

KFC

AdMU Progar

18/03/2021

CONTENTS

1. Data Structures
2. DP
3. Geometry
4. Graphs
5. Math I- Algebra
6. Math II - Combinatorics
7. Math III - Number Theory
8. Math IV - Numerical Methods
9. Strings

1. DATA STRUCTURES

1.1. Fenwick Tree.

```
struct fenwick {
- vi ar;
- fenwick(vi &ar) : ar(_ar.size(), 0) {
-   for (int i = 0; i < ar.size(); ++i) {
-       ar[i] += _ar[i];
-       int j = i | (i+1);
-       if (j < ar.size())
-           ar[j] += ar[i]; } }
- int sum(int i) {
-   int res = 0;
-   for (; i >= 0; i = (i & (i+1)) - 1)
-       res += ar[i];
-   return res; }
- int sum(int i, int j) { return sum(j) - sum(i-1); }
- void add(int i, int val) {
-   for (; i < ar.size(); i |= i+1)
-       ar[i] += val; }
- int get(int i) {
-   int res = ar[i];
-   if (i) {
-       int lca = (i & (i+1)) - 1;
-       for (--i; i != lca; i = (i&(i+1))-1)
-           res -= ar[i]; }
-   return res; }
- void set(int i, int val) { add(i, -get(i) + val); }
- // range update, point query //
- void add(int i, int j, int val) {
-   add(i, val); add(j+1, -val); }
- int get1(int i) { return sum(i); } };
```

1.2. Leq Counter.

```
1.2.1. Leq Counter Array.
#include "segtree.cpp"
struct LeqCounter {
- segtree **roots;
- LeqCounter(int *ar, int n) {
-   std::vector<ii> nums;
-   for (int i = 0; i < n; ++i)
-       nums.push_back({ar[i], i});
-   std::sort(nums.begin(), nums.end());
-   roots = new segtree*[n];
-   roots[0] = new segtree(0, n);
-   int prev = 0;
-   for (ii &e : nums) {
-       for (int i = prev+1; i < e.first; ++i)
-           roots[i] = roots[prev];
-       roots[e.first] = roots[prev]->update(e.second, 1);
-       prev = e.first; }
-   for (int i = prev+1; i < n; ++i)
-       roots[i] = roots[prev]; }
- int count(int i, int j, int x) {
-   return roots[x]->query(i, j); } };
```

1.2.2. Leq Counter Map.

```
struct LeqCounter {
- std::map<int, segtree*> roots;
- std::set<int> neg_nums;
- LeqCounter(int *ar, int n) {
-   std::vector<ii> nums;
-   for (int i = 0; i < n; ++i) {
-       nums.push_back({ar[i], i});
-       neg_nums.insert(-ar[i]);
-   }
-   std::sort(nums.begin(), nums.end());
-   roots[0] = new segtree(0, n);
-   int prev = 0;
-   for (ii &e : nums) {
-       roots[e.first] = roots[prev]->update(e.second, 1);
-       prev = e.first; } }
- int count(int i, int j, int x) {
-   auto it = neg_nums.lower_bound(-x);
-   if (it == neg_nums.end()) return 0;
-   return roots[*it]->query(i, j); } };
```

1.3. Misof Tree. A simple tree data structure for inserting, erasing, and querying the nth largest element.

```
#define BITS 15
struct misof_tree {
- int cnt[BITS][1<<BITS];
- misof_tree() { memset(cnt, 0, sizeof(cnt)); }
- void insert(int x) {
-   for (int i = 0; i < BITS; cnt[i+1][x]++, x >= 1); }
- void erase(int x) {
-   for (int i = 0; i < BITS; cnt[i+1][x]--, x >= 1); }
- int nth(int n) {
-   int res = 0;
-   for (int i = BITS-1; i >= 0; i--)
```

```
---- if (cnt[i][res <= 1] <= n) n -= cnt[i][res], res |= 1;
- return res; } };
```

