Team notebook

Universidad Nacional de Colombia

April 6, 2024

Contents

1 Data structures

1.1 Centroid decomposition

```
namespace decomposition {
  int cnt[N], depth[N], f[N]; //if depth != 0
      means was removed from the tree while
      decomposing
  int dfs (int u, int p = -1) {
    cnt[u] = 1;
    for (int v : g[u])
      if (!depth[v] && v != p)
        cnt[u] += dfs(v, u);
    return cnt[u];
  }
  int get_centroid (int u, int r, int p = -1) {
    for (int v : g[u])
      if (!depth[v] && v != p && cnt[v] > r)
        return get_centroid(v, r, u);
    return u;
  }
  //depth of all tree's centroid is 1 and father
      is 0 (you can set up this when call this)
  int decompose (int u, int d = 1) {
    int centroid = get_centroid(u, dfs(u) >> 1);
    depth[centroid] = d; //remove this node from
        component
    //add here magic function to count properties
        on paths
    for (int v : g[centroid])
      if (!depth[v])
        f[decompose(v, d + 1)] = centroid;
    return centroid;
  }
  int lca (int u, int v) { //lca on centroid tree
    for (; u != v; u = f[u])
      if (depth[v] > depth[u])
        swap(u, v);
    return u;
  }
}
```

1.2 Fenwick tree

```
int lower_find(int val) { //last value < or <= to
    val
  int idx = 0;
  for(int i = 31-__builtin_clz(n); i >= 0; --i) {
    int nidx = idx | (1 << i);
    if(nidx <= n && bit[nidx] <= val) { //change
        <= to <
      val -= bit[nidx];
      idx = nidx;
    }
  }
  return idx;
}
```

1.3 Heavy light decomposition

```
//Complexity: O(|N|)
int idx; //top is father of the chain, up is
    father of a node
vector<int> len, depth, in, out, top, up;
int dfs_len( int u, int p, int d ) {
  up[u] = p; depth[u] = d;
  int sz = 1;
  for( auto& v : g[u] ) {
    if( v == p ) continue;
    sz += dfs_len(v, u, d+1);
    if(len[ g[u][0] ] <= len[v]) swap(g[u][0], v);
  }
  return len[u] = sz;
}
void dfs_hld( int u, int p = 0 ) {
  in[u] = idx++;
  narr[ in[u] ] = val[u]; //to initialize the
      segment tree
  for( auto& v : g[u] ) {
    if( v == p ) continue;
    top[v] = (v == g[u][0] ? top[u] : v);
    dfs_hld(v, u);
  }
  out[u] = idx-1;
}
void update_hld( int u, int val ) {
  update_DS(in[u], val);
}
data query_hld( int u, int v ) {
  data val = NULL_DATA;
  while( top[u] != top[v] ) {
    if( depth[ top[u] ] < depth[ top[v] ] )
        swap(u, v);
    val = val+query_DS(in[ top[u] ], in[u]);
    u = up[ top[u] ];
  }
  if( depth[u] > depth[v] ) swap(u, v);
  val = val+query_DS(in[u], in[v]);
  return val;
//when updates are on edges use:
//  val[v] is cost_edge(up[v], v), mind root's
    cost
//  if(depth[u] == depth[v]) return val;
//  val = val+query_DS(in[u] + 1, in[v]);
}
void build(int n, int root) {
  top = len = in = out = up = depth =
      vector<int>(n+1);
  idx = 1; //DS index [1, n]
  dfs_len(root, root, 0);
  top[root] = root;
  dfs_hld(root, root);
  //initialize DS
}
```

1.4 Iterative Segment Tree

```
//Complexity: O(|N|*log|N|)
struct info { int val; };
info merge(info &a, info &b) {
  return {a.val + b.val};
}
info NEUTRAL = {0};
struct segtree { //for point update and range
    queries, supports left to right merge
  int n; //0-indexed
  vector<info> t;
  segtree(int n, vector<int> &v) : n(n), t(2*n) {
    for(int i = 0; i < n; i++) t[i+n] = {v[i]};
    for(int i = n-1; i > 0; --i) t[i] =
        merge(t[i<<1], t[i<<1|1]);
  }
  info query(int l, int r) {
    info ans_l = NEUTRAL, ans_r = NEUTRAL;
    for(l += n, r += n+1; l < r; l >>= 1, r >>=
        1) {
      if(l&1) ans_l = merge(ans_l, t[l++]);
      if(r&1) ans_r = merge(t[--r], ans_r);
    }
    return merge(ans_l, ans_r);
  }
  void modify(int p, int x) {
    for(t[p += n] = {x}; p >>= 1; ) t[p] =
        merge(t[p<<1], t[p<<1|1]);
  }
};
```

1.5 Mo's

```
//Complexity: O(|N+Q|*sqrt(|N|)*|ADD/DEL|)
//Requires add(), delete() and get_ans()
struct query {
  int l, r, idx;
  query (int l, int r, int idx) : l(l), r(r),
      idx(idx) {}
};
int S; //s = sqrt(n)
bool cmp (const query &a, const query &b) {
  int A = a.l/S, B = b.l/S;
  if (A != B) return A < B;
  return A & 1 ? a.r > b.r : a.r < b.r;
}
S = sqrt(n); //n = size of array
sort(q.begin(), q.end(), cmp);
int l = 0, r = -1;
for (int i = 0; i < q.size(); ++i) {
  while (r < q[i].r) add(++r);
  while (l > q[i].l) add(--l);
  while (r > q[i].r) del(r--);
  while (l < q[i].l) del(l++);
  ans[q[i].idx] = get_ans();
}
```

1.6 Order statistics

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>,
    rb_tree_tag,
tree_order_statistics_node_update> ordered_set;
//methods
tree.find_by_order(k) //returns pointer to the
    k-th smallest element
tree.order_of_key(x) //returns how many elements
    are smaller than x
//if element does not exist
tree.end() == tree.find_by_order(k) //true
```

1.7 Persistent segment tree

```
//Complexity: O(|N|*log|N|)
struct node {
  node *left, *right;
  int v;
  node() : left(this), right(this), v(0) {}
  node(node *left, node *right, int v) :
    left(left), right(right), v(v) {}
  node* update(int l, int r, int x, int value) {
    if (l == r) return new node(nullptr, nullptr,
        v + value);
    int m = (l + r) / 2;
    if (x <= m)
      return new node(left->update(l, m, x,
          value), right, v + value);
    return new node(left, right->update(m + 1, r,
        x, value), v + value);
  }
};
```

1.8 Rmq

```
//Complexity: O(|N|*log|N|)
struct rmq {
  vector<vector<int>> table;
  rmq(vector<int> &v) : table(20,
      vector<int>(v.size())) {
    int n = v.size();
    for (int i = 0; i < n; i++) table[0][i] =
        v[i];
    for (int j = 1; (1<<j) <= n; j++)
      for (int i = 0; i + (1<<j-1) < n; i++)
        table[j][i] = min(table[j-1][i],
            table[j-1][i + (1<<j-1)]);
  }
  int query(int a, int b) {
    int j = 31 - __builtin_clz(b-a+1);
    return min(table[j][a], table[j][b-(1<<j)+1]);
  }
};
```

1.9 Sack

```
//Complexity: |N|*log(|N|)
int dfs(int u, int p = -1) {
  who[t] = u; fr[u] = t++;
```

```
  pii best = {0, -1};
  int sz = 1;
  for(auto v : g[u])
    if(v != p) {
      int cur_sz = dfs(v, u);
      sz += cur_sz;
      best = max(best, {cur_sz, v});
    }
  to[u] = t-1;
  big[u] = best.second;
  return sz;
}
void add(int u, int x) { //x == 1 add, x == -1
    delete
  cnt[u] += x;
}
void go(int u, int p = -1, bool keep = true) {
  for(auto v : g[u])
    if(v != p && v != big[u])
      go(v, u, 0);
  if(big[u] != -1) go(big[u], u, 1);
  //add all small
  for(auto v : g[u])
    if(v != p && v != big[u])
      for(int i = fr[v]; i <= to[v]; i++)
        add(who[i], 1);
  add(u, 1);
  ans[u] = get(u);
  if(!keep)
    for(int i = fr[u]; i <= to[u]; i++)
      add(who[i], -1);
}
void solve(int root) {
  t = 0;
  dfs(root);
  go(root);
}
```

1.10 Segment tree 2D

```
//Complexity: log(|N|)^2 per operation
struct info {
  ll val;
};
info merge(info a, info b) {
  return {a.val > b.val ? a.val : b.val};
}
info NEUTRAL = {LLONG_MIN};
struct segtree_2d {
  int n, m; //0-indexed
  vector<vector<info>> t;
  segtree_2d(int n, int m) : n(n), m(m), t(2*n,
      vector<info>(2*m, NEUTRAL)) {}
  segtree_2d(int n, int m, vector<vector<info>>
      &v) : n(n), m(m), t(2*n, vector<info>(2*m,
      NEUTRAL)) {
    for(int i = 0; i < n; i++)
      for(int j = 0; j < m; j++)
        t[i+n][j+m] = v[i][j];
    for(int i = 0; i < n; i++)
      for(int j = m-1; j; j--)
        t[i+n][j] = merge(t[i+n][j<<1],
            t[i+n][j<<1|1]);
    for(int i = n-1; i; i--)
      for(int j = 0; j < 2*m; j++)
        t[i][j] = merge(t[i<<1][j],t[i<<1|1][j]);
  }
  info get(int x1, int y1, int x2, int y2) {
    info ans = NEUTRAL;
    vector<int> pos(2);
    for(x1 += n, x2 += n+1; x1 < x2; x1 >>= 1, x2
        >>= 1) {
      int q = 0;
      if(x1&1) pos[q++] = x1++;
      if(x2&1) pos[q++] = --x2;
      for(int i = 0; i < q; i++) {
        for(int t1 = m+y1, t2 = m+y2+1, id =
            pos[i]; t1 < t2; t1 >>= 1, t2 >>= 1) {
          if(t1&1) ans = merge(ans, t[id][t1++]);
          if(t2&1) ans = merge(ans, t[id][--t2]);
        }
      }
    }
    return ans;
  }
  void update(int x, int y, info v) {
    t[x+n][y+m] = v;
    for(int j = y+m; j > 1; j >>= 1)
      t[x+n][j>>1] = merge(t[x+n][j], t[x+n][j^1]);
    for(int i = x+n; i > 1; i >>= 1)
      for(int j = y+m; j; j >>= 1)
        t[i>>1][j] = merge(t[i][j],t[i^1][j]);
  }
};
```

1.11 Treap

```
//Complexity: log(|N|) per operation
uniform_int_distribution<ll> rnd(0, LLONG_MAX);
typedef long long T;
struct treap {
  treap *left, *right, *father;
  ll prior;
  int sz, idx;
  T value, lazy_sum, sum;
  treap(T x) {
    left = right = father = NULL;
    prior = rnd(rng);
    sz = 1;
    value = sum = x;
    lazy_sum = 0;
  }
};
int cnt(treap* t) { return !t ? 0 : t->sz; }
T sum(treap* t) { return !t ? 0 : t->sum; }
T value(treap* t) { return !t ? 0 : t->value; }
void propagate(treap* t) {
  if(t && t->lazy_sum) {
    if(t->left) t->left->lazy_sum += t->lazy_sum;
    if(t->right) t->right->lazy_sum +=
        t->lazy_sum;
    t->sum += cnt(t) * t->lazy_sum;
    t->value += t->lazy_sum;
    t->lazy_sum = 0;
  }
}
void update(treap *t) {
  propagate(t->left);
```

