess[v] > 0 && i < SIZE( graph[v] ); i++ )
        push( graph[v][i] );
    if( excess[v] > 0 ) {
        if( count[ dist[v] ] == 1 )
            gap( dist[v] );
        else
            relabel( v );
    }
}

ll max_flow( int s, int t ) {
    count[ 0 ] = n-1;
    count[ n ] = 1;
    dist[ s ] = n;
    active[ s ] = active[ t ] = true;
    for( int i = 0; i < SIZE( graph[s] ); i++ ) {
        excess[ s ] += graph[ s ][ i ].cap;
        push( graph[s][i] );
    }
    while( !q.empty( ) ) {
        int v = q.front( ); q.pop( );
        active[ v ] = false;
        discharge( v );
    }
    ll totflow = 0;
    for( int i = 0; i < SIZE( graph[s] ); i++ )
        totflow += graph[s][i].flow;
}
```

```

    return totflow;
}
};

```

## 7 Strings

### 7.1 aho corasick

```

/*
 * O( |t|+SUM( |p_i| )+matches ) where t is a text and p_i are the
 * patterns
 */

const int alphabet = 26;
int fail[ MAX_N ];
int mv( int node, int c ){
    while( !trie[ node ][ c ] ){
        node = fail[ node ];
    }
    return trie[ node ][ c ];
}

void build_aho_corasick( ) {
    memset( fail, 0, sizeof( fail ) );
    queue< int > q;
    for( int i = 0; i < alphabet; i++ ) {
        if( trie[1][i] ) {
            q.push( trie[1][i] );
            fail[ trie[1][i] ] = 1;
        }
        else {
            trie[1][i] = 1;
        }
    }
    while( !q.empty( ) ) {
        int node = q.front( ); q.pop( );
        for( int i = 0; i < alphabet; i++ ){
            if( trie[node][i] ) {
                fail[ trie[node][i] ] = mv( fail[ node ], i );
                q.push( trie[node][i] );
            }
        }
    }
}

```

```

}

```

### 7.2 hashing

```

/*
 * gen_mod( ) generates two random primes ~10^9
 * fill_hash( acc, t ) acc[ i ] ( 1 <= i <= |t| ) stores the hash of t[0,
 * i-1].
 * get_hash( acc, l, r ) return the hash [ l, r ] using the acc array.
 */

void gen_mod( ) {
    srand( time( nullptr ) );
    for( int i = 0; i < 2; ++i ) {
        int mod = int(1e9) + rand()%int(5e6);
        while( !is_prime( mod ) ) {
            mod++;
        }
        cout << mod << '\n';
    }
}

typedef pair< int, int > mint;
const int MOD[ ] = { 1001864327, 1001265673 };
const mint BASE( 256, 256 ), ZERO( 0, 0 ), ONE( 1, 1 );
inline int add( int a, int b, const int& mod ) { return ( a+b >= mod ) ?
    a+b-mod : a+b; }
inline int sbt( int a, int b, const int& mod ) { return ( a-b < 0 ?
    a-b+mod : a-b ); }
inline int mul( int a, int b, const int& mod ) { return ll(a)*ll(b) %
    ll(mod); }

inline ll operator ! ( const mint a ) { return (ll(a.FI)<<32)|ll(a.SE); }
inline mint operator + ( const mint a, const mint b ) {
    return mint( add( a.FI, b.FI, MOD[0] ), add( a.SE, b.SE, MOD[1] ) );
}
inline mint operator - ( const mint a, const mint b ) {
    return mint( sbt( a.FI, b.FI, MOD[0] ), sbt( a.SE, b.SE, MOD[1] ) );
}
inline mint operator * ( const mint a, const mint b ) {
    return mint( mul( a.FI, b.FI, MOD[0] ), mul( a.SE, b.SE, MOD[1] ) );
}

```

```

void fill_hash( mint* acc, const string& t ) {
    acc[ 0 ] = ZERO;
    for( int i = 1; i <= n; ++i ) {
        acc[ i ] = acc[ i-1 ]*BASE + val[ t[i-1] ];
    }
}

mint get_hash( mint* acc, int l, int r ) {
    return acc[ r+1 ] - acc[ l ]*base[ r-l+1 ];
}

```

### 7.3 kmp automaton

```

/*
 * O( n*alphabet ) where n = |text|
 * Returns a matrix such that a[ i ][ j ] is equal to the transition if
 * I'm at i-th position and see the character j.
 */

const int alphabet = 256;
vector< vi > kmp_automaton( string t ) {
    int len = SIZE( t );
    vi phi = kmp( t );
    vector< vi > aut( len, vi( alphabet ) );
    for( int i = 0; i < len; ++i ) {
        for( int c = 0; c < alphabet; ++c ) {
            if( i > 0 && char(c) != t[ i ] ) {
                aut[ i ][ c ] = aut[ phi[i-1] ][ c ];
            } else {
                aut[ i ][ c ] = i + ( char(c) == t[ i ] );
            }
        }
    }
    return aut;
}

```

### 7.4 kmp

```

/*
 * O( n ) where n = |text|
 * For each i, phi[ i ] is equal to the longest prefix that also is a
 * suffix ending at i.

```

```

*/

vi kmp( string t ) {
    int len = SIZE( t );
    vi phi( len );
    phi[ 0 ] = 0;
    for( int i = 1, j = 0; i < len; ++i ) {
        while( j > 0 && t[ i ] != t[ j ] ) {
            j = phi[ j-1 ];
        }
        if( t[ i ] == t[ j ] ) {
            ++j;
        }
        phi[ i ] = j;
    }
    return phi;
}

```

### 7.5 manacher