1.4. Mo's Algorithm.

```
struct query {
- int id, l, r; ll hilbert_index;
- query(int id, int l, int r) : id(id), l(l), r(r) {
-   hilbert_index = hilbert_order(l, r, LOGN, 0); }
- ll hilbert_order(int x, int y, int pow, int rotate) {
-   if (pow == 0) return 0;
-   int hpow = 1 << (pow-1);
-   int seg = ((x<hpow) ? ((y<hpow)?0:3) : ((y<hpow)?1:2));
-   seg = (seg + rotate) & 3;
-   const int rotate_delta[4] = {3, 0, 0, 1};
-   int nx = x & (x ^ hpow), ny = y & (y ^ hpow);
-   int nrot = (rotate + rotate_delta[seg]) & 3;
-   ll sub_sq_size = ll(1) << (2*pow - 2);
-   ll ans = seg * sub_sq_size;
-   ll add = hilbert_order(nx, ny, pow-1, nrot);
-   ans += (seg==1 || seg==2) ? add : (sub_sq_size-add-1);
-   return ans; }
- bool operator<(const query& other) const {
-   return this->hilbert_index < other.hilbert_index; } };
std::vector<query> queries;
for(const query &q : queries) { // [l,r] inclusive
-   for(; r > q.r; r--) update(r, -1);
-   for(r = r+1; r <= q.r; r++) update(r);
-   r--;
-   for(; l < q.l; l++) update(l, -1);
-   for(l = l-1; l >= q.l; l--) update(l);
-   l++; }
```

1.5. Ordered Statistics Tree.

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T>
using index_set = tree<T, null_type, std::less<T>,
splay_tree_tag, tree_order_statistics_node_update>;
// indexed_set<int> t; t.insert(...);
// t.find_by_order(index); // 0-based
// t.order_of_key(key);
```

1.6. Segment Tree.

1.6.1. Recursive, Point-update Segment Tree.

1.6.2. 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; } };

```

1.6.3. 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);
} };

```

1.6.4. 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); } } };

```

1.6.5. 2D Segment Tree.

```

struct segtree_2d {
- int n, m, **ar;
- segtree_2d(int n, int m) {
-   this->n = n;   this->m = m;
-   ar = new int[n];
-   for (int i = 0; i < n; ++i) {
-     ar[i] = new int[m];
-     for (int j = 0; j < m; ++j)
-       ar[i][j] = 0; } }
- void update(int x, int y, int v) {
-   ar[x + n][y + m] = v;
-   for (int i = x + n; i > 0; i >>= 1) {
-     for (int j = y + m; j > 0; j >>= 1) {

```

```

-       ar[i>>1][j] = min(ar[i][j], ar[i^1][j]);
-       ar[i][j>>1] = min(ar[i][j], ar[i][j^1]);
-     } } // just call update one by one to build
- int query(int x1, int x2, int y1, int y2) {
-   int s = INF;
-   if (~x2) for (int a=x1+n, b=x2+n+1; a<b; a>>=1, b>>=1) {
-     if (a & 1) s = min(s, query(a++, -1, y1, y2));
-     if (b & 1) s = min(s, query(--b, -1, y1, y2));
-   } else for (int a=y1+m, b=y2+m+1; a<b; a>>=1, b>>=1) {
-     if (a & 1) s = min(s, ar[x1][a++]);
-     if (b & 1) s = min(s, ar[x1][--b]);
-   } return s; } };

```

1.6.6. Persistent Segment Tree.

```

struct segtree {
- int i, j, val;
- 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;
-   } }
- segtree(int i, int j, segtree *l, segtree *r, int val) :
-   i(i), j(j), l(l), r(r), val(val) {}
- segtree* update(int _i, int _val) {
-   if (_i <= i and j <= _i)
-     return new segtree(i, j, l, r, val + _val);
-   else if (_i < i or j < _i)
-     return this;
-   else {
-     segtree *nl = l->update(_i, _val);
-     segtree *nr = r->update(_i, _val);
-     return new segtree(i, j, nl, nr, nl->val + nr->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); } };