```
  propagate(t->right);
  t->sz = cnt(t->left) + cnt(t->right) + 1;
  t->sum = sum(t->left) + sum(t->right) +
      value(t);
  if(t->left) t->left->father = t;
  if(t->right) t->right->father = t;
}
void add_value(treap *t, T v) {
  t->value += v;
  update(t);
}
void add_lazy_sum(treap *t, T v) {
  t->lazy_sum += v;
  update(t);
}
pair<treap*, treap*> split(treap* t, int
    left_count) {
  if(!t) return {NULL, NULL};
  propagate(t);
  if(cnt(t->left) >= left_count) {
    auto got = split(t->left, left_count);
    t->left = got.second;
    update(t);
    return {got.first, t};
  } else {
    left_count = left_count-cnt(t->left)-1;
    auto got = split(t->right, left_count);
    t->right = got.first;
    update(t);
    return {t, got.second};
  }
}
treap* merge(treap *s, treap *t) {
  if(!s) return t;
  if(!t) return s;
  propagate(s);
  propagate(t);
  if(s->prior <= t->prior) {
    s->right = merge(s->right, t);
    update(s);
    return s;
  } else {
    t->left = merge(s, t->left);
    update(t);
    return t;
  }
};
void print(treap *x) {
  if(!x) return;
  propagate(x);
  print(x->left);
  cout << value(x) << ", ";
  print(x->right);
}
int find_left_count(treap* root, treap* x) { //x
    not inclusive
  if(!x) return 0;
  int ans = cnt(x->left);
  while(x != root) {
    treap *par = x->father;
    if(par->right == x) ans += cnt(par->left)+1;
    x = par;
  }
  return ans;
}
treap *root = NULL;
```

1.12 lichaor

```
typedef long long type;
struct line {
  type m, b;
  type eval (type x) {
    return m * x + b;
  }
};
//M = 4 * (number of updates) * (log R) where R
    is the range of the segtree
//oo is the max abs value that the function may
    take
const type R = 1e9+5;
const int M = 1e7;
const type oo = 2e18;
struct li_chao {
  //see comments for min
  int nodes;
  line lines[M];
  int lf[M], rg[M];
  void ini () {
    nodes = 0;
    memset(lf, -1, sizeof lf);
    memset(rg, -1, sizeof rg);
    lines[0] = {0, -oo}; //change to {0, oo};
  }
  int add_line (int node, type l, type r, line
      nw) {
    if (node == -1) {
      lines[++nodes] = nw;
      return nodes;
    }
    type m = (l + r) >> 1;
    bool lef = nw.eval(l) > lines[node].eval(l);
        //change > to <
    bool mid = nw.eval(m) > lines[node].eval(m);
        //change > to <
    if (mid) swap(nw, lines[node]);
    lines[++nodes] = lines[node];
    lf[nodes] = lf[node];
    rg[nodes] = rg[node];
    int sv = nodes;
    if (r == l) return sv;
    if (lef != mid) lf[sv] = add_line(lf[node],
        l, m, nw);
    else rg[sv] = add_line(rg[node], m + 1, r,
        nw);
    return sv;
  }
  type get(int node, type l, type r, type x) {
    if (node == -1) return -oo; //change to oo
    type m = (l + r) / 2;
    if(l == r) {
        return lines[node].eval(x);
    } else if(x < m) {
        return max(lines[node].eval(x),
            get(lf[node], l, m, x)); //change max
            to min
    } else {
        return max(lines[node].eval(x),
            get(rg[node], m + 1, r, x)); //change
            max to min
```

```
    }
  }
};
```

2 Dp optimization

2.1 Convex hull trick dynamic

```
//Complexity: O(|N|*log(|N|))
typedef ll T;
const T is_query = -(1LL<<62);
struct line {
	T m, b;
	mutable multiset<line>::iterator it, end;
	bool operator < (const line &rhs) const {
		if(rhs.b != is_query) return m <
		    rhs.m;
		auto s = next(it);
		if(s == end) return 0;
		return b - s->b < (long
		    double)(s->m - m) * rhs.m;
	}
};
struct CHT : public multiset<line> {
	bool bad(iterator y) {
		auto z = next(y);
		if(y == begin()) {
			if(z == end()) return false;
			return y->m == z->m && y->b
			    <= z->b;
		}
		auto x = prev(y);
		if(z == end()) return y->m == x->m
		    && y->b == x->b;
		return (long double)(x->b -
		    y->b)*(z->m - y->m) >= (long
		    double)(y->b - z->b)*(y->m -
		    x->m);
	}
	void add(T m, T b) {
		auto y = insert({m, b});
		y->it = y; y->end = end();
		if(bad(y)) { erase(y); return; }
		while(next(y) != end() &&
		    bad(next(y))) erase(next(y));
		while(y != begin() &&
		    bad(prev(y)))erase(prev(y));
	}
	T eval(T x) { //for maximum
		auto l = *lower_bound({x,
		    is_query});
		return l.m*x+l.b;
	}
}; //for minimum, you must change (b, m) to (-b,
    -m)
```

2.2 Convex hull trick

```
struct line {
  ll m, b;
  ll eval (ll x) { return m*x + b; }
};
struct cht {
  vector<line> lines;
  vector<lf> inter;
  int n;
  lf get_inter(line &a, line &b) { return lf(b.b
      - a.b) / (a.m - b.m); }
  inline bool ok(line &a, line &b, line &c) {
    return lf(a.b-c.b) / (c.m-a.m) > lf(a.b-b.b)
        / (b.m-a.m);
  }
  void add(line l) {
    n = lines.size();
    if(n && lines.back().m == l.m &&
        lines.back().b >= l.b) return;
    if(n == 1 && lines.back().m == l.m &&
        lines.back().b < l.b) lines.pop_back(),
        n--;
    while(n >= 2 && !ok(lines[n-2], lines[n-1],
        l)) {
      n--;
      lines.pop_back(); inter.pop_back();
    }
    lines.push_back(l); n++;
    if(n >= 2)
        inter.push_back(get_inter(lines[n-1],
        lines[n-2]));
  }
  ll get_max(lf x) {
    if(lines.size() == 0) return LLONG_MIN;
    if(lines.size() == 1) return lines[0].eval(x);
    int pos = lower_bound(inter.begin(),
        inter.end(), x) - inter.begin();
    return lines[pos].eval(x);
  }
}; //for max: order slops non-decreasing, for
   min: order slops non-descending
```

2.3 Divide and conquer

```
//Complexity: O(|N|*|K|*log|N|))
//******* Theory *******
//dp[k][i]=min(dp[k1][j]+C[i][j]), j < i
//opt[k][i]    opt[k][i+1].
//A sufficient (but not necessary) condition for
    above is
//C[a][c]  + C  [b][d]    C[a][d]  + C  [b][c]
    where  a <   b   <   c   < d  .
void go(int k, int l, int r, int opl, int opr) {
  if(l > r) return;
  int mid = (l + r) / 2, op = -1;
  ll &best = dp[mid][k];
  best = INF;
  for(int i = min(opr, mid); i >= opl; i--) {
    ll cur = dp[i][k-1] + cost(i+1, mid);
    if(best > cur) {
      best = cur;
      op = i;
    }
  }
  go(k, l, mid-1, opl, op);
  go(k, mid+1, r, op, opr);
}
```

2.4 Knuth

```
//Complexity: O(|N|^2))
//******* Theory *******
//dp[i][j]= min(dp[i][k]+dp[k][j])+C[i][j], i<k<j
//where opt[i][ j1 ]     opt[i][j]    opt[i+1][j].
//sufficient (but not necessary) condition for
    above is
//C[a][c]  + C  [b][d]    C[a][d]  + C  [b][c] and
    C[b][c]   C[a][d] where
              abcd             .
for(int i = 1; i <= n; i++) {
  opt[i][i] = i;
  dp[i][i] = sum[i] - sum[i-1];
}
for(int len = 2; len <= n; len++)
  for(int l = 1; l+len-1 <= n; l++) {
    int r = l+len-1;
    dp[l][r] = oo;
    for(int i = opt[l][r-1]; i <= opt[l+1][r];
        i++) {
      ll cur = dp[l][i-1] + dp[i+1][r] + sum[r] -
          sum[l-1];
      if(cur < dp[l][r]) {
        dp[l][r] = cur;
        opt[l][r] = i;
      }
    }
  }
```

3 Formulas

3.1 Burnside's lemma

$$\#orbitas = \frac{1}{|G|} \sum_{g \in G} |fix(g)|$$

1. **G**: Las acciones que se pueden aplicar sobre un elemento, incluyendo la identidad, eg. Shift 0 veces, Shift 1 veces...
2. **Fix(g)**: Es el número de elementos que al aplicar g vuelven a ser ser ellos mismos
3. **Órbita**: El conjunto de elementos que pueden ser iguales entre si al aplicar alguna de las acciones de G

3.2 Combinatorics

- Hockey-stick identity $\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1}$

3.3 Law of sines and cosines

- a, b, c: lenghts, A, B, C: opposite angles, d: circumcircle
- $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)} = d$
- $c^2 = a^2 + b^2 - 2ab\cos(C)$

3.4 Pythagorean triples ($a^2 + b^2 = c^2$)

- Given an arbitrary pair of integers m and n with $m > n > 0$:
  $a = m^2 - n^2$, $b = 2mn$, $c = m^2 + n^2$
- The triple generated by Euclid's formula is primitive if and only if m and n are coprime and not both odd.
- To generate all Pythagorean triples uniquely:
  $a = k(m^2 - n^2)$, $b = k(2mn)$, $c = k(m^2 + n^2)$

4 Geometry

4.1 2DLibrary

```
typedef long double lf;
const lf EPS = 1e-8L, INF = 5e9;
const lf E0 = 0.0L;//Keep = 0 for integer
    coordinates, otherwise = EPS
enum {OUT,IN,ON};
struct pt {
  lf x,y;
  pt(){}
  pt(lf a , lf b): x(a), y(b){}
  pt operator - (const pt &q ) const {
    return {x - q.x , y - q.y };
  }
  pt operator + (const pt &q ) const {
    return {x + q.x , y + q.y };
  }
  pt operator * (const lf &t ) const {
    return {x * t , y * t };
  }
  pt operator / (const lf &t ) const {
    return {x / t , y / t };
  }
  bool operator < ( const pt & q ) const {
    if( fabsl( x - q.x ) > E0 ) return x < q.x;
    return y < q.y;
  }
  void normalize() {
    lf norm = hypotl( x, y );
    if( fabsl( norm ) > EPS )
      x /= norm, y /= norm;
  }
};
pt rot90( pt p ) { return { -p.y, p.x }; }
pt rot( pt p, lf w ) {
  return { cosl( w ) * p.x - sinl( w ) * p.y,
      sinl( w ) * p.x + cosl( w ) * p.y };
}
lf norm2 ( pt p ) { return p.x * p.x + p.y * p.y;
    }
lf dis2 ( pt p, pt q ) { return norm2(p-q); }
lf norm ( pt p ) { return hypotl ( p.x, p.y ); }
lf dis( pt p, pt q ) { return norm( p - q ); }
lf dot( pt p, pt q ) { return p.x * q.x + p.y *
    q.y; }
lf cross( pt p, pt q ) { return p.x * q.y - q.x *
    p.y ; }
lf orient( pt a, pt b, pt c ) { return cross( b -
    a, c - a ); };
bool in_angle( pt a, pt b, pt c, pt p ) {
  //assert( fabsl( orient( a, b, c ) ) > E0 );
  if( orient( a, b, c ) < -E0 )
```