```

/*
 * O( n ) where n = |text|
 * Returns a vector with size equal to 2*|text|. For each i in such
 * vector, p[ i ] is equal to the maximum palindrome centered at this
 * position.
 */

vi manacher( string t ) {
    int len = SIZE( t );
    vi p( 2*len );
    int C = -1, R = -1;
    int n = (len-1)<<1;
    for( int i = 0; i <= n; i++ ) {
        int j = 2*C-i;
        p[ i ] = ( R >= i ) ? min( R-i+1, p[ j ] ) : !( i%2 );
        int l = (i-p[ i ])>>1;
        int r = (i+p[ i ]+1)>>1;
        while( l >= 0 && r < len && t[ l ] == t[ r ] ) {
            p[ i ] += 2;
            l--; r++;
        }
        int ri = p[ i ] ? ((i+p[ i ])>>1)<<1 : i;
        if( ri > R ) {

```

```

        C = i;
        R = ri;
    }
}
return p;
}

```

## 7.6 minimum expression

```

/*
 * O( n ) where n = |text|
 * Find the lexicographically minimal string rotation.
 */

```

```

int minimum_expression( string t ) {
    t = t+t;
    int len = SIZE( t );
    int i = 0, j = 1, k = 0;
    while( i+k < len && j+k < len ) {
        if( t[ i+k ] == t[ j+k ] ) {
            k++;
        }
        else if( t[ i+k ] > t[ j+k ] ) {
            i = i+k+1;
            if( i <= j ) {
                i = j+1;
            }
            k = 0;
        }
        else {
            j = j+k+1;
            if( j <= i ) {
                j = i+1;
            }
            k = 0;
        }
    }
    return min( i, j );
}

```

## 7.7 suffix array

```

/*
 * O( n*log(n) ) where n = |text|
 * sa[i] contains the starting position of the i-th smallest suffix in t,
 * ensuring that for all 1 < i <= n, t[sa[i-1], n] < t[sa[i], n] holds.
 * O( n ) where n = |text|
 * lcp[i] stores the lengths of the longest common prefixes between all
 * pairs of consecutive suffixes in a sorted suffix array (needs sa).
 */

```

```

int n, mx;
string t;
int pos[ MAXN ], cnt[ MAXN ];
int aux_sa[ MAXN ], aux_pos[ MAXN ];
int sa[ MAXN ], lcp[ MAXN ];

```

```

bool check( int i, int gap ) {
    if( pos[ sa[i-1] ] != pos[ sa[i] ] ) {
        return true;
    }
    if( sa[ i-1 ]+gap < n && sa[ i ]+gap < n ) {
        return ( pos[ sa[i-1]+gap ] != pos[ sa[i]+gap ] );
    }
    return true;
}

void radix_sort( int k ) {
    for( int i = 0; i < mx; ++i ) {
        cnt[ i ] = 0;
    }
    for( int i = 0; i < n; i++ ) {
        cnt[ (i+k < n) ? pos[ i+k ]+1 : 1 ]++;
    }
    for( int i = 1; i < mx; i++ ) {
        cnt[ i ] += cnt[ i-1 ];
    }
    for( int i = 0; i < n; i++ ) {
        aux_sa[ cnt[ (sa[ i ]+k < n) ? pos[ sa[i]+k ] : 0 ]++ ] = sa[ i ];
    }
    for( int i = 0; i < n; i++ ) {
        sa[ i ] = aux_sa[ i ];
    }
}

void build_sa( ) {
    for( int i = 0; i < n; i++ ) {
        sa[ i ] = i;
    }
}

```



```

    pos[ i ] = t[ i ];
}
for( int gap = 1; gap < n; gap <= 1 ) {
    radix_sort( gap );
    radix_sort( 0 );
    aux_pos[ sa[0] ] = 0;
    for( int i = 1; i < n; i++ ) {
        aux_pos[ sa[i] ] = aux_pos[ sa[i-1] ] + check( i, gap );
    }
    for( int i = 0; i < n; i++ ) {
        pos[ i ] = aux_pos[ i ];
    }
    if( pos[ sa[n-1] ] == n-1 ) {
        break;
    }
}
}
void build_lcp( ) {
    int k = 0;
    lcp[ 0 ] = 0;
    for( int i = 0; i < n; i++ ) {
        if( pos[ i ] == 0 ) {
            continue;
        }
        while( t[ i+k ] == t[ sa[ pos[i]-1 ]+k ] ) {
            k++;
        }
        lcp[ pos[ i ] ] = k;
        k = max( 0, k-1 );
    }
}
void build( string s ) {
    n = SIZE( s );
    t = s+"#";
    mx = max( 256, n );
    build_sa( );
    build_lcp( );
}

```

---

## 7.8 z algorithm

```

/*
 * O( n ) where n = |text|

```

```

* For each i, z[ i ] is equal to the longest substring starting at i
  that is prefix of the text.
*/

```

```

vi z_algorithm( string str ) {
    int len = SIZE( str );
    vi z( len );
    z[ 0 ] = 0;
    for( int i = 1, l = 0, r = 0; i < len; ++i ) {
        if( i <= r ) z[ i ] = min( r-i+1, z[ i-l ] );
        while( i+z[ i ] < len && str[ z[i] ] == str[ i+z[i] ] ) z[ i ]++;
        if( i+z[ i ]-1 > r ) {
            l = i;
            r = i+z[ i ]-1;
        }
    }
    return z;
}

```

---