```

1.7. Sparse Table.

1.7.1. 1D Sparse Table.

```

int lg[MAXN+1], spt[20][MAXN];
void build(vi &arr, int n) {
- lg[0] = lg[1] = 0;
- for (int i = 2; i <= n; ++i) lg[i] = lg[i>>1] + 1;
- for (int i = 0; i < n; ++i) spt[0][i] = arr[i];
- for (int j = 0; (2 <= j) <= n; ++j)
-   for (int i = 0; i + (2 <= j) <= n; ++i)
-     spt[j+1][i] = std::min(spt[j][i], spt[j][i+(1<<j)]); }

```

```
int query(int a, int b) {
- int k = lg[b-a+1], ab = b - (1<<k) + 1;
- return std::min(spt[k][a], spt[k][ab]); }
```

1.7.2. 2D Sparse Table .

```
const int N = 100, LGN = 20;
int lg[N], A[N][N], st[LGN][LGN][N][N];
void build(int n, int m) {
- for(int k=2; k<=std::max(n,m); ++k) lg[k] = lg[k>=1]+1;
- for(int i = 0; i < n; ++i)
-   for(int j = 0; j < m; ++j)
-     st[0][0][i][j] = A[i][j];
- for(int bj = 0; (2 << bj) <= m; ++bj)
-   for(int j = 0; j + (2 << bj) <= m; ++j)
-     for(int i = 0; i < n; ++i)
-       st[0][bj+1][i][j] =
-         std::max(st[0][bj][i][j],
-                   st[0][bj][i][j + (1 << bj)]);
- for(int bi = 0; (2 << bi) <= n; ++bi)
-   for(int i = 0; i + (2 << bi) <= n; ++i)
-     for(int j = 0; j < m; ++j)
-       st[bi+1][0][i][j] =
-         std::max(st[bi][0][i][j],
-                   st[bi][0][i + (1 << bi)][j]);
- for(int bi = 0; (2 << bi) <= n; ++bi)
-   for(int i = 0; i + (2 << bi) <= n; ++i)
-     for(int bj = 0; (2 << bj) <= m; ++bj)
-       for(int j = 0; j + (2 << bj) <= m; ++j) {
-         int ik = i + (1 << bi);
-         int jk = j + (1 << bj);
-         st[bi+1][bj+1][i][j] =
-           std::max(std::max(st[bi][bj][i][j],
-                               st[bi][bj][ik][j]),
-                     std::max(st[bi][bj][i][jk],
-                               st[bi][bj][ik][jk])); } }

int query(int x1, int x2, int y1, int y2) {
- int kx = lg[x2 - x1 + 1], ky = lg[y2 - y1 + 1];
- int x12 = x2 - (1<<kx) + 1, y12 = y2 - (1<<ky) + 1;
- return std::max(std::max(st[kx][ky][x1][y1],
-                           st[kx][ky][x1][y12]),
-                 std::max(st[kx][ky][x12][y1],
-                           st[kx][ky][x12][y12])); }
```