```
    return orient( a, b, p ) >= -E0 || orient( a,
        c, p ) <= E0;
  return orient( a, b, p ) >= -E0 && orient( a,
      c, p ) <= E0;
}
struct line {
  pt nv;
  lf c;
  line( pt _nv, lf _c ) : nv( _nv ), c( _c ) {}
  line ( pt p, pt q ) {
    nv = { p.y - q.y, q.x - p.x };
    c = -dot( p, nv );
  }
  lf eval( pt p ) { return dot( nv, p ) + c; }
  lf distance2( pt p ) {
    return eval( p ) / norm2( nv ) * eval( p );
  }
  lf distance( pt p ) {
    return fabsl( eval( p ) ) / norm( nv );
  }
  pt projection( pt p ) {
    return p - nv * ( eval( p ) / norm2( nv ) );
  }
};
pt lines_intersection( line a, line b ) {
  lf d = cross( a.nv, b.nv );
  //assert( fabsl( d ) > E0 );
  lf dx = a.nv.y * b.c - a.c * b.nv.y;
  lf dy = a.c * b.nv.x - a.nv.x * b.c;
  return { dx / d, dy / d };
}
line bisector( pt a, pt b ) {
  pt nv = ( b - a ), p = ( a + b ) * 0.5L;
  lf c = -dot( nv, p );
  return line( nv, c );
}
struct Circle {
  pt center;
  lf r;
  Circle( pt p, lf rad ) : center( p ), r( rad )
      {};
  Circle( pt p, pt q ) {
    center = ( p + q ) * 0.5L;
    r = dis( p, q ) * 0.5L;
  }
  Circle( pt a, pt b, pt c ) {
    line lb = bisector( a, b ), lc = bisector( a,
        c );
    center = lines_intersection( lb, lc );
    r = dis( a, center );
  }
  int contains( pt &p ) {
    lf det = r * r - dis2( center, p );
    if( fabsl( det ) <= E0 ) return ON;
    return ( det > E0 ? IN : OUT );
  }
};
vector< pt > circle_line_intersection( Circle c,
    line l ) {
  lf h2 = c.r * c.r - l.distance2( c.center );
  if( fabsl( h2 ) < EPS ) return { l.projection(
      c.center ) };
  if( h2 < 0.0L ) return {};
  pt dir = rot90( l.nv );
  pt p = l.projection( c.center );
  lf t = sqrtl( h2 / norm2( dir ) );
  return { p + dir * t, p - dir * t };
}
vector< pt > circle_circle_intersection( Circle
    c1, Circle c2 ) {
  pt dir = c2.center - c1.center;
  lf d2 = dis2( c1.center, c2.center );
  if( d2 <= E0 ) { //assert( fabsl( c1.r - c2.r )
      > E0 );
    return {};
  }
  lf td = 0.5L * ( d2 + c1.r * c1.r - c2.r * c2.r
      );
  lf h2 = c1.r * c1.r - td / d2 * td;
  pt p = c1.center + dir * ( td / d2 );
  if( fabsl( h2 ) < EPS ) return { p };
  if( h2 < 0.0L ) return {};
  pt dir_h = rot90(dir) * sqrtl( h2 / d2 );
  return { p + dir_h, p - dir_h };
}
vector< pt > convex_hull( vector< pt > v ) {
  sort( v.begin(), v.end() );//remove repeated
      points if needed
  const int n = v.size();
  if( n < 3 ) return v;
  vector< pt > ch( 2 * n );
  int k = 0;
  for( int i = 0; i < n; ++ i ) {
    while( k > 1 && orient( ch[k-2], ch[k-1],
        v[i] ) <= E0 )
      --k;
    ch[k++] = v[i];
  }
  const int t = k;
  for( int i = n - 2; i >= 0; -- i ) {
    while( k > t && orient( ch[k-2], ch[k-1],
        v[i] ) <= E0 )
      --k;
    ch[k++] = v[i];
  }
  ch.resize( k - 1 );
  return ch;
}
vector<pt> minkowski( vector<pt> P, vector<pt> Q
    ) {
  rotate( P.begin(), min_element( P.begin(),
      P.end() ), P.end() );
  rotate( Q.begin(), min_element( Q.begin(),
      Q.end() ), Q.end() );
  P.push_back(P[0]), P.push_back(P[1]);
  Q.push_back(Q[0]), Q.push_back(Q[1]);
  vector<pt> ans;
  size_t i = 0, j = 0;
  while(i < P.size() - 2 || j < Q.size() - 2) {
      ans.push_back(P[i] + Q[j]);
      lf dt = cross( P[i + 1] - P[i], Q[j + 1] -
          Q[j]);
      if(dt >= E0 && i < P.size() - 2) ++i;
      if(dt <= E0 && j < Q.size() - 2) ++j;
  }
  return ans;
}
vector< pt > cut( const vector< pt > &pol, line l
    ) {
  vector< pt > ans;
  for( int i = 0, n = pol.size(); i < n; ++ i ) {
```

```
    lf s1 = l.eval( pol[i] ), s2 = l.eval(
        pol[(i+1)%n] );
    if( s1 >= -EPS ) ans.push_back( pol[i] );
    if( ( s1 < -EPS && s2 > EPS ) || ( s1 > EPS
        && s2 < -EPS ) ) {
      line li = line( pol[i], pol[(i+1)%n] );
      ans.push_back( lines_intersection( l, li ) );
    }
  }
  return ans;
}
int point_in_polygon( const vector< pt > &pol,
    const pt &p ) {
  int wn = 0;
  for( int i = 0, n = pol.size(); i < n; ++ i ) {
    lf c = orient( p, pol[i], pol[(i+1)%n] );
    if( fabsl( c ) <= E0 && dot( pol[i] - p,
        pol[(i+1)%n] - p ) <= E0 ) return ON;
    if( c > 0 && pol[i].y <= p.y + E0 &&
        pol[(i+1)%n].y - p.y > E0 ) ++wn;
    if( c < 0 && pol[(i+1)%n].y <= p.y + E0 &&
        pol[i].y - p.y > E0 ) --wn;
  }
  return wn ? IN : OUT;
}
int point_in_convex_polygon( const vector < pt >
    &pol, const pt &p ) {
  int low = 1, high = pol.size() - 1;
  while( high - low > 1 ) {
    int mid = ( low + high ) / 2;
    if( orient( pol[0], pol[mid], p ) >= -E0 )
        low = mid;
    else high = mid;
  }
  if( orient( pol[0], pol[low], p ) < -E0 )
      return OUT;
  if( orient( pol[low], pol[high], p ) < -E0 )
      return OUT;
  if( orient( pol[high], pol[0], p ) < -E0 )
      return OUT;
  if( low == 1 && orient( pol[0], pol[low], p )
      <= E0 ) return ON;
  if( orient( pol[low], pol[high], p ) <= E0 )
      return ON;
  if( high == (int) pol.size() -1 && orient(
      pol[high], pol[0], p ) <= E0 ) return ON;
  return IN;
}
```

4.2 Bentley–Ottmann

```
struct seg {
  pt p, q;
  int id;
  lf get_y(double x) const {
    if (fabsl (p.x - q.x) < EPS) return p.y;
    return p.y + (q.y - p.y) * (x - p.x) / (q.x -
        p.x);
  }
};
bool operator<(const seg& a, const seg& b) {
  lf x = max(min(a.p.x, a.q.x), min(b.p.x,
      b.q.x));
  return a.get_y(x) < b.get_y(x) - EPS;
}
struct event {
  double x;
  int tp, id;
  event(double x, int tp, int id) : x(x), tp(tp),
      id(id) {}
  bool operator<(const event& e) const {
    if (abs(x - e.x) > EPS)
      return x < e.x;
    return tp > e.tp;
  }
};
set<seg> s;
vector<set<seg>::iterator> where;
set<seg>::iterator prev(set<seg>::iterator it) {
  return it == s.begin() ? s.end() : --it;
}
set<seg>::iterator next(set<seg>::iterator it) {
  return ++it;
}
pair<int, int> solve(const vector<seg>& a) {
  int n = (int)a.size();
  vector<event> e;
  for (int i = 0; i < n; ++i) {
    e.push_back(event(min(a[i].p.x, a[i].q.x),
        +1, i));
    e.push_back(event(max(a[i].p.x, a[i].q.x),
        -1, i));
  }
  sort(e.begin(), e.end());
  s.clear();
  where.resize(a.size());
  for (size_t i = 0; i < e.size(); ++i) {
    int id = e[i].id;
    if (e[i].tp == +1) {
      set<seg>::iterator nxt =
          s.lower_bound(a[id]), prv = prev(nxt);
      if (nxt != s.end() && intersect(*nxt, a[id]))
        return make_pair(nxt->id, id);
      if (prv != s.end() && intersect(*prv, a[id]))
        return make_pair(prv->id, id);
      where[id] = s.insert(nxt, a[id]);
    } else {
      set<seg>::iterator nxt = next(where[id]),
          prv = prev(where[id]);
      if (nxt != s.end() && prv != s.end() &&
          intersect(*nxt, *prv))
        return make_pair(prv->id, nxt->id);
      s.erase(where[id]);
    }
  }
  return make_pair(-1, -1);
}
```

4.3 Closest Points

```
pair < pt, pt > closest_points ( vector< pt > v )
    {
  sort( v.begin(), v.end() );
  pair< pt, pt > ans;
  lf d2 = INF;
  function< void( int, int ) > solve = [&]( int
      l, int r ) {
    if( l == r ) return;
```

```
    int mid = ( l + r ) / 2;
    lf x_mid = v[mid].x;
    solve( l, mid );
    solve( mid + 1, r );
    vector< pt > aux;
    int p1 = l, p2 = mid + 1;
    while ( p1 <= mid && p2 <= r ) {
      if( v[p1].y < v[p2].y ) aux.push_back(
          v[p1++] );
      else aux.push_back( v[p2++] );
    }
    while( p1 <= mid ) aux.push_back( v[p1++] );
    while( p2 <= r ) aux.push_back( v[p2++] );
    vector < pt > nb;
    for( int i = l; i <= r; ++ i ) {
      v[i] = aux[i-l];
      lf dx = ( x_mid - v[i].x );
      if( dx * dx < d2 )
        nb.push_back( v[i] );
    }
    for( int i = 0; i < (int) nb.size(); ++ i ) {
      for( int k = i + 1; k < (int) nb.size(); ++
          k ) {
        lf dy = ( nb[k].y - nb[i].y );
        if( dy * dy > d2 ) break;
        lf nd2 = dis2( nb[i], nb[k] );
        if( nd2 < d2 ) d2 = nd2, ans = {nb[i],
            nb[k]};
      }
    }
  };
  solve( 0, v.size() -1 );
  return ans;
}
```

4.4 Halfplane Intersection

```
struct Halfplane {
  pt p, pq;
  lf angle;
  Halfplane() {}
  Halfplane(const pt& a, const pt& b) : p(a),
      pq(b - a) {
    angle = atan2l(pq.y, pq.x);
  }
  bool out(const pt& r) {
    return cross(pq, r - p) < -EPS;
  }
  bool operator < (const Halfplane& e) const {
    return angle < e.angle;
  }
  friend pt inter(const Halfplane& s, const
      Halfplane& t) {
    lf alpha = cross((t.p - s.p), t.pq) /
        cross(s.pq, t.pq);
    return s.p + (s.pq * alpha);
  }
};
vector<pt> hp_intersect(vector<Halfplane>& H) {
  pt box[4] = { pt(INF, INF), pt(-INF, INF),
      pt(-INF, -INF), pt(INF, -INF) };
  for(int i = 0; i < 4; ++i ) {
    Halfplane aux(box[i], box[(i+1) % 4]);
    H.push_back(aux);
  }
  sort(H.begin(), H.end());
  deque<Halfplane> dq;
  int len = 0;
  for(int i = 0; i < int(H.size()); ++i ) {
    while (len > 1 && H[i].out(inter(dq[len-1],
        dq[len-2]))) {
      dq.pop_back();
      --len;
    }
    while (len > 1 && H[i].out(inter(dq[0],
        dq[1]))) {
      dq.pop_front();
      --len;
    }
    if (len > 0 && fabsl(cross(H[i].pq,
        dq[len-1].pq)) < EPS ) {
      if (dot(H[i].pq, dq[len-1].pq) < 0.0)
        return vector<pt>();
      if (H[i].out(dq[len-1].p)) {
        dq.pop_back();
        --len;
      }
      else continue;
    }
    dq.push_back(H[i]);
    ++len;
  }
  while (len > 2 && dq[0].out(inter(dq[len-1],
      dq[len-2]))) {
    dq.pop_back();
    --len;
  }
  while (len > 2 && dq[len-1].out(inter(dq[0],
      dq[1]))) {
    dq.pop_front();
    --len;
  }
  if (len < 3) return vector<pt>();
  vector<pt> ret(len);
  for(int i = 0; i+1 < len; ++i )
    ret[i] = inter(dq[i], dq[i+1]);
  ret.back() = inter(dq[len-1], dq[0]);
  //remove repeated points if needed
  return ret;
}
```

4.5 Min Circle

