AdMU Progvar

Team Notebook

06/11/2019

Contents

1. Code Templates

```
#include <bits/stdc++.h> ---------------------------------//001
typedef long long ll; ----------------------------------//002
typedef unsigned long long ull; -------------------------//003
typedef std::pair<int, int> ii; -------------------------//004
typedef std::pair<int, ii> iii; -------------------------//005
typedef std::vector<int> vi; ----------------------------//006
typedef std::vector<vi> vvi; ----------------------------//007
typedef std::vector<ii> vii; ----------------------------//008
typedef std::vector<iii> viii; --------------------------//009
const int INF = ~(1<<31); -------------------------------//00a
const ll LINF = (1LL << 60); ----------------------------//00b
const int MAXN = 1e5+1; ---------------------------------//00c
const double EPS = 1e-9; --------------------------------//00d
const double pi = acos(-1); -----------------------------//00e
```

2. Data Structures

2.1. Union Find.

```
struct union_find { ------------------------------------//2f3
- vi p; union_find(int n) : p(n, -1) { } ----------------//2f4
- int find(int x) { return p[x] < 0 ? x : p[x] = find(p[x]); }
- bool unite(int x, int y) { ----------------------------//2f6
--- int xp = find(x), yp = find(y); ---------------------//2f7
--- if (xp == yp)       return false; -------------------//2f8
--- if (p[xp] > p[yp])  std::swap(xp,yp); ---------------//2f9
--- p[xp] += p[yp], p[yp] = xp; -------------------------//2fa
--- return true; ----------------------------------------//2fb
- } -----------------------------------------------------//2fc
- int size(int x) { return -p[find(x)]; } ---------------//2fd
}; ------------------------------------------------------//2fe
```

2.2. Segment Tree.

2.2.1. *Recursive, Point-update Segment Tree.*

```
struct segtree { ----------------------------------------//1df
- int i, j, val; ----------------------------------------//1e0
- segtree *l, *r; ---------------------------------------//1e1
- segtree(vi &ar, int _i, int _j) : i(_i), j(_j) { ------//1e2
--- if (i == j) { ---------------------------------------//1e3
----- val = ar[i]; --------------------------------------//1e4
```

```
----- l = r = NULL;
--- } else {
----- int k = (i+j) >> 1;
----- l = new segtree(ar, i, k);
----- r = new segtree(ar, k+1, j);
----- val = l->val + r->val;
--- }
- }
- void update(int _i, int _val) {
--- if (_i <= i and j <= _i) {
----- val += _val;
--- } else if (_i < i or j < _i) {
----- // do nothing
--- } else {
----- l->update(_i, _val);
----- r->update(_i, _val);
----- val = l->val + r->val;
--- }
- }
- int query(int _i, int _j) {
--- if (_i <= i and j <= _j) {
----- return val;
--- } else if (_j < i or j < _i) {
----- return 0;
--- } else {
----- return l->query(_i, _j) + r->query(_i, _j);
--- }
- }
};
```

2.2.2. *Iterative, Point-update Segment Tree.*

```
struct segtree {
- int n;
- int *vals;
- segtree(vi &ar, int n) {
--- this->n = n;
--- vals = new int[2*n];
--- for (int i = 0; i < n; ++i)
----- vals[i+n] = ar[i];
--- for (int i = n-1; i > 0; --i)
----- vals[i] = vals[i<<1] + vals[i<<1|1];
- }
- void update(int i, int v) {
--- for (vals[i += n] += v; i > 1; i >>= 1)
----- vals[i>>1] = vals[i] + vals[i^1];
- }
- int query(int l, int r) {
--- int res = 0;
--- for (l += n, r += n+1; l < r; l >>= 1, r >>= 1) {
----- if (l&1)  res += vals[l++];
----- if (r&1)  res += vals[--r];
--- }
--- return res;
- }
};
```

2.2.3. *Pointer-based, Range-update Segment Tree.*

```
struct segtree {
- int i, j, val, temp_val = 0;
- segtree *l, *r;
- segtree(vi &ar, int _i, int _j) : i(_i), j(_j) {
--- if (i == j) {
----- val = ar[i];
----- l = r = NULL;
--- } else {
----- int k = (i + j) >> 1;
----- l = new segtree(ar, i, k);
----- r = new segtree(ar, k+1, j);
----- val = l->val + r->val;
--- }
- }
- void visit() {
--- if (temp_val) {
----- val += (j-i+1) * temp_val;
----- if (l) {
------- l->temp_val += temp_val;
------- r->temp_val += temp_val;
----- }
----- temp_val = 0;
--- }
- }
- void increase(int _i, int _j, int _inc) {
--- visit();
--- if (_i <= i && j <= _j) {
----- temp_val += _inc;
----- visit();
--- } else if (_j < i or j < _i) {
----- // do nothing
--- } else {
----- l->increase(_i, _j, _inc);
----- r->increase(_i, _j, _inc);
----- val = l->val + r->val;
--- }
- }
- int query(int _i, int _j) {
--- visit();
--- if (_i <= i and j <= _j) {
----- return val;
--- } else if (_j < i || j < _i) {
----- return 0;
--- } else {
----- return l->query(_i, _j) + r->query(_i, _j);
--- }
- }
};
```

2.2.4. *Array-based, Range-update Segment Tree.*

```
struct segtree {
- int n, *vals, *deltas;
- segtree(vi &ar) {
--- n = ar.size();
--- vals = new int[4*n];
--- deltas = new int[4*n];
--- build(ar, 1, 0, n-1);
- }
- void build(vi &ar, int p, int i, int j) {
--- deltas[p] = 0;
--- if (i == j)
----- vals[p] = ar[i];
--- else {
----- int k = (i + j) / 2;
----- build(ar, p<<1, i, k);
----- build(ar, p<<1|1, k+1, j);
----- pull(p);
--- }
- }
- void pull(int p) {
--- vals[p] = vals[p<<1] + vals[p<<1|1];
- }
- void push(int p, int i, int j) {
--- if (deltas[p]) {
----- vals[p] += (j - i + 1) * deltas[p];
----- if (i != j) {
------- deltas[p<<1] += deltas[p];
------- deltas[p<<1|1] += deltas[p];
----- }
----- deltas[p] = 0;
--- }
- }
- void update(int _i, int _j, int v,
------------ int p, int i, int j) {
--- push(p, i, j);
--- if (_i <= i && j <= _j) {
----- deltas[p] += v;
----- push(p, i, j);
--- } else if (_j < i || j < _i) {
----- // do nothing
--- } else {
----- int k = (i + j) / 2;
----- update(_i, _j, v, p<<1, i, k);
----- update(_i, _j, v, p<<1|1, k+1, j);
----- pull(p);
--- }
- }
- int query(int _i, int _j,
---------- int p, int i, int j) {
--- push(p, i, j);
--- if (_i <= i and j <= _j) {
----- return vals[p];
--- } else if (_j < i || j < _i) {
----- return 0;
--- } else {
----- int k = (i + j) / 2;
----- return query(_i, _j, p<<1, i, k) +
----------- query(_i, _j, p<<1|1, k+1, j);
--- }
- }
};
```

2.2.5. *Persistent Segment Tree (Point-update).*

```
struct node {  int l, r, lid, rid, val;  }; --------------//17f
struct segtree { -------------------------------------------//180
- node *nodes; ---------------------------------------------//181
- int n, node_cnt = 0; -------------------------------------//182
- segtree(int n, int capacity) { ---------------------------//183
--- this->n = n; -------------------------------------------//184
--- nodes = new node[capacity]; ----------------------------//185
- } --------------------------------------------------------//186
- int build (vi &ar, int l, int r) { -----------------------//187
--- if (l > r)  return -1; ---------------------------------//188
--- int id = node_cnt++; -----------------------------------//189
--- nodes[id].l = l; ---------------------------------------//18a
--- nodes[id].r = r; ---------------------------------------//18b
--- if (l == r) { ------------------------------------------//18c
----- nodes[id].lid = -1; ----------------------------------//18d
----- nodes[id].rid = -1; ----------------------------------//18e
----- nodes[id].val = ar[l]; -------------------------------//18f
--- } else { -----------------------------------------------//190
----- int m = (l + r) / 2; ---------------------------------//191
----- nodes[id].lid = build(ar, l, m); ---------------------//192
----- nodes[id].rid = build(ar, m+1, r); -------------------//193
----- nodes[id].val = nodes[nodes[id].lid].val + ----------//194
-------------------- nodes[nodes[id].rid].val; -----------//195
--- } ------------------------------------------------------//196
--- return id; ---------------------------------------------//197
- } --------------------------------------------------------//198
- int update(int id, int idx, int delta) { -----------------//199
--- if (id == -1) ------------------------------------------//19a
----- return -1; -------------------------------------------//19b
--- if (idx < nodes[id].l or nodes[id].r < idx) -----------//19c
----- return id; -------------------------------------------//19d
--- int nid = node_cnt++; ----------------------------------//19e
--- nodes[nid].l = nodes[id].l; ----------------------------//19f
--- nodes[nid].r = nodes[id].r; ----------------------------//1a0
--- nodes[nid].lid = update(nodes[id].lid, idx, delta); --//1a1
--- nodes[nid].rid = update(nodes[id].rid, idx, delta); --//1a2
--- nodes[nid].val = nodes[id].val + delta; ---------------//1a3
--- return nid; --------------------------------------------//1a4
- } --------------------------------------------------------//1a5
- int query(int id, int l, int r) { ------------------------//1a6
--- if (r < nodes[id].l or nodes[id].r < l) ---------------//1a7
----- return 0; --------------------------------------------//1a8
--- if (l <= nodes[id].l and nodes[id].r <= r) ------------//1a9
----- return nodes[id].val; --------------------------------//1aa
--- return query(nodes[id].lid, l, r) + -------------------//1ab
---------- query(nodes[id].rid, l, r); --------------------//1ac
-- } -------------------------------------------------------//1ad
 }; --------------------------------------------------------//1ae
```

2.2.6. *2D Segment Tree.*

```
struct segtree_2d { ----------------------------------------//10d
- int n, m, **ar; ------------------------------------------//10e
- segtree_2d(int n, int m) { -------------------------------//10f
--- this->n = n;    this->m = m; --------------------------//110
--- ar = new int[n]; ---------------------------------------//111
--- for (int i = 0; i < n; ++i) { --------------------------//112
----- ar[i] = new int[m]; ----------------------------------//113
----- for (int j = 0; j < m; ++j) --------------------------//114
------- ar[i][j] = 0; --------------------------------------//115
--- } ------------------------------------------------------//116
- } --------------------------------------------------------//117
- void update(int x, int y, int v) { -----------------------//118
--- ar[x + n][y + m] = v; ----------------------------------//119
--- for (int i = x + n; i > 0; i >>= 1) { ------------------//11a
----- for (int j = y + m; j > 0; j >>= 1) { ----------------//11b
------- ar[i>>1][j] = min(ar[i][j], ar[i^1][j]); ---------//11c
------- ar[i][j>>1] = min(ar[i][j], ar[i][j^1]); ---------//11d
- }}} // just call update one by one to build -------------//11e
- int query(int x1, int x2, int y1, int y2) { --------------//11f
--- int s = INF; -------------------------------------------//120
--- if(~x2) for(int a=x1+n, b=x2+n+1; a<b; a>>=1, b>>=1) {//121
----- if (a & 1) s = min(s, query(a++, -1, y1, y2)); ----//122
----- if (b & 1) s = min(s, query(--b, -1, y1, y2)); ----//123
--- } else for (int a=y1+m, b=y2+m+1; a<b; a>>=1, b>>=1) {//124
----- if (a & 1) s = min(s, ar[x1][a++]); ----------------//125
----- if (b & 1) s = min(s, ar[x1][--b]); ----------------//126
--- } return s; --------------------------------------------//127
- } --------------------------------------------------------//128
}; ---------------------------------------------------------//129
```

2.3. Fenwick Tree.

2.3.1. *Fenwick Tree w/ Point Queries.*

```
struct fenwick { -------------------------------------------//0c7
- vi ar; ---------------------------------------------------//0c8
- fenwick(vi &_ar) : ar(_ar.size(), 0) { -------------------//0c9
--- for (int i = 0; i < ar.size(); ++i) { ------------------//0ca
----- ar[i] += _ar[i]; -------------------------------------//0cb
----- int j = i | (i+1); -----------------------------------//0cc
----- if (j < ar.size()) -----------------------------------//0cd
------- ar[j] += ar[i]; ------------------------------------//0ce
--- } ------------------------------------------------------//0cf
- } --------------------------------------------------------//0d0
- int sum(int i) { -----------------------------------------//0d1
--- int res = 0; -------------------------------------------//0d2
--- for (; i >= 0; i = (i & (i+1)) - 1) -------------------//0d3
----- res += ar[i]; ----------------------------------------//0d4
--- return res; --------------------------------------------//0d5
- } --------------------------------------------------------//0d6
- int sum(int i, int j) {  return sum(j) - sum(i-1);  } --//0d7
- void add(int i, int val) { -------------------------------//0d8
--- for (; i < ar.size(); i |= i+1) ------------------------//0d9
----- ar[i] += val; ----------------------------------------//0da
- } --------------------------------------------------------//0db
- int get(int i) { -----------------------------------------//0dc
--- int res = ar[i]; ---------------------------------------//0dd
--- if (i) { -----------------------------------------------//0de
----- int lca = (i & (i+1)) - 1; ---------------------------//0df
----- for (--i; i != lca; i = (i&(i+1))-1) -----------------//0e0
------- res -= ar[i]; --------------------------------------//0e1
--- } ------------------------------------------------------//0e2
--- return res; --------------------------------------------//0e3
- } --------------------------------------------------------//0e4
- void set(int i, int val) {  add(i, -get(i) + val);  } --//0e5
- // range update, point query // -------------------------//0e6
- void add(int i, int j, int val) { ------------------------//0e7
--- add(i, val); -------------------------------------------//0e8
--- add(j+1, -val); ----------------------------------------//0e9
- } --------------------------------------------------------//0ea
- int get1(int i) { return sum(i); } ----------------------//0eb
- ////////////////////////////// -------------------------//0ec
}; ---------------------------------------------------------//0ed
```

2.3.2. *Fenwick Tree w/ Max Queries.*

```
struct fenwick { -------------------------------------------//0ee
- vi ar; ---------------------------------------------------//0ef
- fenwick(vi &_ar) : ar(_ar.size(), 0) { -------------------//0f0
--- for (int i = 0; i < ar.size(); ++i) { ------------------//0f1
----- ar[i] = std::max(ar[i], _ar[i]); ---------------------//0f2
----- int j = i | (i+1); -----------------------------------//0f3
----- if (j < ar.size()) -----------------------------------//0f4
------- ar[j] = std::max(ar[j], ar[i]); --------------------//0f5
--- } ------------------------------------------------------//0f6
- } --------------------------------------------------------//0f7
- void set(int i, int v) { ---------------------------------//0f8
--- for (; i < ar.size(); i |= i+1) ------------------------//0f9
----- ar[i] = std::max(ar[i], v); --------------------------//0fa
- } --------------------------------------------------------//0fb
- // max[0..i] ---------------------------------------------//0fc
- int max(int i) { -----------------------------------------//0fd
--- int res = -INF; ----------------------------------------//0fe
--- for (; i >= 0; i = (i & (i+1)) - 1) -------------------//0ff
----- res = std::max(res, ar[i]); --------------------------//100
--- return res; --------------------------------------------//101
- } --------------------------------------------------------//102
}; ---------------------------------------------------------//103
```

2.4. Treap.

2.4.1. *Explicit Treap.*

2.4.2. *Implicit Treap.*

```
struct cartree { -------------------------------------------//28b
- typedef struct _Node { -----------------------------------//28c
--- int node_val, subtree_val, delta, prio, size; ---------//28d
--- _Node *l, *r; ------------------------------------------//28e
--- _Node(int val) : node_val(val), subtree_val(val), ----//28f
------- delta(0), prio((rand()<<16)^rand()), size(1), ----//290
------- l(NULL), r(NULL) {} --------------------------------//291
--- ~_Node() { delete l; delete r; } -----------------------//292
- } *Node; -------------------------------------------------//293
- int get_subtree_val(Node v) { ----------------------------//294
--- return v ? v->subtree_val : 0;  } ---------------------//295
- int get_size(Node v) { return v ? v->size : 0; } -------//296
- void apply_delta(Node v, int delta) { --------------------//297
--- if (!v) return; ----------------------------------------//298
--- v->delta += delta; -------------------------------------//299
--- v->node_val += delta; ----------------------------------//29a
--- v->subtree_val += delta * get_size(v); ----------------//29b
- } --------------------------------------------------------//29c
- void push_delta(Node v) { --------------------------------//29d
```

```
--- if (!v) return; --------------------------------------//29e
--- apply_delta(v->l, v->delta); -------------------------//29f
--- apply_delta(v->r, v->delta); -------------------------//2a0
--- v->delta = 0; ----------------------------------------//2a1
- } ------------------------------------------------------//2a2
- void update(Node v) { ----------------------------------//2a3
--- if (!v) return; --------------------------------------//2a4
--- v->subtree_val = get_subtree_val(v->l) + v->node_val -//2a5
----------------- + get_subtree_val(v->r); ---------------//2a6
--- v->size = get_size(v->l) + 1 + get_size(v->r); -------//2a7
- } ------------------------------------------------------//2a8
- Node merge(Node l, Node r) { ---------------------------//2a9
--- push_delta(l);    push_delta(r); ---------------------//2aa
--- if (!l || !r)   return l ? l : r; --------------------//2ab
--- if (l->size <= r->size) { ----------------------------//2ac
----- l->r = merge(l->r, r); -----------------------------//2ad
----- update(l); -----------------------------------------//2ae
----- return l; ------------------------------------------//2af
--- } else { ---------------------------------------------//2b0
----- r->l = merge(l, r->l); -----------------------------//2b1
----- update(r); -----------------------------------------//2b2
----- return r; ------------------------------------------//2b3
--- } ----------------------------------------------------//2b4
- } ------------------------------------------------------//2b5
- void split(Node v, int key, Node &l, Node &r) { --------//2b6
--- push_delta(v); ---------------------------------------//2b7
--- l = r = NULL; ----------------------------------------//2b8
--- if (!v)     return; ----------------------------------//2b9
--- if (key <= get_size(v->l)) { -------------------------//2ba
----- split(v->l, key, l, v->l); -------------------------//2bb
----- r = v; ---------------------------------------------//2bc
--- } else { ---------------------------------------------//2bd
----- split(v->r, key - get_size(v->l) - 1, v->r, r); ----//2be
----- l = v; ---------------------------------------------//2bf
--- } ----------------------------------------------------//2c0
--- update(v); -------------------------------------------//2c1
- } ------------------------------------------------------//2c2
- Node root; ---------------------------------------------//2c3
public: --------------------------------------------------//2c4
- cartree() : root(NULL) {} ------------------------------//2c5
- ~cartree() { delete root; } ----------------------------//2c6
- int get(Node v, int key) { -----------------------------//2c7
--- push_delta(v); ---------------------------------------//2c8
--- if (key < get_size(v->l)) ----------------------------//2c9
----- return get(v->l, key); -----------------------------//2ca
--- else if (key > get_size(v->l)) -----------------------//2cb
----- return get(v->r, key - get_size(v->l) - 1); --------//2cc
--- return v->node_val; ----------------------------------//2cd
- } ------------------------------------------------------//2ce
- int get(int key) { return get(root, key); } ------------//2cf
- void insert(Node item, int key) { ----------------------//2d0
--- Node l, r; -------------------------------------------//2d1
--- split(root, key, l, r); ------------------------------//2d2
--- root = merge(merge(l, item), r); ---------------------//2d3
- } ------------------------------------------------------//2d4
- void insert(int key, int val) { ------------------------//2d5
--- insert(new _Node(val), key); -------------------------//2d6
- } ------------------------------------------------------//2d7
- void erase(int key) { ----------------------------------//2d8
--- Node l, m, r; ----------------------------------------//2d9
--- split(root, key + 1, m, r); --------------------------//2da
--- split(m, key, l, m); ---------------------------------//2db
--- delete m; --------------------------------------------//2dc
--- root = merge(l, r); ----------------------------------//2dd
- } ------------------------------------------------------//2de
- int query(int a, int b) { ------------------------------//2df
--- Node l1, r1; -----------------------------------------//2e0
--- split(root, b+1, l1, r1); ----------------------------//2e1
--- Node l2, r2; -----------------------------------------//2e2
--- split(l1, a, l2, r2); --------------------------------//2e3
--- int res = get_subtree_val(r2); -----------------------//2e4
--- l1 = merge(l2, r2); ----------------------------------//2e5
--- root = merge(l1, r1); --------------------------------//2e6
--- return res; ------------------------------------------//2e7
- } ------------------------------------------------------//2e8
- void update(int a, int b, int delta) { -----------------//2e9
--- Node l1, r1; -----------------------------------------//2ea
--- split(root, b+1, l1, r1); ----------------------------//2eb
--- Node l2, r2; -----------------------------------------//2ec
--- split(l1, a, l2, r2); --------------------------------//2ed
--- apply_delta(r2, delta); ------------------------------//2ee
--- l1 = merge(l2, r2); ----------------------------------//2ef
--- root = merge(l1, r1); --------------------------------//2f0
- } ------------------------------------------------------//2f1
- int size() { return get_size(root); }  }; --------------//2f2
```