1.8. Splay Tree .

```
struct node *null;
struct node {
- node *left, *right, *parent;
- bool reverse; int size, value;
- node& get(int d) {return d == 0 ? left : right;}
- node(int v=0): reverse(0), size(0), value(v) {
- left = right = parent = null ? null : this; } };
struct SplayTree {
- node *root;
- SplayTree(int arr[] = NULL, int n = 0) {
-   if (!null) null = new node();
-   root = build(arr, n); }
```

```
- node* build(int arr[], int n) {
-   if (n == 0) return null;
-   int mid = n >> 1;
-   node *p = new node(arr ? arr[mid] : 0);
-   link(p, build(arr, mid), 0);
-   link(p, build(arr? arr+mid+1 : NULL, n-mid-1), 1);
-   pull(p); return p; }
- void pull(node *p) {
-   p->size = p->left->size + p->right->size + 1; }
- void push(node *p) {
-   if (p != null && p->reverse) {
-     swap(p->left, p->right);
-     p->left->reverse ^= 1;
-     p->right->reverse ^= 1;
-     p->reverse ^= 1; } }
- void link(node *p, node *son, int d) {
-   p->get(d) = son;
-   son->parent = p; }
- int dir(node *p, node *son) {
-   return p->left == son ? 0 : 1; }
- void rotate(node *x, int d) {
-   node *y = x->get(d), *z = x->parent;
-   link(x, y->get(d ^ 1), d);
-   link(y, x, d ^ 1);
-   link(z, y, dir(z, x));
-   pull(x); pull(y); }
- node* splay(node *p) {
-   while (p->parent != null) {
-     node *m = p->parent, *g = m->parent;
-     push(g); push(m); push(p);
-     int dm = dir(m, p), dg = dir(g, m);
-     if (g == null) rotate(m, dm);
-     else if (dm == dg) rotate(g, dg), rotate(m, dm);
-     else rotate(m, dm), rotate(g, dg);
-   } return root = p; }
- node* get(int k) {
-   node *p = root;
-   while (push(p), p->left->size != k) {
-     if (k < p->left->size) p = p->left;
-     else k -= p->left->size + 1, p = p->right; }
-   return p == null ? null : splay(p); }
- void split(node *&r, int k) {
-   if (k == 0) { r = root; root = null; return; }
-   r = get(k - 1)->right;
-   root->right = r->parent = null;
-   pull(root); }
- void merge(node *r) {
-   if (root == null) {root = r; return;}
-   link(get(root->size - 1), r, 1);
-   pull(root); }
- void assign(int k, int val) {
-   get(k)->value = val; pull(root); }
- void reverse(int L, int R) {
-   node *m, *r; split(r, R + 1); split(m, L);
-   m->reverse ^= 1; push(m); merge(m); merge(r); }
- node* insert(int k, int v) {
```

```
- node *r; split(r, k);
- node *p = new node(v); p->size = 1;
- link(root, p, 1); merge(r);
- return p; }
- void erase(int k) {
-   node *r, *m;
-   split(r, k + 1); split(m, k);
-   merge(r); delete m; } ;}
```

1.9. Treap.

1.9.1. Implicit Treap.

```
struct cartree {
- typedef struct _Node {
-   int node_val, subtree_val, delta, prio, size;
-   _Node *l, *r;
-   _Node(int val) : node_val(val), subtree_val(val),
-     delta(0), prio((rand()<<16)^rand()), size(1),
-     l(NULL), r(NULL) {}
-   ~_Node() { delete l; delete r; }
- } *Node;
- int get_subtree_val(Node v) {
-   return v ? v->subtree_val : 0; }
- int get_size(Node v) { return v ? v->size : 0; }
- void apply_delta(Node v, int delta) {
-   if (!v) return;
-   v->delta += delta;
-   v->node_val += delta;
-   v->subtree_val += delta * get_size(v); }
- void push_delta(Node v) {
-   if (!v) return;
-   apply_delta(v->l, v->delta);
-   apply_delta(v->r, v->delta);
-   v->delta = 0; }
- void update(Node v) {
-   if (!v) return;
-   v->subtree_val = get_subtree_val(v->l) + v->node_val
-     + get_subtree_val(v->r);
-   v->size = get_size(v->l) + 1 + get_size(v->r); }
- Node merge(Node l, Node r) {
-   push_delta(l); push_delta(r);
-   if (!l || !r) return l ? l : r;
-   if (l->size <= r->size) {
-     l->r = merge(l->r, r);
-     update(l);
-     return l;
-   } else {
-     r->l = merge(l, r->l);
-     update(r);
-     return r; } }
- void split(Node v, int key, Node &l, Node &r) {
-   push_delta(v);
-   l = r = NULL;
-   if (!v) return;
-   if (key <= get_size(v->l)) {
-     split(v->l, key, l, v->l);
-     r = v;
```

```

-- } else {
---- split(v->r, key - get_size(v->l) - 1, v->r, r);
---- l = v; }
-- update(v); }
- Node root;
public:
- cartree() : root(NULL) {}
- ~cartree() { delete root; }
- int get(Node v, int key) {
-- push_delta(v);
-- if (key < get_size(v->l))
-- return get(v->l, key);
-- else if (key > get_size(v->l))
-- return get(v->r, key - get_size(v->l) - 1);
-- return v->node_val; }
- int get(int key) { return get(root, key); }
- void insert(Node item, int key) {
-- Node l, r;
-- split(root, key, l, r);
-- root = merge(merge(l, item), r); }
- void insert(int key, int val) {
-- insert(new _Node(val), key); }
- void erase(int key) {
-- Node l, m, r;
-- split(root, key + 1, m, r);
-- split(m, key, l, m);
-- delete m;
-- root = merge(l, r); }
- int query(int a, int b) {
-- Node l1, r1;
-- split(root, b+1, l1, r1);
-- Node l2, r2;
-- split(l1, a, l2, r2);
-- int res = get_subtree_val(r2);
-- l1 = merge(l2, r2);
-- root = merge(l1, r1);
-- return res; }
- void update(int a, int b, int delta) {
-- Node l1, r1;
-- split(root, b+1, l1, r1);
-- Node l2, r2;
-- split(l1, a, l2, r2);
-- apply_delta(r2, delta);
-- l1 = merge(l2, r2);
-- root = merge(l1, r1); }
- int size() { return get_size(root); } };