```
Circle min_circle( vector< pt > v ) {
  random_shuffle( v.begin(), v.end() );
  auto f2 = [&]( int a, int b ){
    Circle ans( v[a], v[b] );
    for( int i = 0; i < a; ++ i )
      if( ans.contains( v[i] ) == OUT )
        ans = Circle( v[i], v[a], v[b] );
    return ans;
  };
  auto f1 = [&]( int a ){
    Circle ans( v[a], 0.0L );
    for( int i = 0; i < a; ++ i )
      if( ans.contains( v[i] ) == OUT )
        ans = f2( i, a );
```

```
    return ans;
  };
  Circle ans( v[0], 0.0L );
  for( int i = 1; i < (int) v.size(); ++ i )
    if( ans.contains( v[i] ) == OUT )
      ans = f1( i );
  return ans;
}
```

4.6 Miscellaneous

```
lf part(pt a, pt b, T r) {
  lf l = abs(a-b);
  pt p = (b-a)/l;
  lf c = dot(a, p), d = 4.0 * (c*c - dot(a, a) +
      r*r);
  if (d < eps) return angle(a, b) * r * r * 0.5;
  d = sqrt(d) * 0.5;
  lf s = -c - d, t = -c + d;
  if (s < 0.0) s = 0.0; else if (s > l) s = l;
  if (t < 0.0) t = 0.0; else if (t > l) t = l;
  pt u = a + p*s, v = a + p*t;
  return (cross(u, v) + (angle(a, u) + angle(v,
      b)) * r * r) * 0.5;
}
lf inter_cp(circle c, polygon p) {
  lf ans = 0;
  int n = p.p.size();
  for (int i = 0; i < n; i++) {
    ans += part(p[i]-c.c, p[(i+1)%4]-c.c, c.r);
  }
  return abs(ans);
}
bool circumcircle_contains( triangle tr, pt D )
    {//triange CCW
  pt A = tr.vert[0] - D, B = tr.vert[1] - D, C =
      tr.vert[2] - D;
  lf norm_a = norm2( tr.vert[0] ) - norm2( D );
  lf norm_b = norm2( tr.vert[1] ) - norm2( D );
  lf norm_c = norm2( tr.vert[2] ) - norm2( D );
  lf det1 = A.x * ( B.y * norm_c - norm_b * C.y );
  lf det2 = B.x * ( C.y * norm_a - norm_c * A.y );
  lf det3 = C.x * ( A.y * norm_b - norm_a * B.y );
  return det1 + det2 + det3 > E0;
}
lf areaOfIntersectionOfTwoCircles( lf r1, lf r2,
    lf d ) {
  if( d >= r1 + r2 )
    return 0.0L;
  if( d <= fabsl( r2 - r1 ) )
    return PI * ( r1 < r2 ? r1 * r1 : r2 * r2 );
  lf alpha = safeAcos( ( r1 * r1 - r2 * r2 + d *
      d ) / ( 2.0L * d * r1 ) );
  lf betha = safeAcos( ( r2 * r2 - r1 * r1 + d *
      d ) / ( 2.0L * d * r2 ) );
  lf a1 = r1 * r1 * ( alpha - sinl( alpha ) *
      cosl( alpha ) );
  lf a2 = r2 * r2 * ( betha - sinl( betha ) *
      cosl( betha ) );
  return a1 + a2;
};
bool half(pt p) { //true if is in (0, 180]
  assert(p.x != 0 || p.y != 0); //the argument of
      (0,0) is undefined
  return p.y > 0 || (p.y == 0 && p.x < 0);
}
bool half_from(pt p, pt v = {1, 0}) {
  return cross(v,p) < 0 || (cross(v,p) == 0 &&
      dot(v,p) < 0);
}
bool polar_cmp(const pt &a, const pt &b) {
  return make_tuple(half(a), 0) <
      make_tuple(half(b), cross(a,b));
}
bool in_disk(pt a, pt b, pt p) {
  return dot(a-p, b-p) <= 0;
}
bool on_segment(pt a, pt b, pt p) {
  return orient(a,b,p) == 0 && in_disk(a,b,p);
}
bool proper_inter(pt a, pt b, pt c, pt d, pt
    &out) {
  T oa = orient(c,d,a), ob = orient(c,d,b),
    oc = orient(a,b,c), od = orient(a,b,d);
  if (oa*ob < 0 && oc*od < 0) {
    out = (a*ob - b*oa) / (ob-oa);
    return true;
  }
  return false;
}
set<pt> inter_ss(pt a, pt b, pt c, pt d) {
  pt out;
  if (proper_inter(a,b,c,d,out)) return {out};
  set<pt> s;
  if (on_segment(c,d,a)) s.insert(a);
  if (on_segment(c,d,b)) s.insert(b);
  if (on_segment(a,b,c)) s.insert(c);
  if (on_segment(a,b,d)) s.insert(d);
  return s;
}
lf pt_to_seg(pt a, pt b, pt p) {
  if(a != b) {
    line l(a,b);
    if (l.cmp_proj(a,p) && l.cmp_proj(p,b)) //if
        closest to projection
      return l.dist(p); //output distance to line
  }
  return min(abs(p-a), abs(p-b)); //otherwise
      distance to A or B
}
lf seg_to_seg(pt a, pt b, pt c, pt d) {
  pt dummy;
  if (proper_inter(a,b,c,d,dummy)) return 0;
  return min({pt_to_seg(a,b,c), pt_to_seg(a,b,d),
          pt_to_seg(c,d,a), pt_to_seg(c,d,b)});
}
enum {IN, OUT, ON};
struct polygon {
  vector<pt> p;
  polygon(int n) : p(n) {}
  pt centroid() {
    pt c{0, 0};
    lf scale = 6. * area(true);
    for(int i = 0, n = p.size(); i < n; ++i) {
      int j = (i+1 == n ? 0 : i+1);
      c = c + (p[i] + p[j]) * cross(p[i], p[j]);
    }
    return c / scale;
  }
  ll pick() {
```

```
    ll boundary = 0;
    for(int i = 0, n = p.size(); i < n; i++) {
      int j = (i+1 == n ? 0 : i+1);
      boundary += __gcd((ll)abs(p[i].x - p[j].x),
          (ll)abs(p[i].y - p[j].y));
    }
    return area() + 1 - boundary/2;
  }
  pt& operator[] (int i){ return p[i]; }
};
int tangents(circle c1, circle c2, bool inner,
    vector<pair<pt,pt>> &out) {
  if(inner) c2.r = -c2.r;
  pt d = c2.c-c1.c;
  double dr = c1.r-c2.r, d2 = norm(d), h2 =
      d2-dr*dr;
  if(d2 == 0 || h2 < 0) { assert(h2 != 0); return
      0; }
  for(double s : {-1,1}) {
    pt v = (d*dr + rot90ccw(d)*sqrt(h2)*s)/d2;
    out.push_back({c1.c + v*c1.r, c2.c + v*c2.r});
  }
  return 1 + (h2 > 0);
}
int tangent_through_pt(pt p, circle c, pair<pt,
    pt> &out) {
  double d = abs(p - c.c);
        if(d < c.r) return 0;
  pt base = c.c-p;
  double w = sqrt(norm(base) - c.r*c.r);
  pt a = {w, c.r}, b = {w, -c.r};
  pt s = p + base*a/norm(base)*w;
  pt t = p + base*b/norm(base)*w;
  out = {s, t};
  return 1 + (abs(c.c-p) == c.r);
}
//cross product 3D (VxW)
{ v.y*w.z - v.z*w.y, v.z*w.x - v.x*w.z, v.x*w.y -
    v.y*w.x };
```

5 Graphs

5.1 2-satisfiability

```
//Complexity: O(|N|)
struct sat2 {
  int n;
  vector<vector<vector<int>>> g;
  vector<int> tag;
  vector<bool> seen, value;
  stack<int> st;
  sat2(int n) : n(n), g(2,
      vector<vector<int>>(2*n)), tag(2*n),
      seen(2*n), value(2*n) { }
  int neg(int x) { return 2*n-x-1; }
  void add_or(int u, int v) { implication(neg(u),
      v); }
  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) {
    g[0][u].push_back(v);
    g[1][v].push_back(u);
  }
  void dfs(int id, int u, int t = 0) {
    seen[u] = true;
    for(auto& v : g[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;
  }
};
```

5.2 Erdos–Gallai theorem

```
//Complexity: O(|N|*log|N|)
//Theorem: it gives a necessary and sufficient
    condition for a finite sequence
//        of natural numbers to be the degree
    sequence of a simple graph
bool erdos(vector<int> &d) {
  ll sum = 0;
  for(int i = 0; i < d.size(); ++i) sum += d[i];
  if(sum & 1) return false;
  sort(d.rbegin(), d.rend());
  ll l = 0, r = 0;
  for(int k = 1, i = d.size() - 1; k <= d.size();
      ++k) {
    l += d[k-1];
    if(k > i) r -= d[++i];
    while (i >= k && d[i] < k+1) r += d[i--];
    if(l > 1ll*k*(k-1) + 1ll*k*(i-k+1) + r)
      return false;
  }
  return true;
}
```

5.3 Eulerian path

```
//Complexity: O(|N|)
struct edge {
  int v; //list<edge>::iterator rev;
  edge(int v) : v(v) {}
};
void add_edge(int a, int b) {
  g[a].push_front(edge(b)); //auto ia =
      g[a].begin();
       g[b].push_front(edge(a)); //auto ib =
           g[b].begin();
       //ia->rev=ib; ib->rev=ia;
}
//for undirected uncomment and check for path
    existance
bool eulerian(vector<int> &tour) { //directed
    graph
  int one_in = 0, one_out = 0, start = -1;
  bool ok = true;
  for (int i = 0; i < n; i++) {
    if(out[i] && start == -1) start = i;
    if(out[i] - in[i] == 1) one_out++, start = i;
    else if(in[i] - out[i] == 1) one_in++;
    else ok &= in[i] == out[i];
  }
  ok &= one_in == one_out && one_in <= 1;
  if (ok) {
    function<void(int)> go = [&](int u) {
      while(g[u].size()) {
        int v = g[u].front().v;
        g[v].erase(g[u].front().rev);
        g[u].pop_front();
        go(v, tour);
      }
      tour.push_back(u);
    };
    go(start);
    reverse(tour.begin(), tour.end());
    if(tour.size() == edges + 1) return true;
  }
  return false;
}
```

5.4 Number of spanning trees

```
//A -> adjacency matrix
//It is necessary to compute the D-A matrix,
    where D is a diagonal matrix
//that contains the degree of each node.
//To compute the number of spanning trees it's
    necessary to compute any
//D-A cofactor
//C(i, j) = (-1)^(i+j) * Mij
//Where Mij is the matrix determinant after
    removing row i and column j
double mat[MAX][MAX];
//call determinant(n - 1)
double determinant(int n) {
  double det = 1.0;
  for(int k = 0; k < n; k++) {
    for(int i = k+1; i < n; i++) {
      assert(mat[k][k] != 0);
      long double factor = mat[i][k]/mat[k][k];
      for(int j = 0; j < n; j++) {
        mat[i][j] = mat[i][j] - factor*mat[k][j];
      }
    }
    det *= mat[k][k];
  }
  return round(det);
}
```

5.5 Scc