2.4.3. *Persistent Treap.*

2.5. Splay Tree.

```
struct node *null; ---------------------------------------//236
struct node { --------------------------------------------//237
- node *left, *right, *parent; ---------------------------//238
- bool reverse; int size, value; -------------------------//239
- node*& get(int d) {return d == 0 ? left : right;} ------//23a
- node(int v=0): reverse(0), size(0), value(v) { ---------//23b
- left = right = parent = null ? null : this; ------------//23c
- }}; ----------------------------------------------------//23d
struct SplayTree { ---------------------------------------//23e
- node *root; --------------------------------------------//23f
- SplayTree(int arr[] = NULL, int n = 0) { ---------------//240
--- if (!null) null = new node(); ------------------------//241
--- root = build(arr, n); --------------------------------//242
- } // build a splay tree based on array values ----------//243
- node* build(int arr[], int n) { ------------------------//244
--- if (n == 0) return null; -----------------------------//245
--- int mid = n >> 1; ------------------------------------//246
--- node *p = new node(arr ? arr[mid] : 0); --------------//247
--- link(p, build(arr, mid), 0); -------------------------//248
--- link(p, build(arr? arr+mid+1 : NULL, n-mid-1), 1); ---//249
--- pull(p); return p; -----------------------------------//24a
- } // pull information from children (editable) ---------//24b
- void pull(node *p) { -----------------------------------//24c
--- p->size = p->left->size + p->right->size + 1; --------//24d
- } // push down lazy flags to children (editable) -------//24e
- void push(node *p) { -----------------------------------//24f
--- if (p != null && p->reverse) { -----------------------//250
----- swap(p->left, p->right); ---------------------------//251
----- p->left->reverse ^= 1; -----------------------------//252
----- p->right->reverse ^= 1; ----------------------------//253
----- p->reverse ^= 1; -----------------------------------//254
--- }} // assign son to be the new child of p ------------//255
- void link(node *p, node *son, int d) { -----------------//256
--- p->get(d) = son; -------------------------------------//257
--- son->parent = p; } -----------------------------------//258
- int dir(node *p, node *son) { --------------------------//259
--- return p->left == son ? 0 : 1;} ----------------------//25a
- void rotate(node *x, int d) { --------------------------//25b
--- node *y = x->get(d), *z = x->parent; -----------------//25c
--- link(x, y->get(d ^ 1), d); ---------------------------//25d
--- link(y, x, d ^ 1); -----------------------------------//25e
--- link(z, y, dir(z, x)); -------------------------------//25f
--- pull(x); pull(y);} -----------------------------------//260
- node* splay(node *p) { // splay node p to root ---------//261
--- while (p->parent != null) { --------------------------//262
----- node *m = p->parent, *g = m->parent; ---------------//263
----- push(g); push(m); push(p); -------------------------//264
----- int dm = dir(m, p), dg = dir(g, m); ----------------//265
----- if (g == null) rotate(m, dm); ----------------------//266
----- else if (dm == dg) rotate(g, dg), rotate(m, dm); ---//267
----- else rotate(m, dm), rotate(g, dg); -----------------//268
--- } return root = p; } ---------------------------------//269
- node* get(int k) { // get the node at index k ----------//26a
--- node *p = root; --------------------------------------//26b
--- while (push(p), p->left->size != k) { ----------------//26c
------- if (k < p->left->size) p = p->left; --------------//26d
------- else k -= p->left->size + 1, p = p->right; -------//26e
--- } ----------------------------------------------------//26f
--- return p == null ? null : splay(p); ------------------//270
- } // keep the first k nodes, the rest in r -------------//271
- void split(node *&r, int k) { --------------------------//272
--- if (k == 0) {r = root; root = null; return;} ---------//273
--- r = get(k - 1)->right; -------------------------------//274
--- root->right = r->parent = null; ----------------------//275
--- pull(root); } ----------------------------------------//276
- void merge(node *r) { //merge current tree with r ------//277
--- if (root == null) {root = r; return;} ----------------//278
--- link(get(root->size - 1), r, 1); ---------------------//279
--- pull(root); } ----------------------------------------//27a
- void assign(int k, int val) { // assign arr[k]= val ----//27b
--- get(k)->value = val; pull(root); } -------------------//27c
- void reverse(int L, int R) {// reverse arr[L...R] ------//27d
--- node *m, *r; split(r, R + 1); split(m, L); -----------//27e
--- m->reverse ^= 1; push(m); merge(m); merge(r); --------//27f
- } // insert a new node before the node at index k ------//280
- node* insert(int k, int v) { ---------------------------//281
--- node *r; split(r, k); --------------------------------//282
--- node *p = new node(v); p->size = 1; ------------------//283
--- link(root, p, 1); merge(r); --------------------------//284
--- return p; } ------------------------------------------//285
```

```
- void erase(int k) { // erase node at index k -----------//286
--- node *r, *m; ------------------------------------------//287
--- split(r, k + 1); split(m, k); -------------------------//288
--- merge(r); delete m;} ---------------------------------//289
}; -------------------------------------------------------//28a
```

2.6. **Ordered Statistics Tree.**

```
#include <ext/pb_ds/assoc_container.hpp> -----------------//104
#include <ext/pb_ds/tree_policy.hpp> --------------------//105
using namespace __gnu_pbds; -------------------------------//106
template <typename T> -------------------------------------//107
using indexed_set = std::tree<T, null_type, less<T>, -----//108
splay_tree_tag, tree_order_statistics_node_update>; ------//109
// indexed_set<int> t; t.insert(...); -------------------//10a
// t.find_by_order(index); // 0-based -------------------//10b
// t.order_of_key(key); ---------------------------------//10c
```

2.7. **Sparse Table.**

2.7.1. *1D Sparse Table.*

```
int lg[MAXN+1], spt[20][MAXN]; ---------------------------//202
void build(vi &arr, int n) { -----------------------------//203
- for (int i = 2; i <= n; ++i) lg[i] = lg[i>>1] + 1; -----//204
- for (int i = 0; i < n; ++i) spt[0][i] = arr[i]; --------//205
- for (int j = 0; (2 << j) <= n; ++j) --------------------//206
--- for (int i = 0; i + (2 << j) <= n; ++i) --------------//207
----- spt[j+1][i] = std::min(spt[j][i], spt[j][i+(1<<j)]);//208
} --------------------------------------------------------//209
int query(int a, int b) { --------------------------------//20a
- int k = lg[b-a+1], ab = b - (1<<k) + 1; ----------------//20b
- return std::min(spt[k][a], spt[k][ab]); ----------------//20c
} --------------------------------------------------------//20d
```

2.7.2. *2D Sparse Table.*

```
const int N = 100, LGN = 20; -----------------------------//20e
int lg[N], A[N][N], st[LGN][LGN][N][N]; ------------------//20f
void build(int n, int m) { -------------------------------//210
- for(int k=2; k<=std::max(n,m); ++k) lg[k] = lg[k>>1]+1; //211
- for(int i = 0; i < n; ++i) -----------------------------//212
--- for(int j = 0; j < m; ++j) ---------------------------//213
----- st[0][0][i][j] = A[i][j]; --------------------------//214
- for(int bj = 0; (2 << bj) <= m; ++bj) ------------------//215
--- for(int j = 0; j + (2 << bj) <= m; ++j) --------------//216
----- for(int i = 0; i < n; ++i) -------------------------//217
------- st[0][bj+1][i][j] = ------------------------------//218
--------- std::max(st[0][bj][i][j], ----------------------//219
----------------- st[0][bj][i][j + (1 << bj)]); ---------//21a
- for(int bi = 0; (2 << bi) <= n; ++bi) ------------------//21b
--- for(int i = 0; i + (2 << bi) <= n; ++i) --------------//21c
----- for(int j = 0; j < m; ++j) -------------------------//21d
------- st[bi+1][0][i][j] = ------------------------------//21e
--------- std::max(st[bi][0][i][j], ----------------------//21f
----------------- st[bi][0][i + (1 << bi)][j]); ---------//220
- for(int bi = 0; (2 << bi) <= n; ++bi) ------------------//221
--- for(int i = 0; i + (2 << bi) <= n; ++i) --------------//222
----- for(int bj = 0; (2 << bj) <= m; ++bj) --------------//223
------- for(int j = 0; j + (2 << bj) <= m; ++j) { --------//224
--------- int ik = i + (1 << bi); ------------------------//225
--------- int jk = j + (1 << bj); ------------------------//226
--------- st[bi+1][bj+1][i][j] = -------------------------//227
----------- std::max(std::max(st[bi][bj][i][j], ----------//228
---------------------------- st[bi][bj][ik][j]), --------//229
------------------- std::max(st[bi][bj][i][jk], ---------//22a
---------------------------- st[bi][bj][ik][jk])); ------//22b
------- } ------------------------------------------------//22c
} --------------------------------------------------------//22d
int query(int x1, int x2, int y1, int y2) { --------------//22e
- int kx = lg[x2 - x1 + 1],   ky = lg[y2 - y1 + 1]; ------//22f
- int x12 = x2 - (1<<kx) + 1, y12 = y2 - (1<<ky) + 1; ----//230
- return std::max(std::max(st[kx][ky][x1][y1], -----------//231
------------------------- st[kx][ky][x1][y12]), ---------//232
----------------- std::max(st[kx][ky][x12][y1], ---------//233
------------------------- st[kx][ky][x12][y12])); -------//234
} --------------------------------------------------------//235
```

3. Graphs

Using adjacency list:

```
struct graph { -------------------------------------------//542
- int n, *dist; ------------------------------------------//543
- vii *adj; ----------------------------------------------//544
- graph(int n) { -----------------------------------------//545
--- this->n = n; -----------------------------------------//546
--- adj = new vii[n]; ------------------------------------//547
--- dist = new int[n]; -----------------------------------//548
- } ------------------------------------------------------//549
- void add_edge(int u, int v, int w) { -------------------//54a
--- adj[u].push_back({v, w}); ----------------------------//54b
--- // adj[v].push_back({u, w}); -------------------------//54c
- } ------------------------------------------------------//54d
}; -------------------------------------------------------//54e
```

Using adjacency matrix:

```
struct graph { -------------------------------------------//54f
- int n, **mat; ------------------------------------------//550
- graph(int n) { -----------------------------------------//551
--- this->n = n; -----------------------------------------//552
--- mat = new int*[n]; -----------------------------------//553
--- for (int i = 0; i < n; ++i) { ------------------------//554
----- mat[i] = new int[n]; -------------------------------//555
----- for (int j = 0; j < n; ++j) ------------------------//556
------- mat[i][j] = INF; ---------------------------------//557
----- mat[i][i] = 0; -------------------------------------//558
--- } ----------------------------------------------------//559
- } ------------------------------------------------------//55a
- void add_edge(int u, int v, int w) { -------------------//55b
--- mat[u][v] = std::min(mat[u][v], w); ------------------//55c
--- // mat[v][u] = std::min(mat[v][u], w); ---------------//55d
- } ------------------------------------------------------//55e
}; -------------------------------------------------------//55f
```

Using edge list:

```
struct graph { -------------------------------------------//560
- int n; ------------------------------------------------//561
- std::vector<iii> edges; --------------------------------//562
- graph(int n) : n(n) {} ---------------------------------//563
- void add_edge(int u, int v, int w) { -------------------//564
--- edges.push_back({w, {u, v}}); ------------------------//565
- } ------------------------------------------------------//566
}; -------------------------------------------------------//567
```

3.1. **Single-Source Shortest Paths.**

3.1.1. *Dijkstra.*

```
#include "graph_template_adjlist.cpp" --------------------//7a1
// insert inside graph; needs n, dist[], and adj[] -------//7a2
void dijkstra(int s) { -----------------------------------//7a3
- for (int u = 0; u < n; ++u) ----------------------------//7a4
--- dist[u] = INF; ---------------------------------------//7a5
- dist[s] = 0; -------------------------------------------//7a6
- std::priority_queue<ii, vii, std::greater<ii> > pq; ----//7a7
- pq.push({0, s}); ---------------------------------------//7a8
- while (!pq.empty()) { ----------------------------------//7a9
--- int u = pq.top().second; -----------------------------//7aa
--- int d = pq.top().first; ------------------------------//7ab
--- pq.pop(); --------------------------------------------//7ac
--- if (dist[u] < d) -------------------------------------//7ad
----- continue; ------------------------------------------//7ae
--- dist[u] = d; -----------------------------------------//7af
--- for (auto &e : adj[u]) { -----------------------------//7b0
----- int v = e.first; -----------------------------------//7b1
----- int w = e.second; ----------------------------------//7b2
----- if (dist[v] > dist[u] + w) { -----------------------//7b3
------- dist[v] = dist[u] + w; ---------------------------//7b4
------- pq.push({dist[v], v}); ---------------------------//7b5
----- } --------------------------------------------------//7b6
--- } ----------------------------------------------------//7b7
- } ------------------------------------------------------//7b8
} --------------------------------------------------------//7b9
```

3.1.2. *Bellman-Ford.*

```
#include "graph_template_adjlist.cpp" --------------------//78d
// insert inside graph; needs n, dist[], and adj[] -------//78e
void bellman_ford(int s) { -------------------------------//78f
- for (int u = 0; u < n; ++u) ----------------------------//790
--- dist[u] = INF; ---------------------------------------//791
- dist[s] = 0; -------------------------------------------//792
- for (int i = 0; i < n-1; ++i) --------------------------//793
--- for (int u = 0; u < n; ++u) --------------------------//794
----- for (auto &e : adj[u]) -----------------------------//795
------- if (dist[u] + e.second < dist[e.first]) ----------//796
--------- dist[e.first] = dist[u] + e.second; -----------//797
} --------------------------------------------------------//798
// you can call this after running bellman_ford() --------//799
bool has_neg_cycle() { -----------------------------------//79a
- for (int u = 0; u < n; ++u) ----------------------------//79b
--- for (auto &e : adj[u]) -------------------------------//79c
----- if (dist[e.first] > dist[u] + e.second) -----------//79d
------- return true; -------------------------------------//79e
- return false; ------------------------------------------//79f
} --------------------------------------------------------//7a0
```

3.1.3. *SPFA.*

```
struct edge { -------------------------------------------//7c3
- int v; long long cost; --------------------------------//7c4
- edge(int v, long long cost): v(v), cost(cost) {} -------//7c5
}; ------------------------------------------------------//7c6
long long dist[N]; int vis[N]; bool inq[N]; --------------//7c7
void spfa(vector<edge*> adj[], int n, int s) { -----------//7c8
- fill(dist, dist + n, LLONG_MAX); -----------------------//7c9
- fill(vis, vis + n, 0); ---------------------------------//7ca
- fill(inq, inq + n, false); -----------------------------//7cb
- queue<int> q; q.push(s); -------------------------------//7cc
- for (dist[s] = 0; !q.empty(); q.pop()) { ---------------//7cd
--- int u = q.front(); inq[u] = false; -------------------//7ce
--- if (++vis[u] >= n) dist[u] = LLONG_MIN; --------------//7cf
--- for (int i = 0; i < adj[u].size(); ++i) { ------------//7d0
----- edge& e = *adj[u][i]; ------------------------------//7d1
----- // uncomment below for min cost max flow -----------//7d2
----- // if (e.cap <= e.flow) continue; ------------------//7d3
----- int v = e.v; ---------------------------------------//7d4
----- long long w = vis[u] >= n ? 0LL : e.cost; ----------//7d5
----- if (dist[u] + w < dist[v]) { -----------------------//7d6
------- dist[v] = dist[u] + w; ---------------------------//7d7
------- if (!inq[v]) { -----------------------------------//7d8
--------- inq[v] = true; ---------------------------------//7d9
--------- q.push(v); -------------------------------------//7da
------- }}}}} --------------------------------------------//7db
```

3.2. All-Pairs Shortest Paths.

3.2.1. *Floyd-Washall.*

```
#include "graph_template_adjmat.cpp" ---------------------//7ba
// insert inside graph; needs n and mat[][] --------------//7bb
void floyd_warshall() { ----------------------------------//7bc
- for (int k = 0; k < n; ++k) ----------------------------//7bd
--- for (int i = 0; i < n; ++i) --------------------------//7be
----- for (int j = 0; j < n; ++j) ------------------------//7bf
------- if (mat[i][k] + mat[k][j] < mat[i][j]) -----------//7c0
--------- mat[i][j] = mat[i][k] + mat[k][j]; -------------//7c1
} --------------------------------------------------------//7c2
```

3.3. Strongly Connected Components.

3.3.1. *Kosaraju.*

```
struct kosaraju_graph { ----------------------------------//74e
- int n; -------------------------------------------------//74f
- int *vis; ----------------------------------------------//750
- vi **adj; ----------------------------------------------//751
- std::vector<vi> sccs; ----------------------------------//752
- kosaraju_graph(int n) { --------------------------------//753
--- this->n = n; -----------------------------------------//754
--- vis = new int[n]; ------------------------------------//755
--- adj = new vi*[2]; ------------------------------------//756
--- for (int dir = 0; dir < 2; ++dir) --------------------//757
----- adj[dir] = new vi[n]; ------------------------------//758
- } ------------------------------------------------------//759
- void add_edge(int u, int v) { --------------------------//75a
--- adj[0][u].push_back(v); ------------------------------//75b
--- adj[1][v].push_back(u); ------------------------------//75c
- } ------------------------------------------------------//75d
- void dfs(int u, int p, int dir, vi &topo) { ------------//75e
--- vis[u] = 1; ------------------------------------------//75f
--- for (int v : adj[dir][u]) ----------------------------//760
----- if (!vis[v] && v != p) -----------------------------//761
------- dfs(v, u, dir, topo); ----------------------------//762
--- topo.push_back(u); -----------------------------------//763
- } ------------------------------------------------------//764
- void kosaraju() { --------------------------------------//765
--- vi topo; ---------------------------------------------//766
--- for (int u = 0; u < n; ++u)  vis[u] = 0; -------------//767
--- for (int u = 0; u < n; ++u) --------------------------//768
----- if (!vis[u]) ---------------------------------------//769
------- dfs(u, -1, 0, topo); -----------------------------//76a
--- for (int u = 0; u < n; ++u)  vis[u] = 0; -------------//76b
--- for (int i = n-1; i >= 0; --i) { ---------------------//76c
----- if (!vis[topo[i]]) { -------------------------------//76d
------- sccs.push_back({}); ------------------------------//76e
------- dfs(topo[i], -1, 1, sccs.back()); ----------------//76f
----- } --------------------------------------------------//770
--- } ----------------------------------------------------//771
- } ------------------------------------------------------//772
}; -------------------------------------------------------//773
```

3.3.2. *Tarjan's Offline Algorithm.*

```
int n, id[N], low[N], st[N], in[N], TOP, ID; -------------//774
int scc[N], SCC_SIZE; // 0 <= scc[u] < SCC_SIZE ----------//775
vector<int> adj[N]; // 0-based adjlist -------------------//776
void dfs(int u) { ----------------------------------------//777
--- id[u] = low[u] = ID++; -------------------------------//778
--- st[TOP++] = u; in[u] = 1; ----------------------------//779
--- for (int v : adj[u]) { -------------------------------//77a
------- if (id[v] == -1) { -------------------------------//77b
----------- dfs(v); --------------------------------------//77c
----------- low[u] = min(low[u], low[v]); ----------------//77d
------- } else if (in[v] == 1) ---------------------------//77e
----------- low[u] = min(low[u], id[v]); -----------------//77f
--- } ----------------------------------------------------//780
--- if (id[u] == low[u]) { -------------------------------//781
------- int sid = SCC_SIZE++; ----------------------------//782
------- do { ---------------------------------------------//783
----------- int v = st[--TOP]; ---------------------------//784
----------- in[v] = 0; scc[v] = sid; ---------------------//785
------- } while (st[TOP] != u); --------------------------//786
--- }} ---------------------------------------------------//787
void tarjan() { // call tarjan() to load SCC -------------//788
--- memset(id, -1, sizeof(int) * n); ---------------------//789
--- SCC_SIZE = ID = TOP = 0; -----------------------------//78a
--- for (int i = 0; i < n; ++i) --------------------------//78b
------- if (id[i] == -1) dfs(i); } -----------------------//78c
```

3.4. Minimum Mean Weight Cycle.

Run this for each strongly connected component

```
double min_mean_cycle(vector<vector<pair<int,double>>> adj){
- int n = size(adj); double mn = INFINITY; ---------------//5b2
- vector<vector<double> > arr(n+1, vector<double>(n, mn));//5b3
- arr[0][0] = 0; -----------------------------------------//5b4
- rep(k,1,n+1) rep(j,0,n) iter(it,adj[j]) ----------------//5b5
--- arr[k][it->first] = min(arr[k][it->first], -----------//5b6
-------------------------- it->second + arr[k-1][j]); ---//5b7
- rep(k,0,n) { -------------------------------------------//5b8
--- double mx = -INFINITY; -------------------------------//5b9
--- rep(i,0,n) mx = max(mx, (arr[n][i]-arr[k][i])/(n-k)); //5ba
--- mn = min(mn, mx); } ----------------------------------//5bb
- return mn; } -------------------------------------------//5bc
```