```

1.9.2. Persistent Treap

1.10. Union Find.

```

struct union_find {
- vi p; union_find(int n) : p(n, -1) {}
- int find(int x) { return p[x] < 0 ? x : p[x] = find(p[x]); }
- bool unite(int x, int y) {
-- int xp = find(x), yp = find(y);
-- if (xp == yp) return false;
-- if (p[xp] > p[yp]) std::swap(xp,yp);

```

```

-- p[xp] += p[yp], p[yp] = xp; return true; }
- int size(int x) { return -p[find(x)]; } };

```

1.11. Unique Counter.

```

struct UniqueCounter {
- int *B; std::map<int, int> last; LeqCounter *leq_cnt;
- UniqueCounter(int *ar, int n) { // 0-index A[i]
-- B = new int[n+1];
-- B[0] = 0;
-- for (int i = 1; i <= n; ++i) {
---- B[i] = last[ar[i-1]];
---- last[ar[i-1]] = i; }
-- leq_cnt = new LeqCounter(B, n+1); }
- int count(int l, int r) {
-- return leq_cnt->count(l+1, r+1, l); } };

```

2. DP

2.1. Dynamic Convex Hull Trick.

```

// USAGE: hull.insert_line(m, b); hull.gety(x);
bool UPPER_HULL = true; // you can edit this
bool IS_QUERY = false, SPECIAL = false;
struct line {
- ll m, b; line(ll m=0, ll b=0): m(m), b(b) {}
- mutable std::multiset<line>::iterator it;
- const line *see(std::multiset<line>::iterator it)const;
- bool operator < (const line& k) const {
-- if (!IS_QUERY) return m < k.m;
-- if (!SPECIAL) {
---- ll x = k.m; const line *s = see(it);
---- if (!s) return 0;
-- return (b - s->b) < (x) * (s->m - m);
-- } else {
---- ll y = k.m; const line *s = see(it);
---- if (!s) return 0;
---- ll n1 = y - b, d1 = m;
---- ll n2 = b - s->b, d2 = s->m - m;
---- if (d1 < 0) n1 *= -1, d1 *= -1;
---- if (d2 < 0) n2 *= -1, d2 *= -1;
---- return (n1) * d2 > (n2) * d1; } } };
struct dynamic_hull : std::multiset<line> {
- bool bad(iterator y) {
-- iterator z = next(y);
-- if (y == begin()) {
---- if (z == end()) return 0;
---- return y->m == z->m && y->b <= z->b; }
-- iterator x = prev(y);
-- if (z == end()) return y->m == x->m && y->b <= x->b;
-- return (x->b - y->b)*(z->m - y->m)>=
-- (y->b - z->b)*(y->m - x->m); }
- iterator next(iterator y) {return ++y;}
- iterator prev(iterator y) {return --y;}
- void insert_line(ll m, ll b) {
-- IS_QUERY = false;
-- if (!UPPER_HULL) m *= -1;
-- iterator y = insert(line(m, b));
-- y->it = y; if (bad(y)) {erase(y); return;}