```
//Complexity: O(|N|)
int scc(int n) {
  vector<int> dfn(n+1), low(n+1), in_stack(n+1);
  stack<int> st;
  int tag = 0;
  function<void(int, int&)> dfs = [&](int u, int
      &t) {
    dfn[u] = low[u] = ++t;
    st.push(u);
    in_stack[u] = true;
    for(auto &v : g[u]) {
      if(!dfn[v]) {
        dfs(v, t);
        low[u] = min(low[u], low[v]);
      } else if(in_stack[v])
        low[u] = min(low[u], dfn[v]);
    }
    if (low[u] == dfn[u]) {
      int v;
      do {
        v = st.top(); st.pop();
//       id[v] = tag;
        in_stack[v] = false;
      } while (v != u);
      tag++;
    }
  };
  for(int u = 1, t; u <= n; ++u) {
    if(!dfn[u]) dfs(u, t = 0);
  }
  return tag;
}
```

5.6 Tarjan tree

```
//Complexity: O(|N|)
struct tarjan_tree {
  int n;
  vector<vector<int>> g, comps;
  vector<pii> bridge;
  vector<int> id, art;
  tarjan_tree(int n) : n(n), g(n+1), id(n+1),
      art(n+1) {}
  void add_edge(vector<vector<int>> &g, int u,
      int v) { //nodes from [1, n]
    g[u].push_back(v);
    g[v].push_back(u);
```

```
}
void add_edge(int u, int v) { add_edge(g, u,
    v); }
void tarjan(bool with_bridge) {
  vector<int> dfn(n+1), low(n+1);
  stack<int> st;
  comps.clear();
  function<void(int, int, int&)> dfs = [&](int
      u, int p, int &t) {
    dfn[u] = low[u] = ++t;
    st.push(u);
    int cntp = 0;
    for(int v : g[u]) {
      cntp += v == p;
      if(!dfn[v])   {
        dfs(v, u, t);
        low[u] = min(low[u], low[v]);
        if(with_bridge && low[v] > dfn[u]) {
          bridge.push_back({min(u,v), max(u,v)});
          comps.push_back({});
          for(int w = -1; w != v; )
            comps.back().push_back(w =
                st.top()), st.pop();
        }
        if(!with_bridge && low[v] >= dfn[u]) {
          art[u] = (dfn[u] > 1 || dfn[v] > 2);
          comps.push_back({u});
          for(int w = -1; w != v; )
            comps.back().push_back(w =
                st.top()), st.pop();
        }
      }
      else if(v != p || cntp > 1) low[u] =
          min(low[u], dfn[v]);
    }
    if(p == -1 && ( with_bridge || g[u].size()
        == 0 )) {
      comps.push_back({});
      for(int w = -1; w != u; )
        comps.back().push_back(w = st.top()),
            st.pop();
    }
  };
  for(int u = 1, t; u <= n; ++u)
    if(!dfn[u]) dfs(u, -1, t = 0);
}
vector<vector<int>> build_block_cut_tree() {
  tarjan(false);
  int t = 0;
  for(int u = 1; u <= n; ++u)
    if(art[u]) id[u] = t++;
  vector<vector<int>> tree(t+comps.size());
  for(int i = 0; i < comps.size(); ++i)
    for(int u : comps[i]) {
      if(!art[u]) id[u] = i+t;
      else add_edge(tree, i+t, id[u]);
    }
  return tree;
}
vector<vector<int>> build_bridge_tree() {
  tarjan(true);
  vector<vector<int>> tree(comps.size());
  for(int i = 0; i < comps.size(); ++i)
    for(int u : comps[i]) id[u] = i;
  for(auto &b : bridge)
    add_edge(tree, id[b.first], id[b.second]);
  return tree;
}
};
```

6 Math

6.1 Berlekamp-Massey

```
namespace linear_seq {
  int m; //a = first m terms
  vector < int > p, a;//p = recurrence, length is
      m
  inline vector < int > BM(vector < int > x) {
      //finds shortest linear recurrence given
      first x terms in O(x^2)
    vector < int > ls, cur;
    int lf, ld;
    for (int i = 0; i < (int) x.size(); ++i) {
      int t = 0;
      for (int j = 0; j < (int) cur.size(); ++j)
        t = (t + 1 LL * x[i - j - 1] * cur[j]) %
            mod;
      if ((t - x[i]) % mod == 0) continue;
      if (!cur.size()) {
        cur.resize(i + 1);
        lf = i;
        ld = (t - x[i]) % mod;
        continue;
      }
      int k = 1 LL * (t - x[i]) * pw(ld, mod - 2)
          % mod;
      vector < int > c(i - lf - 1);
      c.push_back(k);
      for (int j = 0; j < (int) ls.size(); ++j)
        c.push_back(-1 LL * ls[j] * k % mod);
      if (c.size() < cur.size())
          c.resize(cur.size());
      for (int j = 0; j < (int) cur.size(); ++j)
        c[j] = (c[j] + cur[j]) % mod;
      if (i + lf + (int) ls.size() >= (int)
          cur.size())
        ls = cur, lf = i, ld = (t - x[i]) % mod;
      cur = c;
    }
    for (int i = 0; i < (int) cur.size(); ++i)
      cur[i] = (cur[i] % mod + mod) % mod;
    m = cur.size();
    p.resize(m), a.resize(m);
    for (int i = 0; i < m; ++i)
      p[i] = cur[i], a[i] = x[i];
    return cur;
  }
  inline vector < int > mul(vector < int > & a,
      vector < int > & b) {
    vector < int > r(2 * m);
    for (int i = 0; i < m; ++i)
      if (a[i])
        for (int j = 0; j < m; ++j)
          r[i + j] = (r[i + j] + 1 LL * a[i] *
              b[j]) % mod;
    for (int i = 2 * m - 1; i >= m; --i)
      if (r[i])
        for (int j = m - 1; j >= 0; --j)
```

```
      r[i - j - 1] = (r[i - j - 1] + 1 LL *
          p[j] * r[i]) % mod;
    r.resize(m);
    return r;
  }
  inline int calc(long long k) { //O(m*m*log(k))
    if (m == 0) return 0;
    vector < int > bs(m), r(m);
    if (m == 1) bs[0] = p[0];
    else bs[1] = 1;
    r[0] = 1;
    while (k) {
      if (k & 1) r = mul(r, bs);
      bs = mul(bs, bs);
      k >>= 1;
    }
    int res = 0;
    for (int i = 0; i < m; ++i)
      res = (res + 1 LL * r[i] * a[i]) % mod;
    return res;
  }
}
```

6.2 Chinese remainder theorem

```
//Complexity: |N|*log(|N|)
//finds a suitable x that meets: x is congruent
    to a_i mod n_i
/** Works for non-coprime moduli.
 Returns {-1,-1} if solution does not exist or
     input is invalid.
 Otherwise, returns {x,L}, where x is the
     solution unique to mod L = LCM of mods*/
pair<int, int> chinese_remainder_theorem(
    vector<int> A, vector<int> M ) {
  int n = A.size(), a1 = A[0], m1 = M[0];
  for(int i = 1; i < n; i++) {
    int a2 = A[i], m2 = M[i];
    int g = __gcd(m1, m2);
    if( a1 % g != a2 % g ) return {-1,-1};
    int p, q;
    eea(m1/g, m2/g, &p, &q);
    int mod = m1 / g * m2;
    q %= mod; p %= mod;
    int x = ((1ll*(a1%mod)*(m2/g))%mod*q +
        (1ll*(a2%mod)*(m1/g))%mod*p) % mod; //if
        WA there is overflow
    a1 = x;
    if (a1 < 0) a1 += mod;
    m1 = mod;
  }
  return {a1, m1};
}
```

6.3 Constant modular inverse

```
//Complexity: O(|P|)
//Find the multiplicative inverse of all 2<=i<p,
    module p
inv[1] = 1;
for(int i = 2; i < p; ++i)
        inv[i] = (p - (p / i) * inv[p % i] % p) %
            p;
```

6.4 Extended euclides

```
//Complexity: O(log(|N|))
ll eea(ll a, ll b, ll& x, ll& y) {
  ll xx = y = 0; ll yy = x = 1;
  while (b) {
    ll q = a / b; ll 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;
}
ll inverse(ll a, ll n) {
  ll x, y;
  ll g = eea(a, n, x, y);
  if(g > 1)
    return -1;
  return (x % n + n) % n;
}
```

6.5 FWHT

```
vector< long long > haddamard( vector< long long
    > a, bool inverse ) {
  const int n = (int) a.size();
  for( int k = 1; k < n; k <<= 1 ) {
    for( int i = 0; i < n; i += 2 * k ) {
      for( int j = 0; j < k; ++ j ) {
        long long u = a[i+j], v = a[i+j+k];
        a[i+j] = u + v;
        a[i+j+k] = u - v;
      }
    }
  }
  if( inverse )
    for( auto &x : a )
      x >>= 1;
  }
  return a;
} //XOR convolution, |A| = |B| = power of two
vector< long long > FWHT( vector< long long > a,
    vector< long long > b ) {
  auto h_a = haddamard( a, false ), h_b =
      haddamard( b, false );
  vector< long long > h_c( a.size() );
  for( int i = 0; i < (int) a.size(); ++ i )
    h_c[i] = 1LL * h_a[i] * h_b[i];
  return haddamard( h_c, true );
}
```

6.6 Fast Fourier transform module

```
struct FFT {
  int mod, root, root_1, root_pw;
  void fft(vector<int> & a, bool invert) {
      int n = a.size();
      for (int i = 1, j = 0; i < n; ++i) {
          int bit = n >> 1;
          for (; j & bit; bit >>= 1)
```

```
            j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        int wlen = invert ? root_1 : root;
        for (int i = len; i < root_pw; i <<= 1)
            wlen = 1LL * wlen * wlen % mod;
        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; ++j) {
                int u = a[i+j], v = 1LL *
                    a[i+j+len/2] * w % mod;
                a[i+j] = u + v < mod ? u + v : u
                    + v - mod;
                a[i+j+len/2] = u - v >= 0 ? u - v
                    : u - v + mod;
                w = 1LL * w * wlen % mod;
            }
        }
    }
    if (invert) {
        int n_1 = pw(n, mod-2, mod);
        for (int & x : a)
            x = 1LL * x * n_1 % mod;
    }
  }
  vector<int> multiply(vector<int> const& a,
      vector<int> const& b) {
    vector<int> fa(a.begin(), a.end()),
        fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] = 1LL * fa[i] * fb[i] % mod;
    fft(fa, true);
    return fa;
  }
};
```

```
FFT A = { 998244353, 15311432, 469870224, 1<<23 };
```

6.7 Fast fourier transform

```
///Complexity: O(N log N)
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define sz(v) ((int)v.size())
#define trav(a, x) for(auto& a : x)
#define all(v) v.begin(),v.end()
typedef vector<ll> vl;
typedef vector<int> vi;
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
        int n = sz(a), L = 31 - __builtin_clz(n);
        static vector<complex<long double>> R(2,
            1);
        static vector<C> rt(2, 1); //(^ 10% faster
            if double)
        for (static int k = 2; k < n; k *= 2) {
                R.resize(n); rt.resize(n);
                auto x = polar(1.0L, acos(-1.0L) /
                    k);
                rep(i,k,2*k) rt[i] = R[i] = i&1 ?
                    R[i/2] * x : R[i/2];
        }
        vi rev(n);
        rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1)
            << L) / 2;
        rep(i,0,n) if (i < rev[i]) swap(a[i],
            a[rev[i]]);
        for (int k = 1; k < n; k *= 2)
  for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
    //C z = rt[j+k] * a[i+j+k]; //(25% faster if
        hand-rolled) //include-line
    auto x = (double *)&rt[j+k], y = (double
        *)&a[i+j+k]; //exclude-line
    C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] +
        x[1]*y[0]);  //exclude-line
    a[i + j + k] = a[i + j] - z;
    a[i + j] += z;
  }
}
vl conv(const vl& a, const vl& b) {
        if (a.empty() || b.empty()) return {};
        vd res(sz(a) + sz(b) - 1);
        int L = 32 - __builtin_clz(sz(res)), n = 1
            << L;
        vector<C> in(n), out(n);
        copy(all(a), begin(in));
        rep(i,0,sz(b)) in[i].imag(b[i]);
        fft(in);
        trav(x, in) x *= x;
        rep(i,0,n) out[i] = in[-i & (n - 1)] -
            conj(in[i]);
        fft(out);
    vector<ll> resp(sz(res));
        rep(i,0,sz(res)) resp[i] =
            round(imag(out[i]) / (4.0 * n));
        return resp;
}
```

6.8 Gauss jordan