3.5. Cut Points and Bridges.

```
vii bridges; ---------------------------------------------//4d4
vi adj[MAXN], disc, low, articulation_points; ------------//4d5
int TIME; ------------------------------------------------//4d6
void bridges_artics (int u, int p) { ---------------------//4d7
- disc[u] = low[u] = TIME++; -----------------------------//4d8
- int children = 0; --------------------------------------//4d9
- bool has_low_child = false; ----------------------------//4da
- for (int v : adj[u]) { ---------------------------------//4db
--- if (disc[v] == -1) { ---------------------------------//4dc
----- bridges_artics(v, u); ------------------------------//4dd
----- children++; ----------------------------------------//4de
----- if (disc[u] < low[v]) ------------------------------//4df
------- bridges.push_back({u, v}); -----------------------//4e0
----- if (disc[u] <= low[v]) -----------------------------//4e1
------- has_low_child = true; ----------------------------//4e2
----- low[u] = min(low[u], low[v]); ----------------------//4e3
--- } else if (v != p) -----------------------------------//4e4
----- low[u] = min(low[u], disc[v]); ---------------------//4e5
- } ------------------------------------------------------//4e6
- if ((p == -1 && children >= 2) || ----------------------//4e7
----- (p != -1 && has_low_child)) ------------------------//4e8
--- articulation_points.push_back(u); --------------------//4e9
} --------------------------------------------------------//4ea
```

3.6. Biconnected Components.

3.6.1. *Bridge Tree.*

3.6.2. *Block-Cut Tree.*

3.7. Minimum Spanning Tree.

3.7.1. *Kruskal.*

```
#include "graph_template_edgelist.cpp" -------------------//72c
#include "union_find.cpp" --------------------------------//72d
// insert inside graph; needs n, and edges ---------------//72e
void kruskal(viii &res) { --------------------------------//72f
- viii().swap(res); // or use res.clear(); ---------------//730
- std::priority_queue<iii, viii, std::greater<iii> > pq; -//731
- for (auto &edge : edges) -------------------------------//732
--- pq.push(edge); ---------------------------------------//733
- union_find uf(n); --------------------------------------//734
- while (!pq.empty()) { ----------------------------------//735
--- auto node = pq.top();   pq.pop(); --------------------//736
--- int u = node.second.first; ---------------------------//737
--- int v = node.second.second; --------------------------//738
--- if (uf.unite(u, v)) ----------------------------------//739
----- res.push_back(node); -------------------------------//73a
```

```
- } ------------------------------------------------------//73b
} --------------------------------------------------------//73c
```

3.7.2. *Prim.*

```
#include "graph_template_adjlist.cpp" -------------------//73d
// insert inside graph; needs n, vis[], and adj[] --------//73e
void prim(viii &res, int s=0) { -------------------------//73f
- viii().swap(res); // or use res.clear(); ---------------//740
- std::priority_queue<ii, vii, std::greater<ii> > pq; ----//741
- pq.push({0, s}); ---------------------------------------//742
- while (!pq.empty()) { ----------------------------------//743
--- int u = pq.top().second;    pq.pop(); ---------------//744
--- vis[u] = true; ---------------------------------------//745
--- for (auto &[v, w] : adj[u]) { ------------------------//746
----- if (v == u)   continue; ---------------------------//747
----- if (vis[v])   continue; ---------------------------//748
----- res.push_back({w, {u, v}}); -----------------------//749
----- pq.push({w, v}); ----------------------------------//74a
--- } ----------------------------------------------------//74b
- } ------------------------------------------------------//74c
} --------------------------------------------------------//74d
```

3.8. **Euler Path/Cycle.**

3.8.1. *Euler Path/Cycle in a Directed Graph.*

```
#define MAXV 1000 ----------------------------------------//514
#define MAXE 5000 ----------------------------------------//515
vi adj[MAXV]; --------------------------------------------//516
int n, m, indeg[MAXV], outdeg[MAXV], res[MAXE + 1]; ------//517
ii start_end() { -----------------------------------------//518
- int start = -1, end = -1, any = 0, c = 0; --------------//519
- rep(i,0,n) { -------------------------------------------//51a
--- if (outdeg[i] > 0) any = i; --------------------------//51b
--- if (indeg[i] + 1 == outdeg[i]) start = i, c++; -------//51c
--- else if (indeg[i] == outdeg[i] + 1) end = i, c++; ----//51d
--- else if (indeg[i] != outdeg[i]) return ii(-1,-1); } --//51e
- if ((start == -1) != (end == -1) || (c != 2 && c != 0)) //51f
--- return ii(-1,-1); ------------------------------------//520
- if (start == -1) start = end = any; --------------------//521
- return ii(start, end); } -------------------------------//522
bool euler_path() { --------------------------------------//523
- ii se = start_end(); -----------------------------------//524
- int cur = se.first, at = m + 1; ------------------------//525
- if (cur == -1) return false; ---------------------------//526
- stack<int> s; ------------------------------------------//527
- while (true) { -----------------------------------------//528
--- if (outdeg[cur] == 0) { ------------------------------//529
----- res[--at] = cur; ----------------------------------//52a
----- if (s.empty()) break; -----------------------------//52b
----- cur = s.top(); s.pop(); ---------------------------//52c
--- } else s.push(cur), cur = adj[cur][--outdeg[cur]]; } -//52d
- return at == 0; } --------------------------------------//52e
```

3.8.2. (. Euler Path/Cycle in an Undirected Graph)

```
multiset<int> adj[1010]; ---------------------------------//52f
list<int> L; ---------------------------------------------//530
list<int>::iterator euler(int at, int to, ----------------//531
--- list<int>::iterator it) { ----------------------------//532
- if (at == to) return it; -------------------------------//533
- L.insert(it, at), --it; --------------------------------//534
- while (!adj[at].empty()) { -----------------------------//535
--- int nxt = *adj[at].begin(); --------------------------//536
--- adj[at].erase(adj[at].find(nxt)); --------------------//537
--- adj[nxt].erase(adj[nxt].find(at)); -------------------//538
--- if (to == -1) { --------------------------------------//539
----- it = euler(nxt, at, it); --------------------------//53a
----- L.insert(it, at); ---------------------------------//53b
----- --it; ---------------------------------------------//53c
--- } else { ---------------------------------------------//53d
----- it = euler(nxt, to, it); --------------------------//53e
----- to = -1; } } --------------------------------------//53f
- return it; } -------------------------------------------//540
// euler(0,-1,L.begin()) ---------------------------------//541
```

3.9. **Bipartite Matching.**

3.9.1. *Alternating Paths Algorithm.*

```
vi* adj; -------------------------------------------------//5ee
bool* done; ----------------------------------------------//5ef
int* owner; ----------------------------------------------//5f0
int alternating_path(int left) { -------------------------//5f1
- if (done[left]) return 0; ------------------------------//5f2
- done[left] = true; -------------------------------------//5f3
- rep(i,0,size(adj[left])) { -----------------------------//5f4
--- int right = adj[left][i]; ----------------------------//5f5
--- if (owner[right] == -1 || ----------------------------//5f6
------- alternating_path(owner[right])) { ---------------//5f7
----- owner[right] = left; return 1; } } ----------------//5f8
- return 0; } --------------------------------------------//5f9
```

3.9.2. *Hopcroft-Karp Algorithm.*

```
#define MAXN 5000 ----------------------------------------//609
int dist[MAXN+1], q[MAXN+1]; -----------------------------//60a
#define dist(v) dist[v == -1 ? MAXN : v] -----------------//60b
struct bipartite_graph { ---------------------------------//60c
- int N, M, *L, *R; vi *adj; -----------------------------//60d
- bipartite_graph(int _N, int _M) : N(_N), M(_M), --------//60e
--- L(new int[N]), R(new int[M]), adj(new vi[N]) {} ------//60f
- ~bipartite_graph() { delete[] adj; delete[] L; delete[] R; }
- bool bfs() { -------------------------------------------//611
--- int l = 0, r = 0; ------------------------------------//612
--- rep(v,0,N) if(L[v] == -1) dist(v) = 0, q[r++] = v; ---//613
----- else dist(v) = INF; -------------------------------//614
--- dist(-1) = INF; -------------------------------------//615
--- while(l < r) { ---------------------------------------//616
----- int v = q[l++]; -----------------------------------//617
----- if(dist(v) < dist(-1)) { --------------------------//618
------- iter(u, adj[v]) if(dist(R[*u]) == INF) ----------//619
--------- dist(R[*u]) = dist(v) + 1, q[r++] = R[*u]; } } -//61a
--- return dist(-1) != INF; } ----------------------------//61b
- bool dfs(int v) { --------------------------------------//61c
--- if(v != -1) { ----------------------------------------//61d
----- iter(u, adj[v]) -----------------------------------//61e
------- if(dist(R[*u]) == dist(v) + 1) ------------------//61f
--------- if(dfs(R[*u])) { ------------------------------//620
----------- R[*u] = v, L[v] = *u; -----------------------//621
----------- return true; } ------------------------------//622
----- dist(v) = INF; ------------------------------------//623
----- return false; } -----------------------------------//624
--- return true; } ---------------------------------------//625
- void add_edge(int i, int j) { adj[i].push_back(j); } ---//626
- int maximum_matching() { -------------------------------//627
--- int matching = 0; ------------------------------------//628
--- memset(L, -1, sizeof(int) * N); ----------------------//629
--- memset(R, -1, sizeof(int) * M); ----------------------//62a
--- while(bfs()) rep(i,0,N) ------------------------------//62b
----- matching += L[i] == -1 && dfs(i); -----------------//62c
--- return matching; } }; --------------------------------//62d
```

3.9.3. *Minimum Vertex Cover in Bipartite Graphs.*

```
#include "hopcroft_karp.cpp" -----------------------------//5fa
vector<bool> alt; ----------------------------------------//5fb
void dfs(bipartite_graph &g, int at) { -------------------//5fc
- alt[at] = true; ----------------------------------------//5fd
- iter(it,g.adj[at]) { -----------------------------------//5fe
--- alt[*it + g.N] = true; -------------------------------//5ff
--- if (g.R[*it] != -1 && !alt[g.R[*it]]) ----------------//600
----- dfs(g, g.R[*it]); } } -----------------------------//601
vi mvc_bipartite(bipartite_graph &g) { -------------------//602
- vi res; g.maximum_matching(); --------------------------//603
- alt.assign(g.N + g.M,false); ---------------------------//604
- rep(i,0,g.N) if (g.L[i] == -1) dfs(g, i); --------------//605
- rep(i,0,g.N) if (!alt[i]) res.push_back(i); ------------//606
- rep(i,0,g.M) if (alt[g.N + i]) res.push_back(g.N + i); -//607
- return res; } ------------------------------------------//608
```

3.10. **Maximum Flow.**

3.10.1. *Edmonds-Karp.*

```
struct flow_network { ------------------------------------//6b0
- int n, s, t, *par, **c, **f; ---------------------------//6b1
- vi *adj; -----------------------------------------------//6b2
- flow_network(int n, int s, int t) : n(n), s(s), t(t) { -//6b3
--- adj = new std::vector<int>[n]; -----------------------//6b4
--- par = new int[n]; ------------------------------------//6b5
--- c = new int*[n]; -------------------------------------//6b6
--- f = new int*[n]; -------------------------------------//6b7
--- for (int i = 0; i < n; ++i) { ------------------------//6b8
----- c[i] = new int[n]; --------------------------------//6b9
----- f[i] = new int[n]; --------------------------------//6ba
----- for (int j = 0; j < n; ++j) -----------------------//6bb
------- c[i][j] = f[i][j] = 0; --------------------------//6bc
--- } ----------------------------------------------------//6bd
- } ------------------------------------------------------//6be
- void add_edge(int u, int v, int w) { -------------------//6bf
--- adj[u].push_back(v); ---------------------------------//6c0
--- adj[v].push_back(u); ---------------------------------//6c1
--- c[u][v] += w; ----------------------------------------//6c2
- } ------------------------------------------------------//6c3
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//6c4
- bool bfs() { -------------------------------------------//6c5
--- std::queue<int> q; -----------------------------------//6c6
```

```
--- q.push(this->s); ------------------------------------//6c7
--- while (!q.empty()) { --------------------------------//6c8
----- int u = q.front();  q.pop(); ----------------------//6c9
----- for (int v : adj[u]) { ----------------------------//6ca
------- if (res(u, v) > 0 and par[v] == -1) { -----------//6cb
--------- par[v] = u; -----------------------------------//6cc
--------- if (v == this->t) -----------------------------//6cd
----------- return true; --------------------------------//6ce
--------- q.push(v); ------------------------------------//6cf
------- } -----------------------------------------------//6d0
----- } -------------------------------------------------//6d1
--- } ---------------------------------------------------//6d2
--- return false; ---------------------------------------//6d3
- } -----------------------------------------------------//6d4
- bool aug_path() { -------------------------------------//6d5
--- for (int u = 0; u < n; ++u) -------------------------//6d6
----- par[u] = -1; --------------------------------------//6d7
--- par[s] = s; -----------------------------------------//6d8
--- return bfs(); ---------------------------------------//6d9
- } -----------------------------------------------------//6da
- int calc_max_flow() { ---------------------------------//6db
--- int ans = 0; ----------------------------------------//6dc
--- while (aug_path()) { --------------------------------//6dd
----- int flow = INF; -----------------------------------//6de
----- for (int u = t; u != s; u = par[u]) ---------------//6df
------- flow = std::min(flow, res(par[u], u)); ----------//6e0
----- for (int u = t; u != s; u = par[u]) ---------------//6e1
------- f[par[u]][u] += flow, f[u][par[u]] -= flow; -----//6e2
----- ans += flow; --------------------------------------//6e3
--- } ---------------------------------------------------//6e4
--- return ans; -----------------------------------------//6e5
- } -----------------------------------------------------//6e6
}; ------------------------------------------------------//6e7
```

3.10.2. *Dinic.*

```
struct flow_network { -----------------------------------//662
- int n, s, t, *adj_ptr, *dist, *par, **c, **f; ---------//663
- vi *adj; ----------------------------------------------//664
- flow_network(int n, int s, int t) : n(n), s(s), t(t) { -//665
--- adj = new std::vector<int>[n]; ----------------------//666
--- adj_ptr = new int[n]; -------------------------------//667
--- dist = new int[n]; ----------------------------------//668
--- par = new int[n]; -----------------------------------//669
--- c = new int*[n]; ------------------------------------//66a
--- f = new int*[n]; ------------------------------------//66b
--- for (int i = 0; i < n; ++i) { -----------------------//66c
----- c[i] = new int[n]; --------------------------------//66d
----- f[i] = new int[n]; --------------------------------//66e
----- for (int j = 0; j < n; ++j) -----------------------//66f
------- c[i][j] = f[i][j] = 0; --------------------------//670
--- } ---------------------------------------------------//671
- } -----------------------------------------------------//672
- void add_edge(int u, int v, int w) { ------------------//673
--- adj[u].push_back(v); --------------------------------//674
--- adj[v].push_back(u); --------------------------------//675
--- c[u][v] += w; ---------------------------------------//676
- } -----------------------------------------------------//677
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//678
- void reset(int *ar, int val) { ------------------------//679
--- for (int i = 0; i < n; ++i) -------------------------//67a
----- ar[i] = val; --------------------------------------//67b
- } -----------------------------------------------------//67c
- bool make_level_graph() { -----------------------------//67d
--- reset(dist, -1); ------------------------------------//67e
--- std::queue<int> q; ----------------------------------//67f
--- q.push(s); ------------------------------------------//680
--- dist[s] = 0; ----------------------------------------//681
--- while (!q.empty()) { --------------------------------//682
----- int u = q.front();  q.pop(); ----------------------//683
----- for (int v : adj[u]) { ----------------------------//684
------- if (res(u, v) > 0 and dist[v] == -1) { ----------//685
--------- dist[v] = dist[u] + 1; ------------------------//686
--------- q.push(v); ------------------------------------//687
------- } -----------------------------------------------//688
----- } -------------------------------------------------//689
--- } ---------------------------------------------------//68a
--- return dist[t] != -1; -------------------------------//68b
- } -----------------------------------------------------//68c
- bool next(int u, int v) { -----------------------------//68d
--- return dist[v] == dist[u] + 1; ----------------------//68e
- } -----------------------------------------------------//68f
- bool dfs(int u) { -------------------------------------//690
--- if (u == t)   return true; --------------------------//691
--- for (int &i = adj_ptr[u]; i < adj[u].size(); ++i) { --//692
----- int v = adj[u][i]; --------------------------------//693
----- if (next(u, v) and res(u, v) > 0 and dfs(v)) { ----//694
------- par[v] = u; -------------------------------------//695
------- return true; ------------------------------------//696
----- } -------------------------------------------------//697
--- } ---------------------------------------------------//698
--- dist[u] = -1; ---------------------------------------//699
--- return false; ---------------------------------------//69a
- } -----------------------------------------------------//69b
- bool aug_path() { -------------------------------------//69c
--- reset(par, -1); -------------------------------------//69d
--- par[s] = s; -----------------------------------------//69e
--- return dfs(s);  } -----------------------------------//69f
- int calc_max_flow() { ---------------------------------//6a0
--- int ans = 0; ----------------------------------------//6a1
--- while (make_level_graph()) { ------------------------//6a2
----- reset(adj_ptr, 0); --------------------------------//6a3
----- while (aug_path()) { ------------------------------//6a4
------- int flow = INF; ---------------------------------//6a5
------- for (int u = t; u != s; u = par[u]) -------------//6a6
--------- flow = std::min(flow, res(par[u], u)); --------//6a7
------- for (int u = t; u != s; u = par[u]) -------------//6a8
--------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ---//6a9
------- ans += flow; ------------------------------------//6aa
----- } -------------------------------------------------//6ab
--- } ---------------------------------------------------//6ac
--- return ans; -----------------------------------------//6ad
- } -----------------------------------------------------//6ae
}; ------------------------------------------------------//6af
```

3.11. **All-pairs Maximum Flow.**

3.11.1. *Gomory-Hu.*

```
#define MAXV 2000 ------------------------------------//6e8
int q[MAXV], d[MAXV]; -----------------------------------//6e9
struct flow_network { -----------------------------------//6ea
- struct edge { int v, nxt, cap; ------------------------//6eb
--- edge(int _v, int _cap, int _nxt) --------------------//6ec
----- : v(_v), nxt(_nxt), cap(_cap) { } }; --------------//6ed
- int n, *head, *curh; vector<edge> e, e_store; ---------//6ee
- flow_network(int _n) : n(_n) { ------------------------//6ef
--- curh = new int[n]; ----------------------------------//6f0
--- memset(head = new int[n], -1, n*sizeof(int)); } -----//6f1
- void reset() { e = e_store; } -------------------------//6f2
- void add_edge(int u, int v, int uv, int vu=0) { -------//6f3
--- e.push_back(edge(v,uv,head[u])); head[u]=(int)size(e)-1;
--- e.push_back(edge(u,vu,head[v])); head[v]=(int)size(e)-1;}
- int augment(int v, int t, int f) { --------------------//6f6
--- if (v == t) return f; -------------------------------//6f7
--- for (int &i = curh[v], ret; i != -1; i = e[i].nxt) ---//6f8
----- if (e[i].cap > 0 && d[e[i].v] + 1 == d[v]) --------//6f9
------- if ((ret = augment(e[i].v, t, min(f, e[i].cap))) > 0)
--------- return (e[i].cap -= ret, e[i^1].cap += ret, ret);
--- return 0; } -----------------------------------------//6fc
- int max_flow(int s, int t, bool res=true) { -----------//6fd
--- e_store = e; ----------------------------------------//6fe
--- int l, r, f = 0, x; ---------------------------------//6ff
--- while (true) { --------------------------------------//700
----- memset(d, -1, n*sizeof(int)); ---------------------//701
----- l = r = 0, d[q[r++] = t] = 0; ---------------------//702
----- while (l < r) -------------------------------------//703
------- for (int v = q[l++], i = head[v]; i != -1; i=e[i].nxt)
--------- if (e[i^1].cap > 0 && d[e[i].v] == -1) --------//705
----------- d[q[r++] = e[i].v] = d[v]+1; ----------------//706
----- if (d[s] == -1) break; ----------------------------//707
----- memcpy(curh, head, n * sizeof(int)); --------------//708
----- while ((x = augment(s, t, INF)) != 0) f += x; } ---//709
--- if (res) reset(); -----------------------------------//70a
--- return f; } }; --------------------------------------//70b
bool same[MAXV]; ----------------------------------------//70c
pair<vii, vvi> construct_gh_tree(flow_network &g) { -----//70d
- int n = g.n, v; ---------------------------------------//70e
- vii par(n, ii(0, 0)); vvi cap(n, vi(n, -1)); ----------//70f
- rep(s,1,n) { ------------------------------------------//710
--- int l = 0, r = 0; -----------------------------------//711
--- par[s].second = g.max_flow(s, par[s].first, false); --//712
--- memset(d, 0, n * sizeof(int)); ----------------------//713
--- memset(same, 0, n * sizeof(bool)); ------------------//714
--- d[q[r++] = s] = 1; ----------------------------------//715
--- while (l < r) { -------------------------------------//716
----- same[v = q[l++]] = true; --------------------------//717
----- for (int i = g.head[v]; i != -1; i = g.e[i].nxt) ---//718
------- if (g.e[i].cap > 0 && d[g.e[i].v] == 0) ---------//719
--------- d[q[r++] = g.e[i].v] = 1; } -------------------//71a
```

```
--- rep(i,s+1,n) -------------------------------------------//71b
----- if (par[i].first == par[s].first && same[i]) -------//71c
------- par[i].first = s; ---------------------------------//71d
--- g.reset(); } -------------------------------------------//71e
- rep(i,0,n) { ---------------------------------------------//71f
--- int mn = INF, cur = i; ---------------------------------//720
--- while (true) { -----------------------------------------//721
----- cap[cur][i] = mn; ------------------------------------//722
----- if (cur == 0) break; ---------------------------------//723
----- mn = min(mn, par[cur].second), cur = par[cur].first; } }
- return make_pair(par, cap); } ----------------------------//725
int compute_max_flow(int s, int t, const pair<vii, vvi> &gh) {
- int cur = INF, at = s; ------------------------------------//727
- while (gh.second[at][t] == -1) ---------------------------//728
--- cur = min(cur, gh.first[at].second), -----------------//729
--- at = gh.first[at].first; -------------------------------//72a
- return min(cur, gh.second[at][t]); } --------------------//72b
```