```

```

-- while (next(y) != end() && bad(next(y)))
-- erase(next(y));
-- while (y != begin() && bad(prev(y)))
-- erase(prev(y)); }
- ll gety(ll x) {
-- IS_QUERY = true; SPECIAL = false;
-- const line& L = *lower_bound(line(x, 0));
-- ll y = (L.m) * x + L.b;
-- return UPPER_HULL ? y : -y; }
- ll getx(ll y) {
-- IS_QUERY = true; SPECIAL = true;
-- const line& l = *lower_bound(line(y, 0));
-- return /*floor*/ ((y - l.b + l.m - 1) / l.m); }
} hull;
const line* line::see(std::multiset<line>::iterator it)
const {return ++it == hull.end() ? NULL : &*it;}

```

2.2. Divide and Conquer Optimization. For DP problems of the form

$$dp(i,j)=\min_{k\leq j}\{dp(i-1,k)+C(k,j)\}$$

where $C(k,j)$ is some cost function.

```

ll dp[G+1][N+1];
void solve_dp(int g, int k_L, int k_R, int n_L, int n_R) {
- int n_M = (n_L+n_R)/2;
- dp[g][n_M] = INF;
- int best_k = -1;
- for (int k = k_L; k <= n_M && k <= k_R; k++)
-- if (dp[g-1][k]+cost(k+1,n_M) < dp[g][n_M]) {
--- dp[g][n_M] = dp[g-1][k]+cost(k+1,n_M);
--- best_k = k; }
- if (n_L <= n_M-1)
-- solve_dp(g, k_L, best_k, n_L, n_M-1);
- if (n_M+1 <= n_R)
-- solve_dp(g, best_k, k_R, n_M+1, n_R); }

```

3. GEOMETRY

```

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

```

3.1. Dots and Cross Products.

```

double dot(point a, point b) {
- return a.x * b.x + a.y * b.y; } // + a.z * b.z;
double cross(point a, point b) {
- return a.x * b.y - a.y * b.x; }
double cross(point a, point b, point c) {
- return cross(a, b) + cross(b, c) + cross(c, a); }
double cross3D(point a, point b) {
- return point(a.x*b.y - a.y*b.x, a.y*b.z -
-- a.z*b.y, a.z*b.x - a.x*b.z); }

```

3.2. Angles and Rotations.

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

3.3. Spherical Coordinates.

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

3.4. Point Projection.

```
point proj(point p, point v) { -----
- // project point p onto a vector v (2D & 3D) -----
- return dot(p, v) / norm(v) * v; } -----
point projLine(point p, point a, point b) { -----
- // project point p onto line ab (2D & 3D) -----
- return a + dot(p-a, b-a) / norm(b-a) * (b-a); } -----
point projSeg(point p, point a, point b) { -----
- // project point p onto segment ab (2D & 3D) -----
- double s = dot(p-a, b-a) / norm(b-a); -----
- return a + min(1.0, max(0.0, s)) * (b-a); } -----
point projPlane(point p, double a, double b, -----
- double c, double d) { -----
- // project p onto plane ax+by+cz+d=0 (3D) -----
- // same as: o + p - project(p - o, n); -----
- double k = -d / (a*a + b*b + c*c); -----
- point o(a*k, b*k, c*k), n(a, b, c); -----
- point v(p.x-o.x, p.y-o.y, p.z-o.z); -----
- double s = dot(v, n) / dot(n, n); -----
- return point(o.x + p.x + s * n.x, o.y + -----
- p.y + s * n.y, o.z + p.z + s * n.z); } -----
```