```
//Complexity: O(|N|^3)
const int EPS = 1;
int gauss (vector<vector<int>> a, vector<int>
    &ans) {
  int n = a.size(), m = a[0].size()-1;
  vector<int> where(m, -1);
  for(int col = 0, row = 0; col < m && row < n;
      ++col) {
    int sel = row;
    for(int i = row; i < n; ++i)
      if(abs(a[i][col]) > abs(a[sel][col])) sel =
          i;
    if(abs(a[sel][col]) < EPS) continue;
    swap(a[sel], a[row]);
    where[col] = row;
    for(int i = 0; i < n; ++i)
      if(i != row) {
        int c = divide(a[i][col], a[row][col]);
            //precalc inverses
        for(int j = col; j <= m; ++j)
```

```
        a[i][j] = sub(a[i][j], mul(a[row][j],
            c));
      }
    ++row;
  }
  ans.assign(m, 0);
  for(int i = 0; i < m; ++i)
    if(where[i] != -1) ans[i] =
        divide(a[where[i]][m], a[where[i]][i]);
  for(int i = 0; i < n; ++i) {
    int sum = 0;
    for(int j = 0; j < m; ++j)
      sum = add(sum, mul(ans[j], a[i][j]));
    if(sum != a[i][m]) return 0;
  }
  for(int i = 0; i < m; ++i)
    if(where[i] == -1) return -1; //infinite
        solutions
  return 1;
}
```

6.9 Lagrange Interpolation

```
//Complexity: O(|N|^2)
vector<lf> X, F;
lf f(lf x) {
  lf answer = 0;
  for(int i = 0; i < (int)X.size(); i++) {
    lf prod = F[i];
    for(int j = 0; j < (int)X.size(); j++) {
      if(i == j) continue;
      prod = mul(prod, divide(sub(x, X[j]),
          sub(X[i], X[j])));
    }
    answer = add(answer, prod);
  }
  return answer;
}
//given y=f(x) for x [0,degree]
vector< int > interpolation( vector< int > &y ) {
  int n = (int) y.size();
  vector< int > u = y, ans( n ), sum( n );
  ans[0] = u[0], sum[0] = 1;
  for( int i = 1; i < n; ++i ) {
    int inv = modpow( i, mod - 2 );
    for( int j = n - 1; j >= i; --j )
      u[j] = 1LL * (u[j] - u[j - 1] + mod) * inv %
          mod;
    for( int j = i; j > 0; --j ) {
      sum[j] = (sum[j - 1] - 1LL * (i - 1) *
          sum[j] % mod + mod) % mod;
      ans[j] = (ans[j] + 1LL * sum[j] * u[i]) %
          mod;
    }
    sum[0] = 1LL * (i - 1) * (mod - sum[0]) % mod;
    ans[0] = (ans[0] + 1LL * sum[0] * u[i]) % mod;
  }
  return ans;
}
```

6.10 Linear diophantine

```
//Complexity: O(log(|N|))
bool diophantine(ll a, ll b, ll c, ll &x, ll &y,
    ll &g) {
  x = y = 0;
  if(a == 0 && b == 0) return c == 0;
  if(b == 0) swap(a, b), swap(x, y);
  g = eea(abs(a), abs(b), x, y);
  if(c % g) return false;
  a /= g; b /= g; c /= g;
  if(a < 0) x *= -1;
  x = (x % b) * (c % b) % b;
  if(x < 0) x += b;
  y = (c - a*x) / b;
  return true;
}
//finds the first k | x + b * k / gcd(a, b) >= val
ll greater_or_equal_than(ll a, ll b, ll x, ll
    val, ll g) {
  lf got = 1.0 * (val - x) * g / b;
  return b > 0 ? ceil(got) : floor(got);
}
void get_xy (ll a, ll b, ll &x, ll &y, ll k, ll
    g) { //if for y, change the order to b,a y,x
  x = x + b / g * k;
  y = y - a / g * k;
}
```

6.11 Matrix multiplication

```
struct matrix {
  const int N = 2;
  int m[N][N], r, c;
  matrix(int r = N, int c = N, bool iden = false)
      : r(r), c(c) {
    memset(m, 0, sizeof m);
    if(iden)
      for(int i = 0; i < r; i++) m[i][i] = 1;
  }
  matrix operator * (const matrix &o) const {
    matrix ret(r, o.c);
    for(int i = 0; i < r; ++i)
      for(int j = 0; j < o.c; ++j) {
        ll &r = ret.m[i][j];
        for(int k = 0; k < c; ++k)
          r = (r + 1ll*m[i][k]*o.m[k][j]) % MOD;
      }
    return ret;
  }
};
```

6.12 Miller rabin

```
ll mul (ll a, ll b, ll mod) {
  ll ret = 0;
  for(a %= mod, b %= mod; b != 0;
    b >>= 1, a <<= 1, a = a >= mod ? a - mod : a)
        {
    if (b & 1) {
      ret += a;
      if (ret >= mod) ret -= mod;
    }
```

```
  }
  return ret;
}
ll fpow (ll a, ll b, ll mod) {
  ll ans = 1;
  for (; b; b >>= 1, a = mul(a, a, mod))
    if (b & 1)
      ans = mul(ans, a, mod);
  return ans;
}
bool witness (ll a, ll s, ll d, ll n) {
  ll x = fpow(a, d, n);
  if (x == 1 || x == n - 1) return false;
  for (int i = 0; i < s - 1; i++) {
    x = mul(x, x, n);
    if (x == 1) return true;
    if (x == n - 1) return false;
  }
  return true;
}
ll test[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 0};
bool is_prime (ll n) {
  if (n < 2) return false;
  if (n == 2) return true;
  if (n % 2 == 0) return false;
  ll d = n - 1, s = 0;
  while (d % 2 == 0) ++s, d /= 2;
  for (int i = 0; test[i] && test[i] < n; ++i)
    if (witness(test[i], s, d, n))
      return false;
  return true;
}
```

6.13 Pollard's rho

```
ll pollard_rho(ll n, ll c) {
  ll x = 2, y = 2, i = 1, k = 2, d;
  while (true) {
    x = (mul(x, x, n) + c);
    if (x >= n) x -= n;
    d = __gcd(x - y, n);
    if (d > 1) return d;
    if (++i == k) y = x, k <<= 1;
  }
  return n;
}
void factorize(ll n, vector<ll> &f) {
  if (n == 1) return;
  if (is_prime(n)) {
    f.push_back(n);
    return;
  }
  ll d = n;
  for (int i = 2; d == n; i++)
    d = pollard_rho(n, i);
  factorize(d, f);
  factorize(n/d, f);
}
```

6.14 Simplex

```
//Complexity: O(|N|^2 * |M|) N variables, N
    restrictions
const double EPS = 1e-6;
typedef vector<double> vec;
namespace simplex {
  vector<int> X, Y;
  vector<vec> a;
  vec b, c;
  double z;
  int n, m;
  void pivot(int x, int y) {
    swap(X[y], Y[x]);
    b[x] /= a[x][y];
    for(int i = 0; i < m; i++)
      if(i != y)
        a[x][i] /= a[x][y];
    a[x][y] = 1 / a[x][y];
    for(int i = 0; i < n; i++)
      if(i != x && abs(a[i][y]) > EPS) {
      b[i] -= a[i][y] * b[x];
      for(int j = 0; j < m; j++)
        if(j != y)
          a[i][j] -= a[i][y] * a[x][j];
        a[i][y] =- a[i][y] * a[x][y];
      }
      z += c[y] * b[x];
      for(int i = 0; i < m; i++)
        if(i != y)
          c[i] -= c[y] * a[x][i];
      c[y] =- c[y] * a[x][y];
  }
  //A is a vector of 1 and 0. B is the limit
      restriction. C is the factors of O.F.
  pair<double, vec> simplex(vector<vec> &A, vec
      &B, vec &C) {
    a = A; b = B; c = C;
    n = b.size(); m = c.size(); z = 0.0;
    X = vector<int>(m);
    Y = vector<int>(n);
    for(int i = 0; i < m; i++) X[i] = i;
    for(int i = 0; i < n; i++) Y[i] = i + m;
    while(1) {
      int x = -1, y = -1;
      double mn = -EPS;
      for(int i = 0; i < n; i++)
        if(b[i] < mn)
          mn = b[i], x = i;
      if(x < 0) break;
      for(int i = 0; i < m; i++)
        if(a[x][i] < -EPS) { y = i; break; }
      assert(y >= 0); //no sol
      pivot(x, y);
    }
    while(1) {
      double mx = EPS;
      int x = -1,y = -1;
      for(int i = 0; i < m; i++)
        if(c[i] > mx)
          mx = c[i], y = i;
      if(y < 0) break;
      double mn = 1e200;
      for(int i = 0; i < n; i++)
        if(a[i][y] > EPS && b[i] / a[i][y] < mn)
          mn = b[i] / a[i][y], x = i;
      assert(x >= 0); //unbound
      pivot(x, y);
    }
```

```
    vec r(m);
    for(int i = 0; i < n; i++)
      if(Y[i] < m)
        r[ Y[i] ] = b[i];
    return make_pair(z, r);
  }
}
```

6.15 Simpson

```
inline lf simpson(lf fl, lf fr, lf fmid, lf l, lf
    r) {
  return (fl + fr + 4.0 * fmid) * (r - l) / 6.0;
}
lf rsimpson (lf slr, lf fl, lf fr, lf fmid, lf l,
    lf r) {
  lf mid = (l + r) * 0.5;
  lf fml = f((l + mid) * 0.5);
  lf fmr = f((mid + r) * 0.5);
  lf slm = simpson(fl, fmid, fml, l, mid);
  lf 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);
}
lf integrate(lf l,lf r) {
        lf mid = (l + r) * .5, fl = f(l), fr =
            f(r), fmid = f(mid);
        return rsimpson(simpson(fl, fr, fmid, l,
            r), fl, fr, fmid, l, r);
}
```

7 Network flows

7.1 Blossom