3.12. **Minimum Arborescence.** Given a weighted directed graph, finds a subset of edges of minimum total weight so that there is a unique path from the root r to each vertex. Returns a vector of size n, where the ith element is the edge for the ith vertex. The answer for the root is undefined!

```
#include "../data-structures/union_find.cpp" -------------//45b
struct arborescence { --------------------------------------//45c
- int n; union_find uf; ------------------------------------//45d
- vector<vector<pair<ii,int> > > adj; ---------------------//45e
- arborescence(int _n) : n(_n), uf(n), adj(n) { } --------//45f
- void add_edge(int a, int b, int c) { --------------------//460
--- adj[b].push_back(make_pair(ii(a,b),c)); } ------------//461
- vii find_min(int r) { ------------------------------------//462
--- vi vis(n,-1), mn(n,INF); vii par(n); -----------------//463
--- rep(i,0,n) { -------------------------------------------//464
----- if (uf.find(i) != i) continue; ----------------------//465
----- int at = i; ------------------------------------------//466
----- while (at != r && vis[at] == -1) { ------------------//467
------- vis[at] = i; ---------------------------------------//468
------- iter(it,adj[at]) if (it->second < mn[at] && ------//469
----------- uf.find(it->first.first) != at) --------------//46a
--------- mn[at] = it->second, par[at] = it->first; ------//46b
------- if (par[at] == ii(0,0)) return vii(); ------------//46c
------- at = uf.find(par[at].first); } --------------------//46d
----- if (at == r || vis[at] != i) continue; -------------//46e
----- union_find tmp = uf; vi seq; ------------------------//46f
----- do { seq.push_back(at); at = uf.find(par[at].first);//470
----- } while (at != seq.front()); ------------------------//471
----- iter(it,seq) uf.unite(*it,seq[0]); ------------------//472
----- int c = uf.find(seq[0]); -----------------------------//473
----- vector<pair<ii,int> > nw; ----------------------------//474
----- iter(it,seq) iter(jt,adj[*it]) ----------------------//475
------- nw.push_back(make_pair(jt->first, ----------------//476
------------- jt->second - mn[*it])); ---------------------//477
----- adj[c] = nw; -----------------------------------------//478
----- vii rest = find_min(r); ------------------------------//479
----- if (size(rest) == 0) return rest; -------------------//47a
----- ii use = rest[c]; ------------------------------------//47b
----- rest[at = tmp.find(use.second)] = use; --------------//47c
----- iter(it,seq) if (*it != at) -------------------------//47d
------- rest[*it] = par[*it]; -----------------------------//47e
----- return rest; } ---------------------------------------//47f
--- return par; } }; ---------------------------------------//480
```

3.13. **Blossom algorithm.** Finds a maximum matching in an arbitrary graph in $O(|V|^4)$ time. Be vary of loop edges.

```
#define MAXV 300 -------------------------------------------//481
bool marked[MAXV], emarked[MAXV][MAXV]; -------------------//482
int S[MAXV]; -----------------------------------------------//483
vi find_augmenting_path(const vector<vi> &adj,const vi &m){
- int n = size(adj), s = 0; --------------------------------//485
- vi par(n,-1), height(n), root(n,-1), q, a, b; ----------//486
- memset(marked,0,sizeof(marked)); ------------------------//487
- memset(emarked,0,sizeof(emarked)); ----------------------//488
- rep(i,0,n) if (m[i] >= 0) emarked[i][m[i]] = true; -----//489
----------- else root[i] = i, S[s++] = i; -----------------//48a
- while (s) { ----------------------------------------------//48b
--- int v = S[--s]; ----------------------------------------//48c
--- iter(wt,adj[v]) { --------------------------------------//48d
----- int w = *wt; -----------------------------------------//48e
----- if (emarked[v][w]) continue; ------------------------//48f
----- if (root[w] == -1) { ---------------------------------//490
------- int x = S[s++] = m[w]; -----------------------------//491
------- par[w]=v, root[w]=root[v], height[w]=height[v]+1; //492
------- par[x]=w, root[x]=root[w], height[x]=height[w]+1; //493
----- } else if (height[w] % 2 == 0) { --------------------//494
------- if (root[v] != root[w]) { -------------------------//495
--------- while (v != -1) q.push_back(v), v = par[v]; ----//496
--------- reverse(q.begin(), q.end()); --------------------//497
--------- while (w != -1) q.push_back(w), w = par[w]; ----//498
--------- return q; ----------------------------------------//499
------- } else { -------------------------------------------//49a
--------- int c = v; ---------------------------------------//49b
--------- while (c != -1) a.push_back(c), c = par[c]; ----//49c
--------- c = w; -------------------------------------------//49d
--------- while (c != -1) b.push_back(c), c = par[c]; ----//49e
--------- while (!a.empty()&&!b.empty()&&a.back()==b.back())
----------- c = a.back(), a.pop_back(), b.pop_back(); ----//4a0
--------- memset(marked,0,sizeof(marked)); ----------------//4a1
--------- fill(par.begin(), par.end(), 0); ----------------//4a2
--------- iter(it,a) par[*it] = 1; iter(it,b) par[*it] = 1;
--------- par[c] = s = 1; ----------------------------------//4a4
--------- rep(i,0,n) root[par[i] = par[i] ? 0 : s++] = i; //4a5
--------- vector<vi> adj2(s); ------------------------------//4a6
--------- rep(i,0,n) iter(it,adj[i]) { --------------------//4a7
----------- if (par[*it] == 0) continue; ------------------//4a8
----------- if (par[i] == 0) { -----------------------------//4a9
------------- if (!marked[par[*it]])  { -------------------//4aa
--------------- adj2[par[i]].push_back(par[*it]); --------//4ab
--------------- adj2[par[*it]].push_back(par[i]); --------//4ac
--------------- marked[par[*it]] = true; } ----------------//4ad
----------- } else adj2[par[i]].push_back(par[*it]); } ---//4ae
--------- vi m2(s, -1); ------------------------------------//4af
--------- if (m[c] != -1) m2[m2[par[m[c]]] = 0] = par[m[c]];
--------- rep(i,0,n) if(par[i]!=0&&m[i]!=-1&&par[m[i]]!=0)//4b1
----------- m2[par[i]] = par[m[i]]; -----------------------//4b2
--------- vi p = find_augmenting_path(adj2, m2); ---------//4b3
--------- int t = 0; ---------------------------------------//4b4
--------- while (t < size(p) && p[t]) t++; ----------------//4b5
--------- if (t == size(p)) { ------------------------------//4b6
----------- rep(i,0,size(p)) p[i] = root[p[i]]; ----------//4b7
----------- return p; } ------------------------------------//4b8
--------- if (!p[0] || (m[c] != -1 && p[t+1] != par[m[c]]))
----------- reverse(p.begin(), p.end()), t=(int)size(p)-t-1;
--------- rep(i,0,t) q.push_back(root[p[i]]); ------------//4bb
--------- iter(it,adj[root[p[t-1]]]) { --------------------//4bc
----------- if (par[*it] != (s = 0)) continue; -----------//4bd
----------- a.push_back(c), reverse(a.begin(), a.end()); -//4be
----------- iter(jt,b) a.push_back(*jt); ------------------//4bf
----------- while (a[s] != *it) s++; ----------------------//4c0
----------- if((height[*it]&1)^(s<(int)size(a)-(int)size(b)))
------------- reverse(a.begin(),a.end()), s=(int)size(a)-s-1;
----------- while(a[s]!=c)q.push_back(a[s]),s=(s+1)%size(a);
----------- q.push_back(c); --------------------------------//4c4
----------- rep(i,t+1,size(p)) q.push_back(root[p[i]]); --//4c5
----------- return q; } } } --------------------------------//4c6
----- emarked[v][w] = emarked[w][v] = true; } ------------//4c7
--- marked[v] = true; } return q; } -----------------------//4c8
vii max_matching(const vector<vi> &adj) { -----------------//4c9
- vi m(size(adj), -1), ap; vii res, es; -------------------//4ca
- rep(i,0,size(adj)) iter(it,adj[i]) es.emplace_back(i,*it);
- random_shuffle(es.begin(), es.end()); -------------------//4cc
- iter(it,es) if (m[it->first] == -1 && m[it->second] == -1)
--- m[it->first] = it->second, m[it->second] = it->first; //4ce
- do { ap = find_augmenting_path(adj, m); -----------------//4cf
------ rep(i,0,size(ap)) m[m[ap[i^1]] = ap[i]] = ap[i^1]; //4d0
- } while (!ap.empty()); -----------------------------------//4d1
- rep(i,0,size(m)) if (i < m[i]) res.emplace_back(i, m[i]);
- return res; } --------------------------------------------//4d3
```

3.14. **Maximum Density Subgraph.** Given (weighted) undirected graph G. Binary search density. If g is current density, construct flow network: (S,u,m), $(u,T,m+2g-d_u)$, $(u,v,1)$, where m is a large constant (larger than sum of edge weights). Run floating-point max-flow. If minimum cut has empty S-component, then maximum density is smaller than g, otherwise it's larger. Distance between valid densities is at least $1/(n(n-1))$. Edge case when density is 0. This also works for weighted graphs by replacing d_u by the weighted degree, and doing more iterations (if weights are not integers).

3.15. **Maximum-Weight Closure.** Given a vertex-weighted directed graph G. Turn the graph into a flow network, adding weight ∞ to each edge. Add vertices S,T. For each vertex v of weight w, add edge (S,v,w) if $w \geq 0$, or edge $(v,T,-w)$ if $w < 0$. Sum of positive weights minus minimum $S-T$ cut is the answer. Vertices reachable from S are in the closure. The maximum-weight closure is the same as the complement of the minimum-weight closure on the graph with edges reversed.

3.16. **Maximum Weighted Independent Set in a Bipartite Graph.** This is the same as the minimum weighted vertex cover. Solve this by constructing a flow network with edges $(S,u,w(u))$ for $u \in L$,

$(v, T, w(v))$ for $v \in R$ and (u, v, ∞) for $(u, v) \in E$. The minimum S, T-cut is the answer. Vertices adjacent to a cut edge are in the vertex cover.

3.17. **Synchronizing word problem.** A DFA has a synchronizing word (an input sequence that moves all states to the same state) iff. each pair of states has a synchronizing word. That can be checked using reverse DFS over pairs of states. Finding the shortest synchronizing word is NP-complete.

3.18. **Max flow with lower bounds on edges.** Change edge $(u, v, l \leq f \leq c)$ to $(u, v, f \leq c - l)$. Add edge (t, s, ∞). Create super-nodes S, T. Let $M(u) = \sum_v l(v, u) - \sum_v l(u, v)$. If $M(u) < 0$, add edge $(u, T, -M(u))$, else add edge $(S, u, M(u))$. Max flow from S to T. If all edges from S are saturated, then we have a feasible flow. Continue running max flow from s to t in original graph.

3.19. **Tutte matrix for general matching.** Create an $n \times n$ matrix A. For each edge (i, j), $i < j$, let $A_{ij} = x_{ij}$ and $A_{ji} = -x_{ij}$. All other entries are 0. The determinant of A is zero iff. the graph has a perfect matching. A randomized algorithm uses the Schwartz–Zippel lemma to check if it is zero.

3.20. **Heavy Light Decomposition.**

```
#include "segment_tree.cpp" ------------------------------//568
struct heavy_light_tree { --------------------------------//569
- int n; ------------------------------------------------//56a
- std::vector<int> *adj; --------------------------------//56b
- segtree *segment_tree; --------------------------------//56c
- int *par, *heavy, *dep, *path_root, *pos; -------------//56d
- heavy_light_tree(int n) { -----------------------------//56e
--- this->n = n; ----------------------------------------//56f
--- this->adj = new std::vector<int>[n]; ----------------//570
--- segment_tree = new segtree(0, n-1); -----------------//571
--- par = new int[n]; -----------------------------------//572
--- heavy = new int[n]; ---------------------------------//573
--- dep = new int[n]; -----------------------------------//574
--- path_root = new int[n]; -----------------------------//575
--- pos = new int[n]; -----------------------------------//576
- } -----------------------------------------------------//577
- void add_edge(int u, int v) { -------------------------//578
--- adj[u].push_back(v); --------------------------------//579
--- adj[v].push_back(u); --------------------------------//57a
- } -----------------------------------------------------//57b
- void build(int root) { --------------------------------//57c
--- for (int u = 0; u < n; ++u) -------------------------//57d
----- heavy[u] = -1; ------------------------------------//57e
--- par[root] = root; -----------------------------------//57f
--- dep[root] = 0; --------------------------------------//580
--- dfs(root); ------------------------------------------//581
--- for (int u = 0, p = 0; u < n; ++u) { ----------------//582
----- if (par[u] == -1 or heavy[par[u]] != u) { ---------//583
------- for (int v = u; v != -1; v = heavy[v]) { --------//584
--------- path_root[v] = u; -----------------------------//585
--------- pos[v] = p++; ---------------------------------//586
------- } -----------------------------------------------//587
----- } -------------------------------------------------//588
--- } ---------------------------------------------------//589
- } -----------------------------------------------------//58a
- int dfs(int u) { --------------------------------------//58b
--- int sz = 1; -----------------------------------------//58c
--- int max_subtree_sz = 0; -----------------------------//58d
--- for (int v : adj[u]) { ------------------------------//58e
----- if (v != par[u]) { --------------------------------//58f
------- par[v] = u; -------------------------------------//590
------- dep[v] = dep[u] + 1; ----------------------------//591
------- int subtree_sz = dfs(v); ------------------------//592
------- if (max_subtree_sz < subtree_sz) { --------------//593
--------- max_subtree_sz = subtree_sz; ------------------//594
--------- heavy[u] = v; ---------------------------------//595
------- } -----------------------------------------------//596
------- sz += subtree_sz; -------------------------------//597
----- } -------------------------------------------------//598
--- } ---------------------------------------------------//599
--- return sz; ------------------------------------------//59a
- } -----------------------------------------------------//59b
- int query(int u, int v) { -----------------------------//59c
--- int res = 0; ----------------------------------------//59d
--- while (path_root[u] != path_root[v]) { --------------//59e
----- if (dep[path_root[u]] > dep[path_root[v]]) --------//59f
------- std::swap(u, v); --------------------------------//5a0
----- res += segment_tree->sum(pos[path_root[v]], pos[v]);//5a1
----- v = par[path_root[v]]; ----------------------------//5a2
--- } ---------------------------------------------------//5a3
--- res += segment_tree->sum(pos[u], pos[v]); -----------//5a4
--- return res; -----------------------------------------//5a5
- } -----------------------------------------------------//5a6
- void update(int u, int v, int c) { --------------------//5a7
--- for (; path_root[u] != path_root[v]; ----------------//5a8
---------- v = par[path_root[v]]) { --------------------//5a9
----- if (dep[path_root[u]] > dep[path_root[v]]) --------//5aa
------- std::swap(u, v); --------------------------------//5ab
----- segment_tree->increase(pos[path_root[v]], pos[v], c);
--- } ---------------------------------------------------//5ad
--- segment_tree->increase(pos[u], pos[v], c); ----------//5ae
- } -----------------------------------------------------//5af
}; ------------------------------------------------------//5b0
```

3.21. **Centroid Decomposition.**

```
#define MAXV 100100 -------------------------------------//4eb
#define LGMAXV 20 ---------------------------------------//4ec
int jmp[MAXV][LGMAXV], ----------------------------------//4ed
- path[MAXV][LGMAXV], -----------------------------------//4ee
- sz[MAXV], seph[MAXV], ---------------------------------//4ef
- shortest[MAXV]; ---------------------------------------//4f0
struct centroid_decomposition { -------------------------//4f1
- int n; vvi adj; ---------------------------------------//4f2
- centroid_decomposition(int _n) : n(_n), adj(n) { } ----//4f3
- void add_edge(int a, int b) { -------------------------//4f4
--- adj[a].push_back(b); adj[b].push_back(a); } ---------//4f5
- int dfs(int u, int p) { -------------------------------//4f6
--- sz[u] = 1; ------------------------------------------//4f7
--- rep(i,0,size(adj[u])) -------------------------------//4f8
----- if (adj[u][i] != p) sz[u] += dfs(adj[u][i], u); ---//4f9
--- return sz[u]; } -------------------------------------//4fa
- void makepaths(int sep, int u, int p, int len) { ------//4fb
--- jmp[u][seph[sep]] = sep, path[u][seph[sep]] = len; --//4fc
--- int bad = -1; ---------------------------------------//4fd
--- rep(i,0,size(adj[u])) { -----------------------------//4fe
----- if (adj[u][i] == p) bad = i; ----------------------//4ff
----- else makepaths(sep, adj[u][i], u, len + 1); -------//500
--- } ---------------------------------------------------//501
--- if (p == sep) ---------------------------------------//502
----- swap(adj[u][bad], adj[u].back()), adj[u].pop_back(); }
- void separate(int h=0, int u=0) { ---------------------//504
--- dfs(u,-1); int sep = u; -----------------------------//505
--- down: iter(nxt,adj[sep]) ----------------------------//506
----- if (sz[*nxt] < sz[sep] && sz[*nxt] > sz[u]/2) { ---//507
------- sep = *nxt; goto down; } ------------------------//508
--- seph[sep] = h, makepaths(sep, sep, -1, 0); ----------//509
--- rep(i,0,size(adj[sep])) separate(h+1, adj[sep][i]); } //50a
- void paint(int u) { -----------------------------------//50b
--- rep(h,0,seph[u]+1) ----------------------------------//50c
----- shortest[jmp[u][h]] = min(shortest[jmp[u][h]], ----//50d
------------------------------- path[u][h]); } ----------//50e
- int closest(int u) { ----------------------------------//50f
--- int mn = INF/2; -------------------------------------//510
--- rep(h,0,seph[u]+1) ----------------------------------//511
----- mn = min(mn, path[u][h] + shortest[jmp[u][h]]); ---//512
--- return mn; } }; -------------------------------------//513
```

3.22. **Least Common Ancestor.**

3.22.1. *Binary Lifting.*

```
struct graph { ------------------------------------------//62e
- int n; ------------------------------------------------//62f
- int logn; ---------------------------------------------//630
- std::vector<int> *adj; --------------------------------//631
- int *dep; ---------------------------------------------//632
- int **par; --------------------------------------------//633
- graph(int n, int logn=20) { ---------------------------//634
--- this->n = n; ----------------------------------------//635
--- this->logn = logn; ----------------------------------//636
--- adj = new std::vector<int>[n]; ----------------------//637
--- dep = new int[n]; -----------------------------------//638
--- par = new int*[n]; ----------------------------------//639
--- for (int i = 0; i < n; ++i) -------------------------//63a
----- par[i] = new int[logn]; ---------------------------//63b
- } -----------------------------------------------------//63c
- void dfs(int u, int p, int d) { -----------------------//63d
--- dep[u] = d; -----------------------------------------//63e
--- par[u][0] = p; --------------------------------------//63f
--- for (int v : adj[u]) --------------------------------//640
----- if (v != p) ---------------------------------------//641
------- dfs(v, u, d+1); ---------------------------------//642
- } -----------------------------------------------------//643
- int ascend(int u, int k) { ----------------------------//644
--- for (int i = 0; i < logn; ++i) ----------------------//645
----- if (k & (1 << i)) ---------------------------------//646
------- u = par[u][i]; ----------------------------------//647
--- return u; -------------------------------------------//648
- } -----------------------------------------------------//649
```

```
- int lca(int u, int v) { --------------------------------//64a
--- if (dep[u] > dep[v])  u = ascend(u, dep[u] - dep[v]); //64b
--- if (dep[v] > dep[u])  v = ascend(v, dep[v] - dep[u]); //64c
--- if (u == v)           return u; ---------------------//64d
--- for (int k = logn-1; k >= 0; --k) { ------------------//64e
----- if (par[u][k] != par[v][k]) { ----------------------//64f
------- u = par[u][k]; -----------------------------------//650
------- v = par[v][k]; -----------------------------------//651
----- } --------------------------------------------------//652
--- } ----------------------------------------------------//653
--- return par[u][0]; ------------------------------------//654
- } ------------------------------------------------------//655
- bool is_anc(int u, int v) { ----------------------------//656
--- if (dep[u] < dep[v]) ---------------------------------//657
----- std::swap(u, v); -----------------------------------//658
--- return ascend(u, dep[u] - dep[v]) == v; --------------//659
- } ------------------------------------------------------//65a
- void prep_lca(int root=0) { ----------------------------//65b
--- dfs(root, root, 0); ----------------------------------//65c
--- for (int k = 1; k < logn; ++k) -----------------------//65d
----- for (int u = 0; u < n; ++u) ------------------------//65e
------- par[u][k] = par[par[u][k-1]][k-1]; ---------------//65f
- } ------------------------------------------------------//660
}; -------------------------------------------------------//661
```

3.23. **Counting Spanning Trees.** Kirchoff's Theorem: The number of spanning trees of any graph is the determinant of any cofactor of the Laplacian matrix in $O(n^3)$.