3.5. Great Circle Distance.

```
double greatCircleDist(double lat1, double long1, -----
- double lat2, double long2, double R) { -----
- long1 *= PI / 180; lat1 *= PI / 180; // to radians -----
- long2 *= PI / 180; lat2 *= PI / 180; -----
- return R*acos(sin(lat1)*sin(lat2) + -----
- cos(lat1)*cos(lat2)*cos(abs(long1 - long2))); } -----
// another version, using actual (x, y, z) -----
double greatCircleDist(point a, point b) { -----
- return atan2(abs(cross3D(a, b)), dot3D(a, b)); } -----
```

3.6. Point/Line/Plane Distances.

```
double distPtLine(point p, double a, double b, double c) { ---
- // dist from point p to line ax+by+c=0 -----
- return abs(a*p.x + b*p.y + c) / sqrt(a*a + b*b);} -----
double distPtLine(point p, point a, point b) { -----
- // dist from point p to line ab -----
- return abs((a.y - b.y) * (p.x - a.x) + -----
- (b.x - a.x) * (p.y - a.y)) / -----
- hypot(a.x - b.x, a.y - b.y);} -----
double distPtPlane(point p, double a, double b, -----
- double c, double d) { -----
```

```
- // distance to 3D plane ax + by + cz + d = 0 -----
- return (a*p.x+b*p.y+c*p.z+d)/sqrt(a*a+b*b+c*c); } -----
/*! // distance between 3D lines AB & CD (untested) -----
double distLine3D(point A,point B,point C,point D){ -----
- point u = B - A, v = D - C, w = A - C; -----
- double a = dot(u, u), b = dot(u, v); -----
- double c = dot(v, v), d = dot(u, w); -----
- double e = dot(v, w), det = a*c - b*b; -----
- double s = det < EPS ? 0.0 : (b*e - c*d) / det; -----
- double t = det < EPS -----
- ? (b > c ? d/b : e/c) // parallel -----
- : (a*e - b*d) / det; -----
- point top = A + u * s, bot = w - A - v * t; -----
- return dist(top, bot); -----
} // dist<EPS: intersection */ -----
```

3.7. Intersections.

```
3.7.1. Line-Segment Intersection. Get intersection points of 2D -----
lines/segments  $\overline{ab}$  and  $\overline{cd}$ . -----
point null(HUGE_VAL, HUGE_VAL); -----
point line_inter(point a, point b, point c, -----
- point d, bool seg = false) { -----
- point ab(b.x - a.x, b.y - a.y); -----
- point cd(d.x - c.x, d.y - c.y); -----
- point ac(c.x - a.x, c.y - a.y); -----
- double D = -cross(ab, cd); // determinant -----
- double Ds = cross(cd, ac); -----
- double Dt = cross(ab, ac); -----
- if (abs(D) < EPS) { // parallel -----
- if (seg && abs(Ds) < EPS) { // collinear -----
- point p[] = {a, b, c, d}; -----
- sort(p, p + 4, [](point a, point b) { -----
- return a.x < b.x-EPS || -----
- (dist(a,b) < EPS && a.y < b.y-EPS); }); -----
- return dist(p[1], p[2]) < EPS ? p[1] : null; } -----
- return null; } -----
- double s = Ds / D, t = Dt / D; -----
- if (seg && (min(s,t)<-EPS||max(s,t)>1+EPS)) return null; ---
- return point(a.x + s * ab.x, a.y + s * ab.y); } -----
/* double A = cross(d-a, b-a), B = cross(c-a, b-a); -----
return (B*d - A*c)/(B - A); */ -----
```

```
3.7.2. 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, -----
- point a, point b) { -----
- point p = projLine(c, a, b); -----
- double d = abs(c - p); vector<point> ans; -----
- if (d > r + EPS); // none -----
- else if (d > r - EPS) ans.push_back(p); // tangent -----
- else if (d < EPS) { // diameter -----
- point v = r * (b - a) / abs(b - a); -----
- ans.push_back(c + v); -----
- ans.push_back(c - v); -----
- } else { -----
- double t = acos(d / r); -----
```

```
--- p = c + (p - c) * r / d; -----
--- ans.push_back(rotate(c, p, t)); -----
--- ans.push_back(rotate(c, p, -t)); -----
- } return ans; } -----
```