```
//Complexity: O(|E||V|^2)
struct network {
  struct struct_edge { int v; struct_edge * n; };
  typedef struct_edge* edge;
  int n;
  struct_edge pool[MAXE]; //2*n*n;
  edge top;
  vector<edge> adj;
  queue<int> q;
  vector<int> f, base, inq, inb, inp, match;
  vector<vector<int>> ed;
  network(int n) : n(n), match(n, -1), adj(n),
      top(pool), f(n), base(n),
                  inq(n), inb(n), inp(n), ed(n,
                      vector<int>(n)) {}
  void add_edge(int u, int v) {
    if(ed[u][v]) return;
    ed[u][v] = 1;
    top->v = v, top->n = adj[u], adj[u] = top++;
    top->v = u, top->n = adj[v], adj[v] = top++;
  }
  int get_lca(int root, int u, int v) {
    fill(inp.begin(), inp.end(), 0);
    while(1) {
      inp[u = base[u]] = 1;
      if(u == root) break;
      u = f[ match[u] ];
    }
    while(1) {
      if(inp[v = base[v]]) return v;
      else v = f[ match[v] ];
    }
  }
  void mark(int lca, int u) {
    while(base[u] != lca) {
      int v = match[u];
      inb[ base[u ]] = 1;
      inb[ base[v] ] = 1;
      u = f[v];
      if(base[u] != lca) f[u] = v;
    }
  }
  void blossom_contraction(int s, int u, int v) {
    int lca = get_lca(s, u, v);
    fill(inb.begin(), inb.end(), 0);
    mark(lca, u); mark(lca, v);
    if(base[u] != lca) f[u] = v;
    if(base[v] != lca) f[v] = u;
    for(int u = 0; u < n; u++)
      if(inb[base[u]]) {
        base[u] = lca;
        if(!inq[u]) {
          inq[u] = 1;
          q.push(u);
        }
      }
  }
  int bfs(int s) {
    fill(inq.begin(), inq.end(), 0);
    fill(f.begin(), f.end(), -1);
    for(int i = 0; i < n; i++) base[i] = i;
    q = queue<int>();
    q.push(s);
    inq[s] = 1;
   while(q.size()) {
      int u = q.front(); q.pop();
      for(edge e = adj[u]; e; e = e->n) {
        int v = e->v;
        if(base[u] != base[v] && match[u] != v) {
          if((v == s) || (match[v] != -1 &&
              f[match[v]] != -1))
            blossom_contraction(s, u, v);
          else if(f[v] == -1) {
            f[v] = u;
            if(match[v] == -1) return v;
            else if(!inq[match[v]]) {
              inq[match[v]] = 1;
              q.push(match[v]);
            }
          }
        }
      }
    }
    return -1;
  }
  int doit(int u) {
    if(u == -1) return 0;
    int v = f[u];
    doit(match[v]);
    match[v] = u; match[u] = v;
```

```
    return u != -1;
  }
  //(i < net.match[i]) => means match
  int maximum_matching() {
    int ans = 0;
    for(int u = 0; u < n; u++)
      ans += (match[u] == -1) && doit(bfs(u));
    return ans;
  }
};
```

7.2 Dinic

```
//Complexity: O(|E|*|V|^2)
struct edge { int v, cap, inv, flow; };
struct network {
  int n, s, t;
  vector<int> lvl;
  vector<vector<edge>> g;
  network(int n) : n(n), lvl(n), g(n) {}
  void add_edge(int u, int v, int c) {
    g[u].push_back({v, c, g[v].size(), 0});
    g[v].push_back({u, 0, g[u].size()-1, c});
  }
  bool bfs() {
    fill(lvl.begin(), lvl.end(), -1);
    queue<int> q;
    lvl[s] = 0;
    for(q.push(s); q.size(); q.pop()) {
      int u = q.front();
      for(auto &e : g[u]) {
        if(e.cap > 0 && lvl[e.v] == -1) {
          lvl[e.v] = lvl[u]+1;
          q.push(e.v);
        }
      }
    }
    return lvl[t] != -1;
  }
  int dfs(int u, int nf) {
    if(u == t) return nf;
    int res = 0;
    for(auto &e : g[u]) {
      if(e.cap > 0 && lvl[e.v] == lvl[u]+1) {
        int tf = dfs(e.v, min(nf, e.cap));
        res += tf; nf -= tf; e.cap -= tf;
        g[e.v][e.inv].cap += tf;
        g[e.v][e.inv].flow -= tf;
        e.flow += tf;
        if(nf == 0) return res;
      }
    }
    if(!res) lvl[u] = -1;
    return res;
  }
  int max_flow(int so, int si, int res = 0) {
    s = so; t = si;
    while(bfs()) res += dfs(s, INT_MAX);
    return res;
  }
};
```

7.3 Hopcroft karp

```
//Complexity: O(|E|*sqrt(|V|))
struct mbm {
  vector<vector<int>> g;
  vector<int> d, match;
  int nil, l, r;
  //u -> 0 to l, v -> 0 to r
  mbm(int l, int r) : l(l), r(r), nil(l+r),
      g(l+r),
                    d(1+l+r, INF), match(l+r,
                        l+r) {}
  void add_edge(int a, int b) {
    g[a].push_back(l+b);
    g[l+b].push_back(a);
  }
  bool bfs() {
    queue<int> q;
    for(int u = 0; u < l; u++) {
      if(match[u] == nil) {
        d[u] = 0;
        q.push(u);
      } else d[u] = INF;
    }
    d[nil] = INF;
    while(q.size()) {
      int u = q.front(); q.pop();
      if(u == nil) continue;
      for(auto v : g[u]) {
        if(d[ match[v] ] == INF) {
          d[ match[v] ] = d[u]+1;
          q.push(match[v]);
        }
      }
    }
    return d[nil] != INF;
  }
  bool dfs(int u) {
    if(u == nil) return true;
    for(int v : g[u]) {
      if(d[ match[v] ] == d[u]+1 && dfs(match[v]))
          {
        match[v] = u; match[u] = v;
        return true;
      }
    }
    d[u] = INF;
    return false;
  }
  int max_matching() {
    int ans = 0;
    while(bfs()) {
      for(int u = 0; u < l; u++) {
        ans += (match[u] == nil && dfs(u));
      }
    }
    return ans;
  }
};
```

7.4 Maximum bipartite matching

```
//Complexity: O(|E|*|V|)
struct mbm {
```

```
  int l, r;
  vector<vector<int>> g;
  vector<int> match, seen;
  mbm(int l, int r) : l(l), r(r), seen(r),
      match(r), g(l) {}
  void add_edge(int l, int r) {
      g[l].push_back(r); }
  bool dfs(int u) {
    for(auto v : g[u]) {
      if(seen[v]++) continue;
      if(match[v] == -1 || dfs(match[v])) {
        match[v] = u;
        return true;
      }
    }
    return false;
  }
  int max_matching() {
    int ans = 0;
    fill(match.begin(), match.end(), -1);
    for(int u = 0; u < l; ++u) {
      fill(seen.begin(), seen.end(), 0);
      ans += dfs(u);
    }
    return ans;
  }
};
```

7.5 Maximum flow minimum cost

```
//Complexity: O(|V|*|E|^2*log(|E|))
template <class type>
struct mcmf {
  struct edge { int u, v, cap, flow; type cost; };
  int n;
  vector<edge> ed;
  vector<vector<int>> g;
  vector<int> p;
  vector<type> d, phi;
  mcmf(int n) : n(n), g(n), p(n), d(n), phi(n) {}
  void add_edge(int u, int v, int cap, type cost)
      {
    g[u].push_back(ed.size());
    ed.push_back({u, v, cap, 0, cost});
    g[v].push_back(ed.size());
    ed.push_back({v, u, 0, 0, -cost});
  }
  bool dijkstra(int s, int t) {
    fill(d.begin(), d.end(), INF_TYPE);
    fill(p.begin(), p.end(), -1);
    set<pair<type, int>> q;
    d[s] = 0;
    for(q.insert({d[s], s}); q.size();) {
      int u = (*q.begin()).second;
          q.erase(q.begin());
      for(auto v : g[u]) {
        auto &e = ed[v];
        type nd = d[e.u]+e.cost+phi[e.u]-phi[e.v];
        if(0 < (e.cap-e.flow) && nd < d[e.v]) {
          q.erase({d[e.v], e.v});
          d[e.v] = nd; p[e.v] = v;
          q.insert({d[e.v], e.v});
        }
      }
    }
    for(int i = 0; i < n; i++) phi[i] =
        min(INF_TYPE, phi[i]+d[i]);
    return d[t] != INF_TYPE;
  }
  pair<int, type> max_flow(int s, int t) {
    type mc = 0;
    int mf = 0;
    fill(phi.begin(), phi.end(), 0);
    while(dijkstra(s, t)) {
      int flow = INF;
      for(int v = p[t]; v != -1; v = p[ ed[v].u ])
        flow = min(flow, ed[v].cap-ed[v].flow);
      for(int v = p[t]; v != -1; v = p[ ed[v].u ])
          {
        edge &e1 = ed[v];
        edge &e2 = ed[v^1];
        mc += e1.cost*flow;
        e1.flow += flow;
        e2.flow -= flow;
      }
      mf += flow;
    }
    return {mf, mc};
  }
};
```

7.6 Stoer Wagner

```
//Complexity: O(|V|^3)
//Tested: https://tinyurl.com/y8eu433d
 struct stoer_wagner {
  int n;
  vector<vector<int>> g;
  stoer_wagner(int n) : n(n), g(n,
      vector<int>(n)) {}
  void add_edge(int a, int b, int w) { g[a][b] =
      g[b][a] = w; }
  pair<int, vector<int>> min_cut() {
    vector<int> used(n);
    vector<int> cut, best_cut;
    int best_weight = -1;
    for(int p = n-1; p >= 0; --p) {
      vector<int> w = g[0];
      vector<int> added = used;
      int prv, lst = 0;
      for(int i = 0; i < p; ++i) {
        prv = lst; lst = -1;
        for(int j = 1; j < n; ++j)
          if(!added[j] && (lst == -1 || w[j] >
              w[lst]))
            lst = j;
        if(i == p-1) {
          for(int j = 0; j < n; j++)
            g[prv][j] += g[lst][j];
          for(int j = 0; j < n; j++)
            g[j][prv] = g[prv][j];
          used[lst] = true;
          cut.push_back(lst);
          if(best_weight == -1 || w[lst] <
              best_weight) {
            best_cut = cut;
            best_weight = w[lst];
          }
```

```
      } else {
        for(int j = 0; j < n; j++)
          w[j] += g[lst][j];
        added[lst] = true;
      }
    }
  }
  return {best_weight, best_cut}; //best_cut
      contains all nodes in the same set
 }
};
```

7.7 Weighted matching

```
//Complexity: O(|V|^3)
typedef int type;
struct matching_weighted {
  int l, r;
  vector<vector<type>> c;
  matching_weighted(int l, int r) : l(l), r(r),
      c(l, vector<type>(r)) {
    assert(l <= r);
  }
  void add_edge(int a, int b, type cost) {
      c[a][b] = cost; }
  type matching() {
    vector<type> v(r), d(r); //v: potential
    vector<int> ml(l, -1), mr(r, -1); //matching
        pairs
    vector<int> idx(r), prev(r);
    iota(idx.begin(), idx.end(), 0);
    auto residue = [&](int i, int j) { return
        c[i][j]-v[j]; };
    for(int f = 0; f < l; ++f) {
      for(int j = 0; j < r; ++j) {
        d[j] = residue(f, j);
        prev[j] = f;
      }
      type w;
      int j, l;
      for (int s = 0, t = 0;;) {
        if(s == t) {
          l = s;
          w = d[ idx[t++] ];
          for(int k = t; k < r; ++k) {
            j = idx[k];
            type h = d[j];
            if (h <= w) {
              if (h < w) t = s, w = h;
              idx[k] = idx[t];
              idx[t++] = j;
            }
          }
          for (int k = s; k < t; ++k) {
            j = idx[k];
            if (mr[j] < 0) goto aug;
          }
        }
        int q = idx[s++], i = mr[q];
        for (int k = t; k < r; ++k) {
          j = idx[k];
          type h = residue(i, j) - residue(i, q) +
              w;
          if (h < d[j]) {
            d[j] = h;
            prev[j] = i;
            if(h == w) {
              if(mr[j] < 0) goto aug;
              idx[k] = idx[t];
              idx[t++] = j;
            }
          }
        }
      }
      aug: for (int k = 0; k < l; ++k)
        v[ idx[k] ] += d[ idx[k] ] - w;
      int i;
      do {
        mr[j] = i = prev[j];
        swap(j, ml[i]);
      } while (i != f);
    }
    type opt = 0;
    for (int i = 0; i < l; ++i)
      opt += c[i][ml[i]]; //(i, ml[i]) is a
          solution
    return opt;
  }
};
```

8 Strings

8.1 Aho corasick