(1) Let A be the adjacency matrix.
(2) Let D be the degree matrix (matrix with vertex degrees on the diagonal).
(3) Get $D-A$ and delete exactly one row and column. Any row and column will do. This will be the cofactor matrix.
(4) Get the determinant of this cofactor matrix using Gauss-Jordan.
(5) Spanning Trees = $|\text{cofactor}(D-A)|$

3.24. **Erdős-Gallai Theorem.** A sequence of non-negative integers $d_1 \geq \cdots \geq d_n$ can be represented as the degree sequence of finite simple graph on n vertices if and only if $d_1 + \cdots + d_n$ is even and the following holds for $1 \leq k \leq n$:

$$\sum_{i=1}^{n} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k)$$

3.25. **Tree Isomorphism.**

```
// REQUIREMENT: list of primes pr[], see prime sieve -----//5bd
typedef long long LL; ------------------------------------//5be
int pre[N], q[N], path[N]; bool vis[N]; ------------------//5bf
// perform BFS and return the last node visited ----------//5c0
int bfs(int u, vector<int> adj[]) { ----------------------//5c1
--- memset(vis, 0, sizeof(vis)); -------------------------//5c2
--- int head = 0, tail = 0; ------------------------------//5c3
--- q[tail++] = u; vis[u] = true; pre[u] = -1; -----------//5c4
--- while (head != tail) { -------------------------------//5c5
------- u = q[head]; if (++head == N) head = 0; ----------//5c6
------- for (int i = 0; i < adj[u].size(); ++i) { --------//5c7
----------- int v = adj[u][i]; ---------------------------//5c8
----------- if (!vis[v]) { -------------------------------//5c9
--------------- vis[v] = true; pre[v] = u; ---------------//5ca
--------------- q[tail++] = v; if (tail == N) tail = 0; --//5cb
----------- }}} ------------------------------------------//5cc
--- return u; --------------------------------------------//5cd
} // returns the list of tree centers --------------------//5ce
vector<int> tree_centers(int r, vector<int> adj[]) { -----//5cf
--- int size = 0; ----------------------------------------//5d0
--- for (int u=bfs(bfs(r, adj), adj); u!=-1; u=pre[u]) ---//5d1
------- path[size++] = u; --------------------------------//5d2
--- vector<int> med(1, path[size/2]); --------------------//5d3
--- if (size % 2 == 0) med.push_back(path[size/2-1]); ----//5d4
--- return med; ------------------------------------------//5d5
} // returns "unique hashcode" for tree with root u ------//5d6
LL rootcode(int u, vector<int> adj[], int p=-1, int d=15){//5d7
--- vector<LL> k; int nd = (d + 1) % primes; -------------//5d8
--- for (int i = 0; i < adj[u].size(); ++i) --------------//5d9
------- if (adj[u][i] != p) ------------------------------//5da
----------- k.push_back(rootcode(adj[u][i], adj, u, nd)); //5db
--- sort(k.begin(), k.end()); ----------------------------//5dc
--- LL h = k.size() + 1; ---------------------------------//5dd
--- for (int i = 0; i < k.size(); ++i) -------------------//5de
------- h = h * pr[d] + k[i]; ----------------------------//5df
--- return h; --------------------------------------------//5e0
} // returns "unique hashcode" for the whole tree --------//5e1
LL treecode(int root, vector<int> adj[]) { ---------------//5e2
--- vector<int> c = tree_centers(root, adj); -------------//5e3
--- if (c.size()==1) -------------------------------------//5e4
------- return (rootcode(c[0], adj) << 1) | 1; -----------//5e5
--- return (rootcode(c[0],adj)*rootcode(c[1],adj))<<1; ---//5e6
} // checks if two trees are isomorphic ------------------//5e7
bool isomorphic(int r1, vector<int> adj1[], int r2, ------//5e8
--------------- vector<int> adj2[], bool rooted = false) {//5e9
--- if (rooted) ------------------------------------------//5ea
------- return rootcode(r1, adj1) == rootcode(r2, adj2); -//5eb
--- return treecode(r1, adj1) == treecode(r2, adj2); -----//5ec
} --------------------------------------------------------//5ed
```

4. Strings

4.1. **Knuth-Morris-Pratt.** Count and find all matches of string f in string s in $O(n)$ time.

```
int par[N]; // parent table ------------------------------//8db
void buildKMP(string& f) { -------------------------------//8dc
--- par[0] = -1, par[1] = 0; -----------------------------//8dd
--- int i = 2, j = 0; ------------------------------------//8de
--- while (i <= f.length()) { ----------------------------//8df
------- if (f[i-1] == f[j]) par[i++] = ++j; --------------//8e0
------- else if (j > 0) j = par[j]; ----------------------//8e1
------- else par[i++] = 0; }} ----------------------------//8e2
vector<int> KMP(string& s, string& f) { ------------------//8e3
--- buildKMP(f); // call once if f is the same -----------//8e4
--- int i = 0, j = 0; vector<int> ans; -------------------//8e5
--- while (i + j < s.length()) { -------------------------//8e6
------- if (s[i + j] == f[j]) { --------------------------//8e7
----------- if (++j == f.length()) { ---------------------//8e8
--------------- ans.push_back(i); ------------------------//8e9
--------------- i += j - par[j]; -------------------------//8ea
--------------- if (j > 0) j = par[j]; -------------------//8eb
----------- } --------------------------------------------//8ec
------- } else { -----------------------------------------//8ed
----------- i += j - par[j]; -----------------------------//8ee
----------- if (j > 0) j = par[j]; -----------------------//8ef
------- } ------------------------------------------------//8f0
--- } return ans; } --------------------------------------//8f1
```

4.2. **Suffix Array.** Construct a sorted catalog of all substrings of s in $O(n \log n)$ time using counting sort.

```
// sa[i]: ith smallest substring at s[sa[i]:] ------------//94c
// pos[i]: position of s[i:] in suffix array -------------//94d
int sa[N], pos[N], va[N], c[N], gap, n; ------------------//94e
bool cmp(int i, int j) // reverse stable sort ------------//94f
--- {return pos[i]!=pos[j] ? pos[i] < pos[j] : j < i;} ---//950
bool equal(int i, int j) ---------------------------------//951
--- {return pos[i] == pos[j] && i + gap < n && -----------//952
---------- pos[i + gap / 2] == pos[j + gap / 2];} --------//953
void buildSA(string s) { ---------------------------------//954
--- s += '$'; n = s.length(); ----------------------------//955
--- for (int i = 0; i < n; i++){sa[i]=i; pos[i]=s[i];} ---//956
--- sort (sa, sa + n, cmp); ------------------------------//957
--- for (gap = 1; gap < n * 2; gap <<= 1) { --------------//958
------- va[sa[0]] = 0; -----------------------------------//959
------- for (int i = 1; i < n; i++) { --------------------//95a
----------- int prev = sa[i - 1], next = sa[i]; ----------//95b
----------- va[next] = equal(prev, next) ? va[prev] : i; -//95c
------- } ------------------------------------------------//95d
------- for (int i = 0; i < n; ++i) ----------------------//95e
----------- { pos[i] = va[i]; va[i] = sa[i]; c[i] = i; } -//95f
------- for (int i = 0; i < n; i++) { --------------------//960
----------- int id = va[i] - gap; ------------------------//961
----------- if (id >= 0) sa[c[pos[id]]++] = id; ----------//962
------- }}} ----------------------------------------------//963
```

4.3. **Longest Common Prefix.** Find the length of the longest common prefix for every substring in $O(n)$.

```
int lcp[N]; // lcp[i] = LCP(s[sa[i]:], s[sa[i+1]:]) ------//8f2
void buildLCP(string s) {// build suffix array first -----//8f3
--- for (int i = 0, k = 0; i < n; i++) { -----------------//8f4
------- if (pos[i] != n - 1) { ---------------------------//8f5
----------- for(int j = sa[pos[i]+1]; s[i+k]==s[j+k];k++);//8f6
----------- lcp[pos[i]] = k; if (k > 0) k--; -------------//8f7
--- } else { lcp[pos[i]] = 0; }}} ------------------------//8f8
```

4.4. **Aho-Corasick Trie.** Find all multiple pattern matches in $O(n)$ time. This is KMP for multiple strings.

```
class Node { ---------------------------------------------//89a
--- HashMap<Character, Node> next = new HashMap<>(); -----//89b
--- Node fail = null; ------------------------------------//89c
--- long count = 0; --------------------------------------//89d
--- public void add(String s) { // adds string to trie ---//89e
------- Node node = this; --------------------------------//89f
------- for (char c : s.toCharArray()) { -----------------//8a0
----------- if (!node.contains(c)) -----------------------//8a1
--------------- node.next.put(c, new Node()); ------------//8a2
```

```
----------- node = node.get(c); -------------------------//8a3
------- } node.count++; } -------------------------------//8a4
--- public void prepare() { -----------------------------//8a5
------- // prepares fail links of Aho-Corasick Trie ------//8a6
------- Node root = this; root.fail = null; -------------//8a7
------- Queue<Node> q = new ArrayDeque<Node>(); ---------//8a8
------- for (Node child : next.values()) // BFS ---------//8a9
----------- { child.fail = root; q.offer(child); } -------//8aa
------- while (!q.isEmpty()) { --------------------------//8ab
----------- Node head = q.poll(); -----------------------//8ac
----------- for (Character letter : head.next.keySet()) { //8ad
----------- // traverse upwards to get nearest fail link -//8ae
--------------- Node p = head; --------------------------//8af
--------------- Node nextNode = head.get(letter); --------//8b0
--------------- do { p = p.fail; } ----------------------//8b1
--------------- while(p != root && !p.contains(letter)); -//8b2
--------------- if (p.contains(letter)) { // fail link found
------------------- p = p.get(letter); ------------------//8b4
------------------- nextNode.fail = p; ------------------//8b5
------------------- nextNode.count += p.count; ----------//8b6
--------------- } else { nextNode.fail = root; } ---------//8b7
--------------- q.offer(nextNode); ----------------------//8b8
----------- }}} -----------------------------------------//8b9
--- public BigInteger search(String s) { ----------------//8ba
------- // counts the words added in trie present in s ---//8bb
------- Node root = this, p = this; ---------------------//8bc
------- BigInteger ans = BigInteger.ZERO; ---------------//8bd
------- for (char c : s.toCharArray()) { ----------------//8be
----------- while (p != root && !p.contains(c)) p = p.fail;
----------- if (p.contains(c)) { ------------------------//8c0
--------------- p = p.get(c); ---------------------------//8c1
--------------- ans = ans.add(BigInteger.valueOf(p.count));
----------- } -------------------------------------------//8c3
------- } return ans; } ---------------------------------//8c4
--- // helper methods -------------------------------------//8c5
--- private Node get(char c) { return next.get(c); } -----//8c6
--- private boolean contains(char c) { ------------------//8c7
------- return next.containsKey(c); ---------------------//8c8
}} // Usage: Node trie = new Node(); --------------------//8c9
// for (String s : dictionary) trie.add(s); -------------//8ca
// trie.prepare(); BigInteger m = trie.search(str); ------//8cb
```

4.5. **Palindromic Tree.** Find lengths and frequencies of all palindromic substrings of a string in $O(n)$ time.

Theorem: there can only be up to n unique palindromic substrings for any string.

```
int par[N*2+1], child[N*2+1][128]; ----------------------//8f9
int len[N*2+1], node[N*2+1], cs[N*2+1], size; -----------//8fa
long long cnt[N + 2]; // count can be very large ---------//8fb
int newNode(int p = -1) { -------------------------------//8fc
--- cnt[size] = 0; par[size] = p; -----------------------//8fd
--- len[size] = (p == -1 ? 0 : len[p] + 2); -------------//8fe
--- memset(child[size], -1, sizeof child[size]); ---------//8ff
--- return size++; --------------------------------------//900
} -------------------------------------------------------//901
int get(int i, char c) { --------------------------------//902
--- if (child[i][c] == -1) child[i][c] = newNode(i); -----//903
--- return child[i][c]; ---------------------------------//904
} -------------------------------------------------------//905
void manachers(char s[]) { ------------------------------//906
--- int n = strlen(s), cn = n * 2 + 1; ------------------//907
--- for (int i = 0; i < n; i++) -------------------------//908
------- {cs[i * 2] = -1; cs[i * 2 + 1] = s[i];} ---------//909
--- size = n * 2; ---------------------------------------//90a
--- int odd = newNode(), even = newNode(); --------------//90b
--- int cen = 0, rad = 0, L = 0, R = 0; -----------------//90c
--- size = 0; len[odd] = -1; ----------------------------//90d
--- for (int i = 0; i < cn; i++) ------------------------//90e
------- node[i] = (i % 2 == 0 ? even : get(odd, cs[i])); -//90f
--- for (int i = 1; i < cn; i++) { ----------------------//910
------- if (i > rad) { L = i - 1; R = i + 1; } ----------//911
------- else { ------------------------------------------//912
----------- int M = cen * 2 - i; // retrieve from mirror -//913
----------- node[i] = node[M]; --------------------------//914
----------- if (len[node[M]] < rad - i) L = -1; ---------//915
----------- else { --------------------------------------//916
--------------- R = rad + 1; L = i * 2 - R; -------------//917
--------------- while (len[node[i]] > rad - i) ----------//918
------------------- node[i] = par[node[i]]; -------------//919
----------- } -------------------------------------------//91a
------- } // expand palindrome --------------------------//91b
------- while (L >= 0 && R < cn && cs[L] == cs[R]) { -----//91c
----------- if (cs[L] != -1) node[i] = get(node[i],cs[L]);//91d
----------- L--, R++; -----------------------------------//91e
------- } -----------------------------------------------//91f
------- cnt[node[i]]++; ---------------------------------//920
------- if (i + len[node[i]] > rad) ---------------------//921
------- { rad = i + len[node[i]]; cen = i; } ------------//922
--- } ---------------------------------------------------//923
--- for (int i = size - 1; i >= 0; --i) -----------------//924
--- cnt[par[i]] += cnt[i]; // update parent count --------//925
} -------------------------------------------------------//926
int countUniquePalindromes(char s[]) --------------------//927
--- {manachers(s); return size;} ------------------------//928
int countAllPalindromes(char s[]) { ---------------------//929
--- manachers(s); int total = 0; ------------------------//92a
--- for (int i = 0; i < size; i++) total += cnt[i]; ------//92b
--- return total;} --------------------------------------//92c
// longest palindrome substring of s --------------------//92d
string longestPalindrome(char s[]) { --------------------//92e
--- manachers(s); ---------------------------------------//92f
--- int n = strlen(s), cn = n * 2 + 1, mx = 0; ----------//930
--- for (int i = 1; i < cn; i++) ------------------------//931
------- if (len[node[mx]] < len[node[i]]) ---------------//932
----------- mx = i; -------------------------------------//933
--- int pos = (mx - len[node[mx]]) / 2; -----------------//934
--- return string(s + pos, s + pos + len[node[mx]]); } ---//935
```

4.6. **Z Algorithm.** Find the longest common prefix of all substrings of s with itself in $O(n)$ time.

```
int z[N]; // z[i] = lcp(s, s[i:]) -----------------------//964
void computeZ(string s) { -------------------------------//965
--- int n = s.length(), L = 0, R = 0; z[0] = n; ----------//966
--- for (int i = 1; i < n; i++) { -----------------------//967
------- if (i > R) { ------------------------------------//968
----------- L = R = i; ----------------------------------//969
----------- while (R < n && s[R - L] == s[R]) R++; -------//96a
----------- z[i] = R - L; R--; --------------------------//96b
------- } else { ----------------------------------------//96c
----------- int k = i - L; ------------------------------//96d
----------- if (z[k] < R - i + 1) z[i] = z[k]; ----------//96e
----------- else { --------------------------------------//96f
--------------- L = i; ----------------------------------//970
--------------- while (R < n && s[R - L] == s[R]) R++; ---//971
--------------- z[i] = R - L; R--; ----------------------//972
----------- }}}} ----------------------------------------//973
```

4.7. **Booth's Minimum String Rotation.** Booth's Algo: Find the index of the lexicographically least string rotation in $O(n)$ time.

```
int f[N * 2]; -------------------------------------------//8cc
int booth(string S) { -----------------------------------//8cd
--- S.append(S); // concatenate itself ------------------//8ce
--- int n = S.length(), i, j, k = 0; --------------------//8cf
--- memset(f, -1, sizeof(int) * n); ---------------------//8d0
--- for (j = 1; j < n; j++) { ---------------------------//8d1
------- i = f[j-k-1]; -----------------------------------//8d2
------- while (i != -1 && S[j] != S[k + i + 1]) { --------//8d3
----------- if (S[j] < S[k + i + 1]) k = j - i - 1; ------//8d4
----------- i = f[i]; -----------------------------------//8d5
------- } if (i == -1 && S[j] != S[k + i + 1]) { ---------//8d6
----------- if (S[j] < S[k + i + 1]) k = j; -------------//8d7
----------- f[j - k] = -1; ------------------------------//8d8
------- } else f[j - k] = i + 1; ------------------------//8d9
--- } return k; } ---------------------------------------//8da
```

4.8. **Hashing.**

4.8.1. *Polynomial Hashing.*

```
int MAXN = 1e5+1, MOD = 1e9+7; --------------------------//936
struct hasher { -----------------------------------------//937
- int n; ------------------------------------------------//938
- std::vector<ll> *p_pow; -------------------------------//939
- std::vector<ll> *h_ans; -------------------------------//93a
- hash(vi &s, vi primes) { ------------------------------//93b
--- n = primes.size(); ----------------------------------//93c
--- p_pow = new std::vector<ll>[n]; ---------------------//93d
--- h_ans = new std::vector<ll>[n]; ---------------------//93e
--- for (int i = 0; i < n; ++i) { -----------------------//93f
----- p_pow[i] = std::vector<ll>(MAXN); -----------------//940
----- p_pow[i][0] = 1; ----------------------------------//941
----- for (int j = 0; j+1 < MAXN; ++j) ------------------//942
------- p_pow[i][j+1] = (p_pow[i][j] * primes[i]) % MOD; -//943
----- h_ans[i] = std::vector<ll>(MAXN); -----------------//944
----- h_ans[i][0] = 0; ----------------------------------//945
----- for (int j = 0; j < s.size(); ++j) ----------------//946
------- h_ans[i][j+1] = (h_ans[i][j] + ------------------//947
----------------------- s[j] * p_pow[i][j]) % MOD; ------//948
--- } ---------------------------------------------------//949
```

```
- } ----------------------------------------------------------//94a
}; -----------------------------------------------------------//94b
```

5. Dynamic Programming

5.1. Longest Common Subsequence.

5.2. Longest Increasing Subsequence.

5.3. Traveling Salesman.

6. Number Theory

6.1. Eratosthenes Prime Sieve.

```
bitset<N> is; // #include <bitset> -------------------------//87a
int pr[N], primes = 0; -------------------------------------//87b
void sieve() { ---------------------------------------------//87c
--- is[2] = true; pr[primes++] = 2; ------------------------//87d
--- for (int i = 3; i < N; i += 2) is[i] = 1; --------------//87e
--- for (int i = 3; i*i < N; i += 2) -----------------------//87f
------- if (is[i]) -----------------------------------------//880
----------- for (int j = i*i; j < N; j += i) ---------------//881
--------------- is[j] = 0; ---------------------------------//882
--- for (int i = 3; i < N; i += 2) -------------------------//883
------- if (is[i]) -----------------------------------------//884
----------- pr[primes++] = i;} -----------------------------//885
```

6.2. Divisor Sieve.

```
int divisors[N]; // initially 0 ----------------------------//829
void divisorSieve() { --------------------------------------//82a
--- for (int i = 1; i < N; i++) ----------------------------//82b
------- for (int j = i; j < N; j += i) ---------------------//82c
----------- divisors[j]++;} --------------------------------//82d
```