```
//Complexity: O(|text|+SUM(|pattern_i|)+matches)
const static int alpha = 26;
int trie[N][alpha], fail[N], nodes;
void add(string &s, int i) {
  int cur = 0;
  for(char c : s) {
    int x = c-'a';
    if(!trie[cur][x]) trie[cur][x] = ++nodes;
    cur = trie[cur][x];
  }
  //cnt_word[cur]++;
  //end_word[cur] = i; //for i > 0
}
void build() {
  queue<int> q; q.push(0);
  while(q.size()) {
    int u = q.front(); q.pop();
    for(int i = 0; i < alpha; ++i) {
      int v = trie[u][i];
      if(!v) trie[u][i] = trie[ fail[u] ][i];
          //construir automata
      else q.push(v);
      if(!u || !v) continue;
      fail[v] = trie[ fail[u] ][i];
      //fail_out[v] = end_word[ fail[v] ] ?
          fail[v] : fail_out[ fail[v] ];
      //cnt_word[v] += cnt_word[ fail[v] ];
          //obtener informacion del fail_padre
    }
  }
}
```

8.2 Hashing

```
//1000234999, 1000567999, 1000111997, 1000777121
const int MODS[] = { 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 1ll*a*b%mod; }
inline ll operator ! (const mint a) { return
    (ll(a.first)<<32)|ll(a.second); }
inline mint operator + (const mint a, const mint
    b) {
  return {add(a.first, b.first, MODS[0]),
      add(a.second, b.second, MODS[1])};
}
inline mint operator - (const mint a, const mint
    b) {
  return {sbt(a.first, b.first, MODS[0]),
      sbt(a.second, b.second, MODS[1])};
}
inline mint operator * (const mint a, const mint
    b) {
  return {mul(a.first, b.first, MODS[0]),
      mul(a.second, b.second, MODS[1])};
}
mint base[MAXN];
void prepare() {
  base[0] = ONE;
  for(int i = 1; i < MAXN; i++) base[i] =
      base[i-1]*BASE;
}
template <class type>
struct hashing {
  vector<mint> code;
  hashing(type &t) {
    code.resize(t.size()+1);
    code[0] = ZERO;
    for (int i = 1; i < code.size(); ++i)
      code[i] = code[i-1]*BASE + mint{t[i-1],
          t[i-1]};
  }
  mint query(int l, int r) {
    return code[r+1] - code[l]*base[r-l+1];
  }
};
```

8.3 Kmp automaton

```
//Complexity: O(|N|*alphabet)
const int alpha = 256;
vector<vector<int>> kmp_automaton(string &t) {
  vector<vector<int>> aut( t.size() + 1,
       vector<int> ( alpha ) );
  aut[0][ t[0] ] = 1;
  for(int i = 1, j = 0; i <= t.size(); ++i) {
    aut[i] = aut[j];
    if(i < t.size()) {
      aut[i][ t[i] ] = i+1;
      j = aut[j][ t[i] ];
    }
  }
  return aut;
}
```

8.4 Kmp

```
//Complexity: O(|N|)
vector<int> get_phi(string &p) {
  vector<int> phi(p.size());
  phi[0] = 0;
  for(int i = 1, j = 0; i < p.size(); ++i ) {
    while(j > 0 && p[i] != p[j] ) j = phi[j-1];
    if(p[i] == p[j]) ++j;
    phi[i] = j;
  }
  return phi;
}
int get_matches(string &t, string &p) {
  vector<int> phi = get_phi(p);
  int matches = 0;
  for(int i = 0, j = 0; i < t.size(); ++i ) {
    while(j > 0 && t[i] != p[j] ) j = phi[j-1];
    if(t[i] == p[j]) ++j;
    if(j == p.size()) {
      matches++;
      j = phi[j-1];
    }
  }
  return matches;
}
```

8.5 Manacher

```
//Complexity: O(|N|)
//to = i - from[i];
//len = to - from[i] + 1 = i - 2 * from[i] + 1;
vector<int> manacher(string &s) {
  int n = s.size(), p = 0, pr = -1;
  vector<int> from(2*n-1);
  for(int i = 0; i < 2*n-1; ++i) {
    int r = i <= 2*pr ? min(p - from[2*p - i],
        pr) : i/2;
    int l = i - r;
    while(l > 0 && r < n-1 && s[l-1] == s[r+1])
        --l, ++r;
    from[i] = l;
    if (r > pr) {
      pr = r;
      p = i;
    }
  }
  return from;
}
```

8.6 Minimun expression

```
//Complexity: O(|N|)
int minimum_expression(string s) {
  s = s+s;
  int len = s.size(), i = 0, j = 1, k = 0;
```

```
  while(i+k < len && j+k < len) {
    if(s[i+k] == s[j+k]) k++;
    else if(s[i+k] > s[j+k]) i = i+k+1, k = 0;
    else j = j+k+1, k = 0;
    if(i == j) j++;
  }
  return min(i, j);
}
```

8.7 Palindromic Tree

```
//Complexity: O(|N|*log(|alphabet|))
struct palindromic_tree {
  struct node{
    int len, link;
    map <char,int> next;
  };
  vector <node> pt;
  int last, it;
  string s;
  palindromic_tree (string str = "") {
    pt.reserve ( s.size()+2 );
    s = '#', it = 1, last = 1; //# isn't used in
        string
    add_node (), add_node();
    pt[1].len = -1, pt[0].link = 1;
    for ( char &c: str ) add_letter ( c );
  }
  int add_node () {
    pt.push_back({});
    return pt.size()-1;
  }
  int get_link ( int v ) {
    while ( s[it-pt[v].len-2] != s[it-1] ) v =
        pt[v].link;
    return v;
  }
  void add_letter ( char c ) {
    s += c, ++it;
    last = get_link ( last );
    if ( !pt[last].next.count (c) ) {
      int curr = add_node ();
      pt[curr].len = pt[last].len+2;
      pt[curr].link =
          pt[get_link(pt[last].link)].next[c];
      pt[last].next[c] = curr;
    }
    last = pt[last].next[c];
  }
  node& operator[](int i) { return pt[i]; }
  int size() { return pt.size(); }
};
```

8.8 Suffix array

```
//Complexity: O(|N|*log(|N|))
const int alpha = 400;
struct suffix_array { //s MUST not have 0 value
  vector<int> sa, pos, lcp;
  suffix_array(string s) {
    s.push_back('$'); //always add something less
        to input, so it stays in pos 0
    int n = s.size(), mx = max(alpha, n)+2;
    vector<int> a(n), a1(n), c(n+1), c1(n+1),
        head(mx), cnt(mx);
    pos = lcp = a;
    for(int i = 0; i < n; i++) c[i] = s[i], a[i]
        = i, cnt[ c[i] ]++;
    for(int i = 0; i < mx-1; i++) head[i+1] =
        head[i] + cnt[i];
    for(int k = 0; k < n; k = max(1, k<<1)) {
      for(int i = 0; i < n; i++) {
        int j = (a[i] - k + n) % n;
        a1[ head[ c[j] ]++ ] = j;
      }
      swap(a1, a);
      for(int i = 0, x = a[0], y, col = 0; i < n;
          i++, x = a[i], y = a[i-1]) {
        c1[x] = (i && c[x] == c[y] && c[x+k] ==
            c[y+k]) ? col : ++col;
        if(!i || c1[x] != c1[y]) head[col] = i;
      }
      swap(c1, c);
      if(c[ a[n-1] ] == n) break;
    }
    swap(sa, a);
    for(int i = 0; i < n; i++) pos[ sa[i] ] = i;
    for(int i = 0, k = 0, j; i < n; lcp[ pos[i++]
        ] = k) { //lcp[i, i+1]
      if(pos[i] == n-1) continue;
      for(k = max(0, k-1), j = sa[ pos[i]+1 ];
          s[i+k] == s[j+k]; k++);
    }
  }
  int& operator[] ( int i ){ return sa[i]; }
};
```

8.9 Suffix automaton

```
//Complexity: O(|N|*log(|alphabet|))
struct suffix_automaton {
  struct node {
    int len, link, cnt, first_pos; bool end;
        //cnt is endpos size, first_pos is
        minimum of endpos
    map<char, int> next;
  };
  vector<node> sa;
  int last;
  suffix_automaton() {}
  suffix_automaton(string &s) {
    sa.reserve(s.size()*2);
    last = add_node();
    sa[last].len = sa[last].cnt =
        sa[last].first_pos = sa[last].end = 0;
    sa[last].link = -1;
    for(char c : s) sa_append(c);
    mark_suffixes();
    build_cnt();
  }
  int add_node() {
    sa.push_back({});
    return sa.size()-1;
  }
  void mark_suffixes() {
    //t0 is not suffix
```

```
  for(int cur = last; cur; cur = sa[cur].link)
    sa[cur].end = 1;
}
//This is O(N*log(N)). Can be done O(N) by
    doing dfs and counting paths to terminal
    nodes.
void build_cnt() {
  vector<int> order(sa.size()-1);
  iota(order.begin(), order.end(), 1);
  sort(order.begin(), order.end(), [&](int a,
      int b) { return sa[a].len > sa[b].len; });
  for(auto &i : order) sa[ sa[i].link ].cnt +=
      sa[i].cnt;
  sa[0].cnt = 0; //t0 is empty string
}
void sa_append(char c) {
  int cur = add_node();
  sa[cur].len = sa[last].len + 1;
  sa[cur].end = 0; sa[cur].cnt = 1;
  sa[cur].first_pos = sa[cur].len-1;
  int p = last;
  while(p != -1 && !sa[p].next[c] ){
    sa[p].next[c] = cur;
    p = sa[p].link;
  }
  if(p == -1) sa[cur].link = 0;
  else {
    int q = sa[p].next[c];
    if(sa[q].len == sa[p].len+1) sa[cur].link =
        q;
    else {
      int clone = add_node();
      sa[clone] = sa[q];
      sa[clone].len = sa[p].len+1;
      sa[clone].cnt = 0;
      sa[q].link = sa[cur].link = clone;
      while(p != -1 && sa[p].next[c] == q) {
        sa[p].next[c] = clone;
        p = sa[p].link;
      }
    }
  }
  last = cur;
  }
  node& operator[](int i) { return sa[i]; }
};
```

8.10 Z algorithm

```
//Complexity: O(|N|)
vector<int> z_algorithm (string &s) {
  int n = s.size();
  vector<int> z(n);
  int l = 0, r = 0;
  for(int i = 1; i < n; ++i) {
    z[i] = max(0, min(z[i-l], r-i+1));
    while (i+z[i] < n && s[z[i]] == s[i+z[i]])
      l = i, r = i+z[i], ++z[i];
  }
  return z;
}
```

9 Utilities

9.1 Makefile

```
compile:
        g++ -std=c++17 -static -DLOCAL -O2 -Wall
            -Wshadow -Wno-unused-result -o sol
            sol.cpp
```

9.2 Pragma optimizations

```
#pragma GCC optimize ("O3")
#pragma GCC target ("sse4")
#pragma GCC target ("avx,tune=native")
```

9.3 Random

```
mt19937 / mt19937_64
rng(chrono::steady_clock::now().
            time_since_epoch().count())
shuffle(permutation.begin(), permutation.end(),
    rng)
uniform_int_distribution<T> /
    uniform_real_distribution<T> dis(fr, to);
dis(rng)
```

9.4 gen

```
int main(int argv, char * argc[]) {
  srand( atoi( argc[1] ) );
}
```

9.5 test

```
for((i = 0; ;++i));do
  echo $i
  ./gen $i > int
done
```

9.6 vimrc

```
set spr ai sw=2 ts=2
au vimenter *.cpp :vert term ++cols=60
nmap <f5> :w <cr> <c-l> clear <cr> make <cr>
nmap <f6> :w <cr> <c-l> clear <cr> make && ./sol
    < in <cr>
tnoremap <f7> ./sol < in <cr>
nmap <c-l> <c-w>l
tnoremap <c-h> <c-w>h
nmap QQ :qa! <cr>
```