6.3. Number/Sum of Divisors.

If a number n is prime factorized where $n = {p_1}^{e_1} \times {p_2}^{e_2} \times \cdots \times {p_k}^{e_k}$, where σ_0 is the number of divisors while σ_1 is the sum of divisors:

$$\sum_{d|n} d^k = \sigma_k(n) = \prod \frac{{p_i}^{k(e_i)+1} - 1}{p_i - 1}$$

$$\text{Product: } \prod_{d|n} d = n^{\frac{\sigma_1(n)}{2}}$$

6.4. Möbius Sieve.

The Möbius function μ is the Möbius inverse of e such that $e(n) = \sum_{d|n} \mu(d)$.

```
bitset<N> is; int mu[N]; -----------------------------------//868
void mobiusSieve() { ---------------------------------------//869
--- for (int i = 1; i < N; ++i) mu[i] = 1; -----------------//86a
--- for (int i = 2; i < N; ++i) if (!is[i]) { --------------//86b
------- for (int j = i; j < N; j += i){ --------------------//86c
----------- is[j] = 1; -------------------------------------//86d
----------- mu[j] *= -1; -----------------------------------//86e
------- } --------------------------------------------------//86f
------- for (long long j = 1LL*i*i; j < N; j += i*i) -------//870
----------- mu[j] = 0;} ------------------------------------//871
```

6.5. Möbius Inversion.

Given arithmetic functions f and g:

$$g(n) = \sum_{d|n} f(d) \quad \Leftrightarrow \quad f(n) = \sum_{d|n} \mu(d)\, g\left(\frac{n}{d}\right)$$

6.6. GCD Subset Counting.

Count number of subsets $S \subseteq A$ such that $\gcd(S) = g$ (modifiable).

```
int f[MX+1]; // MX is maximum number of array --------------//83d
long long gcnt[MX+1]; // gcnt[G]: answer when gcd==G -------//83e
long long C(int f) {return (1ll << f) - 1;} ----------------//83f
// f: frequency count --------------------------------------//840
// C(f): # of subsets of f elements (YOU CAN EDIT) ---------//841
void gcd_counter(int a[], int n) { -------------------------//842
--- memset(f, 0, sizeof f); --------------------------------//843
--- memset(gcnt, 0, sizeof gcnt); --------------------------//844
--- int mx = 0; --------------------------------------------//845
--- for (int i = 0; i < n; ++i) { --------------------------//846
------- f[a[i]] += 1; --------------------------------------//847
------- mx = max(mx, a[i]); --------------------------------//848
--- } ------------------------------------------------------//849
--- for (int i = mx; i >= 1; --i) { ------------------------//84a
------- int add = f[i]; ------------------------------------//84b
------- long long sub = 0; ---------------------------------//84c
------- for (int j = 2*i; j <= mx; j += i) { ---------------//84d
----------- add += f[j]; -----------------------------------//84e
----------- sub += gcnt[j]; --------------------------------//84f
------- } --------------------------------------------------//850
------- gcnt[i] = C(add) - sub; ----------------------------//851
--- }} // Usage: int subsets_with_gcd_1 = gcnt[1]; ---------//852
```

6.7. Euler Totient.

Counts all integers from 1 to n that are relatively prime to n in $O(\sqrt{n})$ time.

```
LL totient(LL n) { -----------------------------------------//891
--- if (n <= 1) return 1; ----------------------------------//892
--- LL tot = n; --------------------------------------------//893
--- for (int i = 2; i * i <= n; i++) { ---------------------//894
------- if (n % i == 0) tot -= tot / i; --------------------//895
------- while (n % i == 0) n /= i; -------------------------//896
--- } ------------------------------------------------------//897
--- if (n > 1) tot -= tot / n; -----------------------------//898
--- return tot; } ------------------------------------------//899
```

6.8. Euler Phi Sieve.

Sieve version of Euler totient, runs in $O(N \log N)$ time. Note that $n = \sum_{d|n} \varphi(d)$.

```
bitset<N> is; int phi[N]; ----------------------------------//872
void phiSieve() { ------------------------------------------//873
--- for (int i = 1; i < N; ++i) phi[i] = i; ----------------//874
--- for (int i = 2; i < N; ++i) if (!is[i]) { --------------//875
------- for (int j = i; j < N; j += i) { -------------------//876
----------- phi[j] -= phi[j] / i; --------------------------//877
----------- is[j] = true; ----------------------------------//878
------- }}} ------------------------------------------------//879
```

6.9. Extended Euclidean.

Assigns x, y such that $ax + by = \gcd(a, b)$ and returns $\gcd(a, b)$.

```
typedef long long LL; --------------------------------------//82e
typedef pair<LL, LL> PAIR; ---------------------------------//82f
LL mod(LL x, LL m) { // use this instead of x % m ----------//830
--- if (m == 0) return 0; ----------------------------------//831
--- if (m < 0) m *= -1; ------------------------------------//832
--- return (x%m + m) % m; // always nonnegative ------------//833
} ----------------------------------------------------------//834
LL extended_euclid(LL a, LL b, LL &x, LL &y) { -------------//835
--- if (b==0) {x = 1; y = 0; return a;} --------------------//836
--- LL g = extended_euclid(b, a%b, x, y); ------------------//837
--- LL z = x - a/b*y; --------------------------------------//838
--- x = y; y = z; return g; --------------------------------//839
} ----------------------------------------------------------//83a
```

6.10. Modular Inverse.

Find unique x such that $ax \equiv 1 \pmod{m}$. Returns 0 if no unique solution is found. Please use modulo solver for the non-unique case.

```
LL modinv(LL a, LL m) { ------------------------------------//85e
--- LL x, y; LL g = extended_euclid(a, m, x, y); -----------//85f
--- if (g == 1 || g == -1) return mod(x * g, m); -----------//860
--- return 0; // 0 if invalid ------------------------------//861
} ----------------------------------------------------------//862
```

6.11. Modulo Solver.

Solve for values of x for $ax \equiv b \pmod{m}$. Returns $(-1, -1)$ if there is no solution. Returns a pair (x, M) where solution is $x \bmod M$.

```
PAIR modsolver(LL a, LL b, LL m) { -------------------------//863
--- LL x, y; LL g = extended_euclid(a, m, x, y); -----------//864
--- if (b % g != 0) return PAIR(-1, -1); -------------------//865
--- return PAIR(mod(x*b/g, m/g), abs(m/g)); ----------------//866
} ----------------------------------------------------------//867
```

6.12. Linear Diophantine.

Computes integers x and y such that $ax + by = c$, returns $(-1, -1)$ if no solution. Tries to return positive integer answers for x and y if possible.

```
PAIR null(-1, -1); // needs extended euclidean -------------//853
PAIR diophantine(LL a, LL b, LL c) { -----------------------//854
--- if (!a && !b) return c ? null : PAIR(0, 0); ------------//855
--- if (!a) return c % b ? null : PAIR(0, c / b); ----------//856
--- if (!b) return c % a ? null : PAIR(c / a, 0); ----------//857
--- LL x, y; LL g = extended_euclid(a, b, x, y); -----------//858
--- if (c % g) return null; --------------------------------//859
--- y = mod(y * (c/g), a/g); -------------------------------//85a
--- if (y == 0) y += abs(a/g); // prefer positive sol. -----//85b
--- return PAIR((c - b*y)/a, y); ---------------------------//85c
} ----------------------------------------------------------//85d
```

6.13. Chinese Remainder Theorem.

Solves linear congruence $x \equiv b_i \pmod{m_i}$. Returns $(-1, -1)$ if there is no solution. Returns a pair (x, M) where solution is $x \bmod M$.

```
PAIR chinese(LL b1, LL m1, LL b2, LL m2) { -----------------//81b
--- LL x, y; LL g = extended_euclid(m1, m2, x, y); ---------//81c
--- if (b1 % g != b2 % g) return PAIR(-1, -1); -------------//81d
--- LL M = abs(m1 / g * m2); -------------------------------//81e
--- return PAIR(mod(mod(x*b2*m1+y*b1*m2, M*g)/g,M),M); -----//81f
} ----------------------------------------------------------//820
PAIR chinese_remainder(LL b[], LL m[], int n) { ------------//821
--- PAIR ans(0, 1); ----------------------------------------//822
--- for (int i = 0; i < n; ++i) { --------------------------//823
------- ans = chinese(b[i],m[i],ans.first,ans.second); -----//824
------- if (ans.second == -1) break; -----------------------//825
------- } --------------------------------------------------//826
--- return ans; --------------------------------------------//827
} ----------------------------------------------------------//828
```

6.13.1. *Super Chinese Remainder.* Solves linear congruence $a_i x \equiv b_i \pmod{m_i}$. Returns $(-1, -1)$ if there is no solution.

```
PAIR super_chinese(LL a[], LL b[], LL m[], int n) { ------//886
--- PAIR ans(0, 1); ---------------------------------------//887
--- for (int i = 0; i < n; ++i) { -------------------------//888
------- PAIR two = modsolver(a[i], b[i], m[i]); ----------//889
------- if (two.second == -1) return two; ----------------//88a
------- ans = chinese(ans.first, ans.second, -------------//88b
------- two.first, two.second); --------------------------//88c
------- if (ans.second == -1) break; ---------------------//88d
--- } -----------------------------------------------------//88e
--- return ans; -------------------------------------------//88f
} ---------------------------------------------------------//890
```

7. Algebra

7.1. **Fast Fourier Transform.** Compute the Discrete Fourier Transform (DFT) of a polynomial in $O(n \log n)$ time.

```
struct poly { ---------------------------------------------//01d
--- double a, b; ------------------------------------------//01e
--- poly(double a=0, double b=0): a(a), b(b) {} ----------//01f
--- poly operator+(const poly& p) const { ----------------//020
------- return poly(a + p.a, b + p.b);} ------------------//021
--- poly operator-(const poly& p) const { ----------------//022
------- return poly(a - p.a, b - p.b);} ------------------//023
--- poly operator*(const poly& p) const { ----------------//024
------- return poly(a*p.a - b*p.b, a*p.b + b*p.a);} ------//025
}; --------------------------------------------------------//026
void fft(poly in[], poly p[], int n, int s) { ------------//027
--- if (n < 1) return; ------------------------------------//028
--- if (n == 1) {p[0] = in[0]; return;} ------------------//029
--- n >>= 1; fft(in, p, n, s << 1); ----------------------//02a
--- fft(in + s, p + n, n, s << 1); -----------------------//02b
--- poly w(1), wn(cos(M_PI/n), sin(M_PI/n)); -------------//02c
--- for (int i = 0; i < n; ++i) { ------------------------//02d
------- poly even = p[i], odd = p[i + n]; ----------------//02e
------- p[i] = even + w * odd; ---------------------------//02f
------- p[i + n] = even - w * odd; -----------------------//030
------- w = w * wn; --------------------------------------//031
--- } -----------------------------------------------------//032
} ---------------------------------------------------------//033
void fft(poly p[], int n) { -------------------------------//034
--- poly *f = new poly[n]; fft(p, f, n, 1); --------------//035
--- copy(f, f + n, p); delete[] f; -----------------------//036
} ---------------------------------------------------------//037
void inverse_fft(poly p[], int n) { -----------------------//038
--- for(int i=0; i<n; i++) {p[i].b *= -1;} fft(p, n); ----//039
--- for(int i=0; i<n; i++) {p[i].a/=n; p[i].b/= -1*n;} ---//03a
} ---------------------------------------------------------//03b
```

7.2. **FFT Polynomial Multiplication.** Multiply integer polynomials a, b of size an, bn using FFT in $O(n \log n)$. Stores answer in an array c, rounded to the nearest integer (or double).

```
// note: c[] should have size of at least (an+bn) --------//00f
int mult(int a[],int an,int b[],int bn,int c[]) { --------//010
--- int n, degree = an + bn - 1; -------------------------//011
--- for (n = 1; n < degree; n <<= 1); // power of 2 ------//012
--- poly *A = new poly[n], *B = new poly[n]; -------------//013
--- copy(a, a + an, A); fill(A + an, A + n, 0); ----------//014
--- copy(b, b + bn, B); fill(B + bn, B + n, 0); ----------//015
--- fft(A, n); fft(B, n); --------------------------------//016
--- for (int i = 0; i < n; i++) A[i] = A[i] * B[i]; ------//017
--- inverse_fft(A, n); -----------------------------------//018
--- for (int i = 0; i < degree; i++) --------------------//019
------- c[i] = int(A[i].a + 0.5); // same as round(A[i].a)//01a
--- delete[] A, B; return degree; ------------------------//01b
} ---------------------------------------------------------//01c
```

7.3. **Polynomial Long Division.** Divide two polynomials A and B to get Q and R, where $\frac{A}{B} = Q + \frac{R}{B}$.

```
typedef vector<double> Poly; -----------------------------//063
Poly Q, R; // quotient and remainder ---------------------//064
void trim(Poly& A) { // remove trailing zeroes -----------//065
--- while (!A.empty() && abs(A.back()) < EPS) ------------//066
--- A.pop_back(); ----------------------------------------//067
} ---------------------------------------------------------//068
void divide(Poly A, Poly B) { -----------------------------//069
--- if (B.size() == 0) throw exception(); ----------------//06a
--- if (A.size() < B.size()) {Q.clear(); R=A; return;} ---//06b
--- Q.assign(A.size() - B.size() + 1, 0); ----------------//06c
--- Poly part; --------------------------------------------//06d
--- while (A.size() >= B.size()) { -----------------------//06e
------- int As = A.size(), Bs = B.size(); ----------------//06f
------- part.assign(As, 0); ------------------------------//070
------- for (int i = 0; i < Bs; i++) ---------------------//071
----------- part[As-Bs+i] = B[i]; ------------------------//072
------- double scale = Q[As-Bs] = A[As-1] / part[As-1]; --//073
------- for (int i = 0; i < As; i++) ---------------------//074
----------- A[i] -= part[i] * scale; ---------------------//075
------- trim(A); -----------------------------------------//076
--- } R = A; trim(Q); } ----------------------------------//077
```

7.4. **Matrix Multiplication.** Multiplies matrices $A_{p \times q}$ and $B_{q \times r}$ in $O(n^3)$ time, modulo `MOD`.

```
long[][] multiply(long A[][], long B[][]) { ---------------//052
--- int p = A.length, q = A[0].length, r = B[0].length; --//053
--- // if(q != B.length) throw new Exception(":((("); ----//054
--- long AB[][] = new long[p][r]; ------------------------//055
--- for (int i = 0; i < p; i++) --------------------------//056
--- for (int j = 0; j < q; j++) --------------------------//057
--- for (int k = 0; k < r; k++) --------------------------//058
------- (AB[i][k] += A[i][j] * B[j][k]) %= MOD; ----------//059
--- return AB; } ------------------------------------------//05a
```

7.5. **Matrix Power.** Computes for B^e in $O(n^3 \log e)$ time. Refer to Matrix Multiplication.

```
long[][] power(long B[][], long e) { ----------------------//05b
--- int n = B.length; ------------------------------------//05c
--- long ans[][]= new long[n][n]; ------------------------//05d
--- for (int i = 0; i < n; i++) ans[i][i] = 1; -----------//05e
--- while (e > 0) { --------------------------------------//05f
------- if (e % 2 == 1) ans = multiply(ans, b); ----------//060
------- b = multiply(b, b); e /= 2; ----------------------//061
--- } return ans;} ---------------------------------------//062
```

7.6. **Fibonacci Matrix.** Fast computation for nth Fibonacci $\{F_1, F_2, \ldots, F_n\}$ in $O(\log n)$:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

7.7. **Gauss-Jordan/Matrix Determinant.** Row reduce matrix A in $O(n^3)$ time. Returns `true` if a solution exists.

```
boolean gaussJordan(double A[][]) { ----------------------//03c
--- int n = A.length, m = A[0].length; -------------------//03d
--- boolean singular = false; -----------------------------//03e
--- // double determinant = 1; ---------------------------//03f
--- for (int i=0, p=0; i<n && p<m; i++, p++) { -----------//040
------- for (int k = i + 1; k < n; k++) { ----------------//041
----------- if (Math.abs(A[k][p]) > EPS) { // swap -------//042
----------- // determinant *= -1; ------------------------//043
----------- double t[]=A[i]; A[i]=A[k]; A[k]=t; ----------//044
----------- break; ---------------------------------------//045
----------- } --------------------------------------------//046
------- } ------------------------------------------------//047
------- // determinant *= A[i][p]; -----------------------//048
------- if (Math.abs(A[i][p]) < EPS) ---------------------//049
----------- { singular = true; i--; continue; } ---------//04a
------- for (int j = m-1; j >= p; j--) A[i][j]/= A[i][p]; //04b
------- for (int k = 0; k < n; k++) { --------------------//04c
----------- if (i == k) continue; ------------------------//04d
----------- for (int j = m-1; j >= p; j--) ---------------//04e
--------------- A[k][j] -= A[k][p] * A[i][j]; ------------//04f
------- } ------------------------------------------------//050
--- } return !singular; } --------------------------------//051
```

8. Combinatorics

8.1. **Lucas Theorem.** Compute $\binom{n}{k} \bmod p$ in $O(p + \log_p n)$ time, where p is a prime.

```
LL f[P], lid; // P: biggest prime ------------------------//0bd
LL lucas(LL n, LL k, int p) { -----------------------------//0be
--- if (k == 0) return 1; --------------------------------//0bf
--- if (n < p && k < p) { --------------------------------//0c0
------- if (lid != p) { ----------------------------------//0c1
----------- lid = p; f[0] = 1; ---------------------------//0c2
----------- for (int i = 0; i < p; ++i) f[i]=f[i-1]*i%p; -//0c3
------- } ------------------------------------------------//0c4
------- return f[n] * modpow(f[n-k]*f[k]%p, p-2, p) % p;} //0c5
--- return lucas(n/p,k/p,p) * lucas(n%p,k%p,p) % p; } ----//0c6
```

8.2. **Granville's Theorem.** Compute $\binom{n}{k} \bmod m$ (for any m) in $O(m^2 \log^2 n)$ time.

```
def fprime(n, p): -----------------------------------------//088
--- # counts the number of prime divisors of n! ----------//089
--- pk, ans = p, 0 ---------------------------------------//08a
--- while pk <= n: ---------------------------------------//08b
------- ans += n // pk -----------------------------------//08c
------- pk *= p ------------------------------------------//08d
--- return ans -------------------------------------------//08e
def granville(n, k, p, E): -------------------------------//08f
--- # n choose k (mod p^E) -------------------------------//090
--- prime_pow = fprime(n,p)-fprime(k,p)-fprime(n-k,p) ----//091
```

```
--- if prime_pow >= E: return 0 ---------------------------//092
--- e = E - prime_pow -------------------------------------//093
--- pe = p ** e -------------------------------------------//094
--- r, f = n - k, [1]*pe ----------------------------------//095
--- for i in range(1, pe): --------------------------------//096
------- x = i ---------------------------------------------//097
------- if x % p == 0: ------------------------------------//098
----------- x = 1 -----------------------------------------//099
------- f[i] = f[i-1] * x % pe ----------------------------//09a
--- numer, denom, negate, ptr = 1, 1, 0, 0 ----------------//09b
--- while n: ----------------------------------------------//09c
------- if f[-1] != 1 and ptr >= e: -----------------------//09d
----------- negate ^= (n&1) ^ (k&1) ^ (r&1) ---------------//09e
------- numer = numer * f[n%pe] % pe ----------------------//09f
------- denom = denom * f[k%pe] % pe * f[r%pe] % pe -------//0a0
------- n, k, r = n//p, k//p, r//p ------------------------//0a1
------- ptr += 1 ------------------------------------------//0a2
--- ans = numer * modinv(denom, pe) % pe ------------------//0a3
--- if negate and (p != 2 or e < 3): ----------------------//0a4
------- ans = (pe - ans) % pe -----------------------------//0a5
--- return mod(ans * p**prime_pow, p**E) ------------------//0a6
def choose(n, k, m): # generalized (n choose k) mod m ----//0a7
--- factors, x, p = [], m, 2 ------------------------------//0a8
--- while p*p <= x: ---------------------------------------//0a9
------- e = 0 ---------------------------------------------//0aa
------- while x % p == 0: ---------------------------------//0ab
----------- e += 1 ----------------------------------------//0ac
----------- x //= p ---------------------------------------//0ad
------- if e: factors.append((p, e)) ----------------------//0ae
------- p += 1 --------------------------------------------//0af
--- if x > 1: factors.append((x, 1)) ----------------------//0b0
--- crt_array = [granville(n,k,p,e) for p, e in factors] -//0b1
--- mod_array = [p**e for p, e in factors] ---------------//0b2
--- return chinese_remainder(crt_array, mod_array)[0] ----//0b3
```

8.3. Derangements.

Compute the number of permutations with n elements such that no element is at their original position:

$$D(n) = (n-1)\left(D(n-1) + D(n-2)\right) = nD(n-1) + (-1)^n$$

8.4. Factoradics.

Convert a permutation of n items to factoradics and vice versa in $O(n \log n)$.

```
// use fenwick tree add, sum, and low code ---------------//078
typedef long long LL; ------------------------------------//079
void factoradic(int arr[], int n) { // 0 to n-1 ---------//07a
--- for (int i = 0; i <=n; i++) fen[i] = 0; --------------//07b
--- for (int i = 1; i < n; i++) add(i, 1); ---------------//07c
--- for (int i = 0; i < n; i++) { ------------------------//07d
--- int s = sum(arr[i]); ---------------------------------//07e
--- add(arr[i], -1); arr[i] = s; -------------------------//07f
--- }} ---------------------------------------------------//080
void permute(int arr[], int n) { // factoradic to perm ---//081
--- for (int i = 0; i <=n; i++) fen[i] = 0; --------------//082
--- for (int i = 1; i < n; i++) add(i, 1); ---------------//083
--- for (int i = 0; i < n; i++) { ------------------------//084
--- arr[i] = low(arr[i] - 1); ----------------------------//085
--- add(arr[i], -1); -------------------------------------//086
--- }} ---------------------------------------------------//087
```

8.5. kth Permutation.

Get the next kth permutation of n items, if exists, using factoradics. All values should be from 0 to $n-1$. Use factoradics methods as discussed above.

```
bool kth_permutation(int arr[], int n, LL k) { ----------//0b4
--- factoradic(arr, n); // values from 0 to n-1 ---------//0b5
--- for (int i = n-1; i >= 0 && k > 0; --i){ ------------//0b6
------- LL temp = arr[i] + k; ---------------------------//0b7
------- arr[i] = temp % (n - i); ------------------------//0b8
------- k = temp / (n - i); -----------------------------//0b9
--- } ---------------------------------------------------//0ba
--- permute(arr, n); ------------------------------------//0bb
--- return k == 0; } ------------------------------------//0bc
```

8.6. Catalan Numbers.

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1}$$

(1) The number of non-crossing partitions of an n-element set
(2) The number of expressions with n pairs of parentheses
(3) The number of ways $n+1$ factors can be parenthesized
(4) The number of full binary trees with $n+1$ leaves
(5) The number of monotonic lattice paths of an $n \times n$ grid (5-SAT problem)
(6) The number of triangulations of a convex polygon with $n+2$ sides (non-rotational)
(7) The number of permutations $\{1, \ldots, n\}$ without a 3-term increasing subsequence
(8) The number of ways to form a mountain range with n ups and n downs

8.7. Stirling Numbers.

s_1: Count the number of permutations of n elements with k disjoint cycles

s_2: Count the ways to partition a set of n elements into k nonempty subsets

$$s_1(n,k) = \begin{cases} 1 & n = k = 0 \\ s_1(n-1,k-1) - (n-1)s_1(n-1,k) & n,k > 0 \\ 0 & \text{elsewhere} \end{cases}$$

$$s_2(n,k) = \begin{cases} 1 & n = k = 0 \\ s_2(n-1,k-1) + ks_2(n-1,k) & n,k > 0 \\ 0 & \text{elsewhere} \end{cases}$$

8.8. Partition Function.

Pregenerate the number of partitions of positive integer n with n positive addends.

$$p(n,k) = \begin{cases} 1 & n = k = 0 \\ 0 & n < k \\ p(n-1,k-1) + p(n-k,k) & n \geq k \end{cases}$$

9. Geometry

```
#include <complex> ---------------------------------------//33c
#define x real() -----------------------------------------//33d
#define y imag() -----------------------------------------//33e
typedef std::complex<double> point; // 2D point only -----//33f
const double PI = acos(-1.0), EPS = 1e-7; ----------------//340
```

9.1. Dots and Cross Products.

```
double dot(point a, point b) -----------------------------//38a
- {return a.x * b.x + a.y * b.y;} // + a.z * b.z; --------//38b
double cross(point a, point b) ---------------------------//38c
- {return a.x * b.y - a.y * b.x;} ------------------------//38d
double cross(point a, point b, point c) ------------------//38e
- {return cross(a, b) + cross(b, c) + cross(c, a);} ------//38f
double cross3D(point a, point b) { -----------------------//390
- return point(a.x*b.y - a.y*b.x, a.y*b.z - --------------//391
-------------- a.z*b.y, a.z*b.x - a.x*b.z);} -------------//392
```

9.2. Angles and Rotations.

```
double angle(point a, point b, point c) { ----------------//2ff
- // angle formed by abc in radians: PI < x <= PI --------//300
- return abs(remainder(arg(a-b) - arg(c-b), 2*PI));} -----//301
point rotate(point p, point a, double d) { ---------------//302
- //rotate point a about pivot p CCW at d radians --------//303
- return p + (a - p) * point(cos(d), sin(d));} -----------//304
```

9.3. Spherical Coordinates.

$$\begin{aligned} x &= r\cos\theta\cos\phi & r &= \sqrt{x^2+y^2+z^2} \\ y &= r\cos\theta\sin\phi & \theta &= \cos^{-1} x/r \\ z &= r\sin\theta & \phi &= \operatorname{atan2}(y,x) \end{aligned}$$

9.4. Point Projection.

```
point proj(point p, point v) { ---------------------------//417
- // project point p onto a vector v (2D & 3D) -----------//418
- return dot(p, v) / norm(v) * v;} -----------------------//419
point projLine(point p, point a, point b) { --------------//41a
- // project point p onto line ab (2D & 3D) --------------//41b
- return a + dot(p-a, b-a) / norm(b-a) * (b-a);} ---------//41c
point projSeg(point p, point a, point b) { ---------------//41d
- // project point p onto segment ab (2D & 3D) -----------//41e
- double s = dot(p-a, b-a) / norm(b-a); ------------------//41f
- return a + min(1.0, max(0.0, s)) * (b-a);} -------------//420
point projPlane(point p, double a, double b, -------------//421
--------------- double c, double d) { --------------------//422
- // project p onto plane ax+by+cz+d=0 (3D) --------------//423
- // same as: o + p - project(p - o, n); -----------------//424
- double k = -d / (a*a + b*b + c*c); ---------------------//425
- point o(a*k, b*k, c*k), n(a, b, c); --------------------//426
- point v(p.x-o.x, p.y-o.y, p.z-o.z); --------------------//427
- double s = dot(v, n) / dot(n, n); ----------------------//428
- return point(o.x + p.x + s * n.x, o.y + ----------------//429
-------------- p.y +s * n.y, o.z + p.z + s * n.z);} ------//42a
```

9.5. Great Circle Distance.

```
double greatCircleDist(double lat1, double long1, --------//393
--- double lat2, double long2, double R) { ---------------//394
- long1 *= PI / 180; lat1 *= PI / 180; // to radians -----//395
- long2 *= PI / 180; lat2 *= PI / 180; -------------------//396
- return R*acos(sin(lat1)*sin(lat2) + --------------------//397
------- cos(lat1)*cos(lat2)*cos(abs(long1 - long2))); ---//398
} --------------------------------------------------------//399
// another version, using actual (x, y, z) ---------------//39a
double greatCircleDist(point a, point b) { ---------------//39b
```

```
- return atan2(abs(cross3D(a, b)), dot3D(a, b)); ---------//39c
} -------------------------------------------------------//39d
```

9.6. **Point/Line/Plane Distances.**

```
double distPtLine(point p, double a, double b, ----------//370
--- double c) { ----------------------------------------//371
- // dist from point p to line ax+by+c=0 -----------------//372
- return abs(a*p.x + b*p.y + c) / sqrt(a*a + b*b);} ------//373
double distPtLine(point p, point a, point b) { ----------//374
- // dist from point p to line ab -------------------------//375
- return abs((a.y - b.y) * (p.x - a.x) + -----------------//376
-------- (b.x - a.x) * (p.y - a.y)) / -------------------//377
-------- hypot(a.x - b.x, a.y - b.y);} ------------------//378
double distPtPlane(point p, double a, double b, ---------//379
----- double c, double d) { ----------------------------//37a
- // distance to 3D plane ax + by + cz + d = 0 -----------//37b
- return (a*p.x+b*p.y+c*p.z+d)/sqrt(a*a+b*b+c*c); --------//37c
} /*! // distance between 3D lines AB & CD (untested) ----//37d
double distLine3D(point A,point B,point C,point D){ ------//37e
- point u = B - A, v = D - C, w = A - C; -----------------//37f
- double a = dot(u, u), b = dot(u, v); -------------------//380
- double c = dot(v, v), d = dot(u, w); -------------------//381
- double e = dot(v, w), det = a*c - b*b; -----------------//382
- double s = det < EPS ? 0.0 : (b*e - c*d) / det; --------//383
- double t = det < EPS -----------------------------------//384
--- ? (b > c ? d/b : e/c) // parallel -------------------//385
--- : (a*e - b*d) / det; --------------------------------//386
- point top = A + u * s, bot = w - A - v * t; -----------//387
- return dist(top, bot); ---------------------------------//388
} // dist<EPS: intersection     */ ----------------------//389
```

9.7. **Intersections.**

9.7.1. *Line-Segment Intersection.* Get intersection points of 2D lines/segments $\overline{ab}$ and $\overline{cd}$.

```
point null(HUGE_VAL, HUGE_VAL); -------------------------//3c9
point line_inter(point a, point b, point c, -------------//3ca
---------------- point d, bool seg = false) { -----------//3cb
- point ab(b.x - a.x, b.y - a.y); ------------------------//3cc
- point cd(d.x - c.x, d.y - c.y); ------------------------//3cd
- point ac(c.x - a.x, c.y - a.y); ------------------------//3ce
- double D = -cross(ab, cd); // determinant --------------//3cf
- double Ds = cross(cd, ac); -----------------------------//3d0
- double Dt = cross(ab, ac); -----------------------------//3d1
- if (abs(D) < EPS) { // parallel ------------------------//3d2
--- if (seg && abs(Ds) < EPS) { // collinear ------------//3d3
----- point p[] = {a, b, c, d}; -------------------------//3d4
----- sort(p, p + 4, [](point a, point b) { -------------//3d5
------- return a.x < b.x-EPS || -------------------------//3d6
-------------- (dist(a,b) < EPS && a.y < b.y-EPS); -------//3d7
----- }); -----------------------------------------------//3d8
----- return dist(p[1], p[2]) < EPS ? p[1] : null; -------//3d9
--- } ---------------------------------------------------//3da
--- return null; ----------------------------------------//3db
- } -----------------------------------------------------//3dc
- double s = Ds / D, t = Dt / D; -------------------------//3dd
- if (seg && (min(s,t)<-EPS||max(s,t)>1+EPS)) ------------//3de
--- return null; ----------------------------------------//3df
- return point(a.x + s * ab.x, a.y + s * ab.y); ----------//3e0
}/* double A = cross(d-a, b-a), B = cross(c-a, b-a); -----//3e1
return (B*d - A*c)/(B - A); */ --------------------------//3e2
```

9.7.2. *Circle-Line Intersection.* Get intersection points of circle at center c, radius r, and line $\overline{ab}$.

```
std::vector<point> CL_inter(point c, double r, ----------//316
--- point a, point b) { --------------------------------//317
- point p = projLine(c, a, b); ---------------------------//318
- double d = abs(c - p); vector<point> ans; --------------//319
- if (d > r + EPS); // none ------------------------------//31a
- else if (d > r - EPS) ans.push_back(p); // tangent -----//31b
- else if (d < EPS) { // diameter ------------------------//31c
--- point v = r * (b - a) / abs(b - a); -----------------//31d
--- ans.push_back(c + v); -------------------------------//31e
--- ans.push_back(c - v); -------------------------------//31f
- } else { -----------------------------------------------//320
--- double t = acos(d / r); -----------------------------//321
--- p = c + (p - c) * r / d; ----------------------------//322
--- ans.push_back(rotate(c, p, t)); ---------------------//323
--- ans.push_back(rotate(c, p, -t)); --------------------//324
- } return ans; ------------------------------------------//325
} -------------------------------------------------------//326
```

9.7.3. *Circle-Circle Intersection.*

```
std::vector<point> CC_intersection(point c1, ------------//305
--- double r1, point c2, double r2) { ------------------//306
- double d = dist(c1, c2); -------------------------------//307
- vector<point> ans; -------------------------------------//308
- if (d < EPS) { -----------------------------------------//309
--- if (abs(r1-r2) < EPS); // inf intersections ---------//30a
- } else if (r1 < EPS) { ---------------------------------//30b
--- if (abs(d - r2) < EPS) ans.push_back(c1); -----------//30c
- } else { -----------------------------------------------//30d
--- double s = (r1*r1 + d*d - r2*r2) / (2*r1*d); --------//30e
--- double t = acos(max(-1.0, min(1.0, s))); ------------//30f
--- point mid = c1 + (c2 - c1) * r1 / d; ----------------//310
--- ans.push_back(rotate(c1, mid, t)); ------------------//311
--- if (abs(sin(t)) >= EPS) -----------------------------//312
----- ans.push_back(rotate(c2, mid, -t)); ---------------//313
- } return ans; ------------------------------------------//314
} -------------------------------------------------------//315
```

9.8. **Polygon Areas.** Find the area of any 2D polygon given as points in $O(n)$.

```
double area(point p[], int n) { -------------------------//3fd
- double a = 0; ------------------------------------------//3fe
- for (int i = 0, j = n - 1; i < n; j = i++) -------------//3ff
--- a += cross(p[i], p[j]); -----------------------------//400
- return abs(a) / 2; } -----------------------------------//401
```

9.8.1. *Triangle Area.* Find the area of a triangle using only their lengths. Lengths must be valid.

```
double area(double a, double b, double c) { -------------//436
- double s = (a + b + c) / 2; ----------------------------//437
- return sqrt(s*(s-a)*(s-b)*(s-c)); } --------------------//438
```

Cyclic Quadrilateral Area. Find the area of a cyclic quadrilateral using only their lengths. A quadrilateral is cyclic if its inner angles sum up to 360°.

```
double area(double a, double b, double c, double d) { ----//36d
- double s = (a + b + c + d) / 2; ------------------------//36e
- return sqrt((s-a)*(s-b)*(s-c)*(s-d)); } ----------------//36f
```

9.9. **Polygon Centroid.** Get the centroid/center of mass of a polygon in $O(m)$.

```
point centroid(point p[], int n) { ----------------------//402
- point ans(0, 0); ---------------------------------------//403
- double z = 0; ------------------------------------------//404
- for (int i = 0, j = n - 1; i < n; j = i++) { -----------//405
--- double cp = cross(p[j], p[i]); ----------------------//406
--- ans += (p[j] + p[i]) * cp; --------------------------//407
--- z += cp; --------------------------------------------//408
- } return ans / (3 * z); } ------------------------------//409
```

9.10. **Convex Hull.** Get the convex hull of a set of points using Graham-Andrew's scan. This sorts the points at $O(n \log n)$, then performs the Monotonic Chain Algorithm at $O(n)$.

```
// counterclockwise hull in p[], returns size of hull ----//341
bool xcmp(const point& a, const point& b) ---------------//342
- {return a.x < b.x || (a.x == b.x && a.y < b.y);} -------//343
int convex_hull(point p[], int n) { ---------------------//344
- sort(p, p + n, xcmp); if (n <= 1) return n; ------------//345
- int k = 0; point *h = new point[2 * n]; ----------------//346
- double zer = EPS; // -EPS to include collinears --------//347
- for (int i = 0; i < n; h[k++] = p[i++]) ----------------//348
--- while (k >= 2 && cross(h[k-2],h[k-1],p[i]) < zer) ----//349
----- --k; ----------------------------------------------//34a
- for(int i = n-2, t = k; i >= 0; h[k++] = p[i--]) -------//34b
--- while (k > t && cross(h[k-2],h[k-1],p[i]) < zer) -----//34c
----- --k; ----------------------------------------------//34d
- k -= 1 + (h[0].x==h[1].x&&h[0].y==h[1].y ? 1 : 0); -----//34e
- copy(h, h + k, p); delete[] h; return k; } -------------//34f
```

9.11. **Point in Polygon.** Check if a point is strictly inside (or on the border) of a polygon in $O(n)$.

```
bool inPolygon(point q, point p[], int n) { -------------//40a
- bool in = false; ---------------------------------------//40b
- for (int i = 0, j = n - 1; i < n; j = i++) -------------//40c
--- in ^= (((p[i].y > q.y) != (p[j].y > q.y)) && --------//40d
----- q.x < (p[j].x - p[i].x) * (q.y - p[i].y) / --------//40e
----- (p[j].y - p[i].y) + p[i].x); ----------------------//40f
- return in; } -------------------------------------------//410
bool onPolygon(point q, point p[], int n) { -------------//411
- for (int i = 0, j = n - 1; i < n; j = i++) -------------//412
- if (abs(dist(p[i], q) + dist(p[j], q) - ----------------//413
--------- dist(p[i], p[j])) < EPS) ----------------------//414
--- return true; ----------------------------------------//415
- return false; } ----------------------------------------//416
```

9.12. **Cut Polygon by a Line.** Cut polygon by line $\overline{ab}$ to its left in $O(n)$, such that $\angle abp$ is counter-clockwise.

```
vector<point> cut(point p[],int n,point a,point b) { -----//364
- vector<point> poly; ------------------------------------//365
- for (int i = 0, j = n - 1; i < n; j = i++) { ----------//366
--- double c1 = cross(a, b, p[j]); ----------------------//367
--- double c2 = cross(a, b, p[i]); ----------------------//368
--- if (c1 > -EPS) poly.push_back(p[j]); ----------------//369
--- if (c1 * c2 < -EPS) ---------------------------------//36a
----- poly.push_back(line_inter(p[j], p[i], a, b)); ------//36b
- } return poly; } --------------------------------------//36c
```

9.13. **Triangle Centers.**

```
point bary(point A, point B, point C, -------------------//439
--------- double a, double b, double c) { ---------------//43a
- return (A*a + B*b + C*c) / (a + b + c);} ---------------//43b
point trilinear(point A, point B, point C, --------------//43c
-------------- double a, double b, double c) { ----------//43d
- return bary(A,B,C,abs(B-C)*a, -------------------------//43e
------------ abs(C-A)*b,abs(A-B)*c);} -------------------//43f
point centroid(point A, point B, point C) { -------------//440
- return bary(A, B, C, 1, 1, 1);} -----------------------//441
point circumcenter(point A, point B, point C) { ---------//442
- double a=norm(B-C), b=norm(C-A), c=norm(A-B); ---------//443
- return bary(A,B,C,a*(b+c-a),b*(c+a-b),c*(a+b-c));} -----//444
point orthocenter(point A, point B, point C) { ----------//445
- return bary(A,B,C, tan(angle(B,A,C)), -----------------//446
------------ tan(angle(A,B,C)), tan(angle(A,C,B)));} ----//447
point incenter(point A, point B, point C) { -------------//448
- return bary(A,B,C,abs(B-C),abs(A-C),abs(A-B));} --------//449
// incircle radius given the side lengths a, b, c --------//44a
double inradius(double a, double b, double c) { ---------//44b
- double s = (a + b + c) / 2; ---------------------------//44c
- return sqrt(s * (s-a) * (s-b) * (s-c)) / s;} -----------//44d
point excenter(point A, point B, point C) { -------------//44e
- double a = abs(B-C), b = abs(C-A), c = abs(A-B); -------//44f
- return bary(A, B, C, -a, b, c); ------------------------//450
- // return bary(A, B, C, a, -b, c); --------------------//451
- // return bary(A, B, C, a, b, -c); --------------------//452
} -------------------------------------------------------//453
point brocard(point A, point B, point C) { --------------//454
- double a = abs(B-C), b = abs(C-A), c = abs(A-B); -------//455
- return bary(A,B,C,c/b*a,a/c*b,b/a*c); // CCW ----------//456
- // return bary(A,B,C,b/c*a,c/a*b,a/b*c); // CW ---------//457
} -------------------------------------------------------//458
point symmedian(point A, point B, point C) { ------------//459
- return bary(A,B,C,norm(B-C),norm(C-A),norm(A-B));} -----//45a
```

9.14. **Convex Polygon Intersection.** Get the intersection of two convex polygons in $O(n^2)$.

```
std::vector<point> convex_polygon_inter(point a[], -------//350
--- int an, point b[], int bn) { ------------------------//351
- point ans[an + bn + an*bn]; ---------------------------//352
- int size = 0; -----------------------------------------//353
- for (int i = 0; i < an; ++i) --------------------------//354
--- if (inPolygon(a[i],b,bn) || onPolygon(a[i],b,bn)) ----//355
----- ans[size++] = a[i]; -------------------------------//356
- for (int i = 0; i < bn; ++i) --------------------------//357
--- if (inPolygon(b[i],a,an) || onPolygon(b[i],a,an)) ----//358
----- ans[size++] = b[i]; -------------------------------//359
- for (int i = 0, I = an - 1; i < an; I = i++) ----------//35a
--- for (int j = 0, J = bn - 1; j < bn; J = j++) { -------//35b
----- try { ---------------------------------------------//35c
------- point p=line_inter(a[i],a[I],b[j],b[J],true); ----//35d
------- ans[size++] = p; --------------------------------//35e
----- } catch (exception ex) {} -------------------------//35f
--- } ---------------------------------------------------//360
- size = convex_hull(ans, size); ------------------------//361
- return vector<point>(ans, ans + size); ----------------//362
} -------------------------------------------------------//363
```

9.15. **Pick's Theorem for Lattice Points.** Count points with integer coordinates inside and on the boundary of a polygon in $O(n)$ using Pick's theorem: Area $= I + B/2 - 1$.

```
int interior(point p[], int n) --------------------------//3f6
- {return area(p,n) - boundary(p,n) / 2 + 1;} -----------//3f7
int boundary(point p[], int n) { ------------------------//3f8
- int ans = 0; ------------------------------------------//3f9
- for (int i = 0, j = n - 1; i < n; j = i++) ------------//3fa
--- ans += gcd(p[i].x - p[j].x, p[i].y - p[j].y); --------//3fb
- return ans;} ------------------------------------------//3fc
```

9.16. **Minimum Enclosing Circle.** Get the minimum bounding ball that encloses a set of points (2D or 3D) in Θn.

```
pair<point, double> bounding_ball(point p[], int n){ -----//3e3
- random_shuffle(p, p + n); -----------------------------//3e4
- point center(0, 0); double radius = 0; ----------------//3e5
- for (int i = 0; i < n; ++i) { -------------------------//3e6
--- if (dist(center, p[i]) > radius + EPS) { ------------//3e7
----- center = p[i]; radius = 0; ------------------------//3e8
----- for (int j = 0; j < i; ++j) -----------------------//3e9
------- if (dist(center, p[j]) > radius + EPS) { --------//3ea
--------- center.x = (p[i].x + p[j].x) / 2; -------------//3eb
--------- center.y = (p[i].y + p[j].y) / 2; -------------//3ec
--------- // center.z = (p[i].z + p[j].z) / 2; ----------//3ed
--------- radius = dist(center, p[i]); // midpoint -------//3ee
--------- for (int k = 0; k < j; ++k) -------------------//3ef
----------- if (dist(center, p[k]) > radius + EPS) { -----//3f0
------------- center=circumcenter(p[i], p[j], p[k]); -----//3f1
------------- radius = dist(center, p[i]); --------------//3f2
---------- }}}} -----------------------------------------//3f3
- return make_pair(center, radius); ---------------------//3f4
} -------------------------------------------------------//3f5
```

9.17. **Shamos Algorithm.** Solve for the polygon diameter in $O(n \log n)$.

```
double shamos(point p[], int n) { -----------------------//42b
- point *h = new point[n+1]; copy(p, p + n, h); ----------//42c
- int k = convex_hull(h, n); if (k <= 2) return 0; -------//42d
- h[k] = h[0]; double d = HUGE_VAL; ---------------------//42e
- for (int i = 0, j = 1; i < k; ++i) { ------------------//42f
--- while (distPtLine(h[j+1], h[i], h[i+1]) >= ----------//430
---------- distPtLine(h[j], h[i], h[i+1])) { ------------//431
----- j = (j + 1) % k; ----------------------------------//432
--- } ---------------------------------------------------//433
--- d = min(d, distPtLine(h[j], h[i], h[i+1])); ----------//434
- } return d; } -----------------------------------------//435
```

9.18. **kD Tree.** Get the k-nearest neighbors of a point within pruned radius in $O(k \log k \log n)$.

```
#define cpoint const point& --------------------------------//39e
bool cmpx(cpoint a, cpoint b) {return a.x < b.x;} --------//39f
bool cmpy(cpoint a, cpoint b) {return a.y < b.y;} --------//3a0
struct KDTree { -----------------------------------------//3a1
- KDTree(point p[],int n): p(p), n(n) {build(0,n);} ------//3a2
- priority_queue< pair<double, point*> > pq; -------------//3a3
- point *p; int n, k; double qx, qy, prune; --------------//3a4
- void build(int L, int R, bool dvx=false) { ------------//3a5
--- if (L >= R) return; ---------------------------------//3a6
--- int M = (L + R) / 2; --------------------------------//3a7
--- nth_element(p + L, p + M, p + R, dvx?cmpx:cmpy); -----//3a8
--- build(L, M, !dvx); build(M + 1, R, !dvx); ------------//3a9
- } -----------------------------------------------------//3aa
- void dfs(int L, int R, bool dvx) { --------------------//3ab
--- if (L >= R) return; ---------------------------------//3ac
--- int M = (L + R) / 2; --------------------------------//3ad
--- double dx = qx - p[M].x, dy = qy - p[M].y; ----------//3ae
--- double delta = dvx ? dx : dy; -----------------------//3af
--- double D = dx * dx + dy * dy; -----------------------//3b0
--- if(D<=prune && (pq.size()<k||D<pq.top().first)){ -----//3b1
----- pq.push(make_pair(D, &p[M])); ---------------------//3b2
----- if (pq.size() > k) pq.pop(); ----------------------//3b3
--- } ---------------------------------------------------//3b4
--- int nL = L, nR = M, fL = M + 1, fR = R; --------------//3b5
--- if (delta > 0) {swap(nL, fL); swap(nR, fR);} ---------//3b6
--- dfs(nL, nR, !dvx); ----------------------------------//3b7
--- D = delta * delta; ----------------------------------//3b8
--- if (D<=prune && (pq.size()<k||D<pq.top().first)) -----//3b9
--- dfs(fL, fR, !dvx); ----------------------------------//3ba
- } -----------------------------------------------------//3bb
- // returns k nearest neighbors of (x, y) in tree -------//3bc
- // usage: vector<point> ans = tree.knn(x, y, 2); -------//3bd
- vector<point> knn(double x, double y, -----------------//3be
------------------- int k=1, double r=-1) { -------------//3bf
--- qx=x; qy=y; this->k=k; prune=r<0?HUGE_VAL:r*r; -------//3c0
--- dfs(0, n, false); vector<point> v; -------------------//3c1
--- while (!pq.empty()) { -------------------------------//3c2
----- v.push_back(*pq.top().second); --------------------//3c3
----- pq.pop(); -----------------------------------------//3c4
--- } reverse(v.begin(), v.end()); ----------------------//3c5
--- return v; -------------------------------------------//3c6
- } -----------------------------------------------------//3c7
}; ------------------------------------------------------//3c8
```

9.19. **Line Sweep (Closest Pair).** Get the closest pair distance of a set of points in $O(n \log n)$ by sweeping a line and keeping a bounded rectangle. Modifiable for other metrics such as Minkowski and Manhattan distance. For external point queries, see kD Tree.

```
bool cmpy(const point& a, const point& b) ---------------//327
- {return a.y < b.y;} -----------------------------------//328
double closest_pair_sweep(point p[], int n) { -----------//329
- if (n <= 1) return HUGE_VAL; --------------------------//32a
```

```
- sort(p, p + n, cmpy);
- set<point> box; box.insert(p[0]);
- double best = 1e13; // infinity, but not HUGE_VAL
- for (int L = 0, i = 1; i < n; ++i) {
--- while(L < i && p[i].y - p[L].y > best)
----- box.erase(p[L++]);
--- point bound(p[i].x - best, p[i].y - best);
--- set<point>::iterator it= box.lower_bound(bound);
--- while (it != box.end() && p[i].x+best >= it->x){
----- double dx = p[i].x - it->x;
----- double dy = p[i].y - it->y;
----- best = min(best, sqrt(dx*dx + dy*dy));
----- ++it;
--- }
--- box.insert(p[i]);
- } return best;
}
```

9.20. **Line upper/lower envelope.** To find the upper/lower envelope of a collection of lines $a_i + b_i x$, plot the points (b_i, a_i), add the point $(0, \pm\infty)$ (depending on if upper/lower envelope is desired), and then find the convex hull.

9.21. **Formulas.** Let $a = (a_x, a_y)$ and $b = (b_x, b_y)$ be two-dimensional vectors.

- $a \cdot b = |a||b| \cos\theta$, where θ is the angle between a and b.
- $a \times b = |a||b| \sin\theta$, where θ is the signed angle between a and b.
- $a \times b$ is equal to the area of the parallelogram with two of its sides formed by a and b. Half of that is the area of the triangle formed by a and b.
- The line going through a and b is $Ax+By = C$ where $A = b_y - a_y$, $B = a_x - b_x$, $C = Aa_x + Ba_y$.
- Two lines $A_1x + B_1y = C_1$, $A_2x + B_2y = C_2$ are parallel iff. $D = A_1B_2 - A_2B_1$ is zero. Otherwise their unique intersection is $(B_2C_1 - B_1C_2, A_1C_2 - A_2C_1)/D$.
- **Euler's formula:** $V - E + F = 2$
- Side lengths a, b, c can form a triangle iff. $a + b > c$, $b + c > a$ and $a + c > b$.
- Sum of internal angles of a regular convex n-gon is $(n - 2)\pi$.
- **Law of sines:** $\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$
- **Law of cosines:** $b^2 = a^2 + c^2 - 2ac\cos B$
- Internal tangents of circles $(c_1, r_1), (c_2, r_2)$ intersect at $(c_1r_2 + c_2r_1)/(r_1 + r_2)$, external intersect at $(c_1r_2 - c_2r_1)/(r_1 + r_2)$.

10. Other Algorithms

10.1. **Coordinate Compression.**

10.2. **2SAT.**

10.3. **Nth Permutation.**

10.4. **Floyd's Cycle-Finding.**

10.5. **Simulated Annealing.**

10.6. **Hexagonal Grid Algorithms.**

11. Useful Information (CLEAN THIS UP!!)

12. Misc

12.1. Debugging Tips.

- Stack overflow? Recursive DFS on tree that is actually a long path?
- Floating-point numbers
  - Getting `NaN`? Make sure `acos` etc. are not getting values out of their range (perhaps `1+eps`).
  - Rounding negative numbers?
  - Outputting in scientific notation?
- Wrong Answer?
  - Read the problem statement again!
  - Are multiple test cases being handled correctly? Try repeating the same test case many times.
  - Integer overflow?
  - Think very carefully about boundaries of all input parameters
  - Try out possible edge cases:
    - $n = 0, n = -1, n = 1, n = 2^{31} - 1$ or $n = -2^{31}$
    - List is empty, or contains a single element
    - n is even, n is odd
    - Graph is empty, or contains a single vertex
    - Graph is a multigraph (loops or multiple edges)
    - Polygon is concave or non-simple
  - Is initial condition wrong for small cases?
  - Are you sure the algorithm is correct?
  - Explain your solution to someone.
  - Are you using any functions that you don't completely understand? Maybe STL functions?
  - Maybe you (or someone else) should rewrite the solution?
  - Can the input line be empty?
- Run-Time Error?
  - Is it actually Memory Limit Exceeded?

12.2. Solution Ideas.

- Dynamic Programming
  - Parsing CFGs: CYK Algorithm
  - Drop a parameter, recover from others
  - Swap answer and a parameter
  - When grouping: try splitting in two
  - 2^k trick
  - When optimizing
    - Convex hull optimization
      - $\text{dp}[i] = \min_{j<i}\{\text{dp}[j] + b[j] \times a[i]\}$
      - $b[j] \geq b[j+1]$
      - optionally $a[i] \leq a[i+1]$
      - $O(n^2)$ to $O(n)$
    - Divide and conquer optimization
      - $\text{dp}[i][j] = \min_{k<j}\{\text{dp}[i-1][k] + C[k][j]\}$
      - $A[i][j] \leq A[i][j+1]$
      - $O(kn^2)$ to $O(kn \log n)$
      - sufficient: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$, $a \leq b \leq c \leq d$ (QI)
    - Knuth optimization
      - $\text{dp}[i][j] = \min_{i<k<j}\{\text{dp}[i][k] + \text{dp}[k][j] + C[i][j]\}$
      - $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
      - $O(n^3)$ to $O(n^2)$
      - sufficient: QI and $C[b][c] \leq C[a][d]$, $a \leq b \leq c \leq d$
- Greedy
- Randomized
- Optimizations
  - Use bitset (/64)
  - Switch order of loops (cache locality)
- Process queries offline
  - Mo's algorithm
- Square-root decomposition
- Precomputation
- Efficient simulation
  - Mo's algorithm
  - Sqrt decomposition
  - Store 2^k jump pointers
- Data structure techniques
  - Sqrt buckets
  - Store 2^k jump pointers
  - 2^k merging trick
- Counting
  - Inclusion-exclusion principle
  - Generating functions
- Graphs
  - Can we model the problem as a graph?
  - Can we use any properties of the graph?
  - Strongly connected components
  - Cycles (or odd cycles)
  - Bipartite (no odd cycles)
    - Bipartite matching
    - Hall's marriage theorem
    - Stable Marriage
  - Cut vertex/bridge
  - Biconnected components
  - Degrees of vertices (odd/even)
  - Trees
    - Heavy-light decomposition
    - Centroid decomposition
    - Least common ancestor
    - Centers of the tree
  - Eulerian path/circuit
  - Chinese postman problem
  - Topological sort
  - (Min-Cost) Max Flow
  - Min Cut
    - Maximum Density Subgraph
  - Huffman Coding
  - Min-Cost Arborescence
  - Steiner Tree
  - Kirchoff's matrix tree theorem
  - Prüfer sequences
  - Lovász Toggle
  - Look at the DFS tree (which has no cross-edges)
  - Is the graph a DFA or NFA?
    - Is it the Synchronizing word problem?
- Mathematics
  - Is the function multiplicative?
  - Look for a pattern
  - Permutations
    - Consider the cycles of the permutation
  - Functions
    - Sum of piecewise-linear functions is a piecewise-linear function
    - Sum of convex (concave) functions is convex (concave)
  - Modular arithmetic
    - Chinese Remainder Theorem
    - Linear Congruence
  - Sieve
  - System of linear equations
  - Values too big to represent?
    - Compute using the logarithm
    - Divide everything by some large value
  - Linear programming
    - Is the dual problem easier to solve?
  - Can the problem be modeled as a different combinatorial problem? Does that simplify calculations?
- Logic
  - 2-SAT
  - XOR-SAT (Gauss elimination or Bipartite matching)
- Meet in the middle
- Only work with the smaller half ($\log(n)$)
- Strings
  - Trie (maybe over something weird, like bits)
  - Suffix array
  - Suffix automaton (+DP?)
  - Aho-Corasick
  - eerTree
  - Work with $S + S$
- Hashing
- Euler tour, tree to array
- Segment trees
  - Lazy propagation
  - Persistent
  - Implicit
  - Segment tree of X
- Geometry
  - Minkowski sum (of convex sets)
  - Rotating calipers
  - Sweep line (horizontally or vertically?)
  - Sweep angle
  - Convex hull
- Fix a parameter (possibly the answer).
- Are there few distinct values?
- Binary search
- Sliding Window (+ Monotonic Queue)
- Computing a Convolution? Fast Fourier Transform
- Computing a 2D Convolution? FFT on each row, and then on each column
- Exact Cover (+ Algorithm X)
- Cycle-Finding
- What is the smallest set of values that identify the solution? The cycle structure of the permutation? The powers of primes in the factorization?
- Look at the complement problem

- Minimize something instead of maximizing
- Immediately enforce necessary conditions. (All values greater than 0? Initialize them all to 1)
- Add large constant to negative numbers to make them positive
- Counting/Bucket sort

13. Formulas

- **Legendre symbol:** $\left(\frac{a}{b}\right) = a^{(b-1)/2} \pmod b$, b odd prime.
- **Heron's formula:** A triangle with side lengths a, b, c has area $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{a+b+c}{2}$.
- **Pick's theorem:** A polygon on an integer grid strictly containing i lattice points and having b lattice points on the boundary has area $i + \frac{b}{2} - 1$. (Nothing similar in higher dimensions)
- **Euler's totient:** The number of integers less than n that are coprime to n are $n\prod_{p\mid n}\left(1-\frac{1}{p}\right)$ where each p is a distinct prime factor of n.
- **König's theorem:** In any bipartite graph $G = (L \cup R, E)$, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover. Let U be the set of unmatched vertices in L, and Z be the set of vertices that are either in U or are connected to U by an alternating path. Then $K = (L \setminus Z) \cup (R \cap Z)$ is the minimum vertex cover.
- A minumum Steiner tree for n vertices requires at most $n-2$ additional Steiner vertices.
- The number of vertices of a graph is equal to its minimum vertex cover number plus the size of a maximum independent set.
- **Lagrange polynomial** through points $(x_0, y_0), \ldots, (x_k, y_k)$ is $L(x) = \sum_{j=0}^{k} y_j \prod_{\substack{0\le m\le k \\ m\ne j}} \frac{x-x_m}{x_j-x_m}$
- **Hook length formula:** If λ is a Young diagram and $h_\lambda(i,j)$ is the hook-length of cell (i,j), then then the number of Young tableux $d_\lambda = n!/\prod h_\lambda(i,j)$.
- **Möbius inversion formula:** If $f(n) = \sum_{d\mid n} g(d)$, then $g(n) = \sum_{d\mid n}\mu(d)f(n/d)$. If $f(n) = \sum_{m=1}^{n} g(\lfloor n/m\rfloor)$, then $g(n) = \sum_{m=1}^{n}\mu(m)f(\lfloor\frac{n}{m}\rfloor)$.
- #primitive pythagorean triples with hypotenuse $< n$ approx $n/(2\pi)$.
- **Frobenius Number:** largest number which can't be expressed as a linear combination of numbers $a_1, \ldots, a_n$ with non-negative coefficients. $g(a_1, a_2) = a_1a_2 - a_1 - a_2$, $N(a_1, a_2) = (a_1-1)(a_2-1)/2$. $g(d\cdot a_1, d\cdot a_2, a_3) = d\cdot g(a_1, a_2, a_3) + a_3(d-1)$. An integer $x > (\max_i a_i)^2$ can be expressed in such a way iff. $x \mid \gcd(a_1, \ldots, a_n)$.

13.1. Physics.

- **Snell's law:** $\frac{\sin\theta_1}{v_1} = \frac{\sin\theta_2}{v_2}$

13.2. Markov Chains.

A Markov Chain can be represented as a weighted directed graph of states, where the weight of an edge represents the probability of transitioning over that edge in one timestep. Let $P^{(m)} = (p_{ij}^{(m)})$ be the probability matrix of transitioning from state i to state j in m timesteps, and note that $P^{(1)}$ is the adjacency matrix of the graph. **Chapman-Kolmogorov:** $p_{ij}^{(m+n)} = \sum_k p_{ik}^{(m)} p_{kj}^{(n)}$. It follows that $P^{(m+n)} = P^{(m)}P^{(n)}$ and $P^{(m)} = P^m$. If $p^{(0)}$ is the initial probability distribution (a vector), then $p^{(0)}P^{(m)}$ is the probability distribution after m timesteps.

The return times of a state i is $R_i = \{m \mid p_{ii}^{(m)} > 0\}$, and i is *aperiodic* if $\gcd(R_i) = 1$. A MC is aperiodic if any of its vertices is aperiodic. A MC is *irreducible* if the corresponding graph is strongly connected.

A distribution π is stationary if $\pi P = \pi$. If MC is irreducible then $\pi_i = 1/\mathbb{E}[T_i]$, where T_i is the expected time between two visits at i. π_j/π_i is the expected number of visits at j in between two consecutive visits at i. A MC is *ergodic* if $\lim_{m\to\infty} p^{(0)}P^m = \pi$. A MC is ergodic iff. it is irreducible and aperiodic.

A MC for a random walk in an undirected weighted graph (unweighted graph can be made weighted by adding 1-weights) has $p_{uv} = w_{uv}/\sum_x w_{ux}$. If the graph is connected, then $\pi_u = \sum_x w_{ux}/\sum_v\sum_x w_{vx}$. Such a random walk is aperiodic iff. the graph is not bipartite.

An *absorbing* MC is of the form $P = \begin{pmatrix} Q & R \\ 0 & I_r \end{pmatrix}$. Let $N = \sum_{m=0}^{\infty} Q^m = (I_t - Q)^{-1}$. Then, if starting in state i, the expected number of steps till absorption is the i-th entry in $N1$. If starting in state i, the probability of being absorbed in state j is the (i,j)-th entry of NR.

Many problems on MC can be formulated in terms of a system of recurrence relations, and then solved using Gaussian elimination.

13.3. Burnside's Lemma.

Let G be a finite group that acts on a set X. For each g in G let X^g denote the set of elements in X that are fixed by g. Then the number of orbits

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|$$

$$Z(S_n) = \frac{1}{n}\sum_{l=1}^{n} a_l Z(S_{n-l})$$

13.4. Bézout's identity.

If (x, y) is any solution to $ax + by = d$ (e.g. found by the Extended Euclidean Algorithm), then all solutions are given by

$$\left(x + k\frac{b}{\gcd(a,b)}, y - k\frac{a}{\gcd(a,b)}\right)$$

13.5. Misc.

13.5.1. *Determinants and PM.*

$$det(A) = \sum_{\sigma\in S_n}\operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$perm(A) = \sum_{\sigma\in S_n}\prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$\begin{aligned} pf(A) &= \frac{1}{2^n n!}\sum_{\sigma\in S_{2n}}\operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{\sigma(2i-1),\sigma(2i)} \\ &= \sum_{M\in \mathrm{PM}(n)}\operatorname{sgn}(M)\prod_{(i,j)\in M} a_{i,j} \end{aligned}$$

13.5.2. *BEST Theorem.*

Count directed Eulerian cycles. Number of OST given by Kirchoff's Theorem (remove r/c with root) $\#\mathrm{OST}(G,r)\cdot\prod_v(d_v-1)!$

13.5.3. *Primitive Roots.*

Only exists when n is $2, 4, p^k, 2p^k$, where p odd prime. Assume n prime. Number of primitive roots $\phi(\phi(n))$ Let g be primitive root. All primitive roots are of the form g^k where $k, \phi(p)$ are coprime.
k-roots: $g^{i\cdot\phi(n)/k}$ for $0 \le i < k$

13.5.4. *Sum of primes.*

For any multiplicative f:

$$S(n,p) = S(n,p-1) - f(p)\cdot(S(n/p,p-1) - S(p-1,p-1))$$

13.5.5. *Floor.*

$$\lfloor\lfloor x/y\rfloor/z\rfloor = \lfloor x/(yz)\rfloor$$

$$x\%y = x - y\lfloor x/y\rfloor$$