3.7.3. Circle-Circle Intersection.

```
std::vector<point> CC_intersection(point c1, -----
- double r1, point c2, double r2) { -----
- double d = dist(c1, c2); -----
- vector<point> ans; -----
- if (d < EPS) { -----
- if (abs(r1-r2) < EPS); // inf intersections -----
- } else if (r1 < EPS) { -----
- if (abs(d - r2) < EPS) ans.push_back(c1); -----
- } else { -----
- double s = (r1*r1 + d*d - r2*r2) / (2*r1*d); -----
- double t = acos(max(-1.0, min(1.0, s))); -----
- point mid = c1 + (c2 - c1) * r1 / d; -----
- ans.push_back(rotate(c1, mid, t)); -----
- if (abs(sin(t)) >= EPS) -----
- ans.push_back(rotate(c2, mid, -t)); -----
- } return ans; } -----
```

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

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

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

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

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

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

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

3.10. Convex Hull.

3.10.1. *2D 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 -----
bool xcmp(const point& a, const point& b) { -----
- return a.x < b.x || (a.x == b.x && a.y < b.y); }
int convex_hull(point p[], int n) { -----
- std::sort(p, p + n, xcmp); if (n <= 1) return n;
- int k = 0; point *h = new point[2 * n];
- double zer = EPS; // -EPS to include collinears -----
- for (int i = 0; i < n; h[k++] = p[i++]) -----
-- while (k >= 2 && cross(h[k-2],h[k-1],p[i]) < zer) -----
-- k; -----
- for(int i = n-2, t = k; i >= 0; h[k++] = p[i--]) -----
-- while (k > t && cross(h[k-2],h[k-1],p[i]) < zer) -----
-- k; -----
- k -= 1 + (h[0].x==h[1].x&&h[0].y==h[1].y ? 1 : 0);
- copy(h, h + k, p); delete[] h; return k; } -----
```

3.10.2. *3D Convex Hull*. Currently $O(N^2)$, but can be optimized to a randomized $O(N \log N)$ using the Clarkson-Shor algorithm. Sauce: Efficient 3D Convex Hull Tutorial on CF.

```
typedef std::vector<bool> vb; -----
struct point3D { -----
- ll x, y, z;
- point3D(ll x = 0, ll y = 0, ll z = 0) : x(x), y(y), z(z) {}
- point3D operator-(const point3D &o) const { -----
-- return point3D(x - o.x, y - o.y, z - o.z); } -----
- point3D cross(const point3D &o) const { -----
-- return point3D(y*o.z-z*o.y, z*o.x-x*o.z, x*o.y-y*o.x); }
- ll dot(const point3D &o) const { -----
-- return x*o.x + y*o.y + z*o.z; }
- bool operator==(const point3D &o) const { -----
-- return std::tie(x, y, z) == std::tie(o.x, o.y, o.z); }
- bool operator<(const point3D &o) const { -----
-- return std::tie(x, y, z) < std::tie(o.x, o.y, o.z); } };
struct face { -----
- std::vector<int> p_idx;
- point3D q; };
std::vector<face> convex_hull_3D(std::vector<point3D> &points) {
- int n = points.size();
- std::vector<face> faces;
- std::vector<vb> dead(points.size(), vb(points.size(), true));
- auto add_face = [&](int a, int b, int c) { -----
-- faces.push_back({a, b, c}, -----
-- (points[b] - points[a]).cross(points[c] - points[a]));
-- dead[a][b] = dead[b][c] = dead[c][a] = false; };
- add_face(0, 1, 2);
- add_face(0, 2, 1);
- for (int i = 3; i < n; ++i) { -----
-- std::vector<face> faces_inv;
-- for(face &f : faces) { -----
---- if ((points[i] - points[f.p_idx[0]]).dot(f.q) > 0) -----
---- for (int j = 0; j < 3; ++j) -----
----- dead[f.p_idx[j]][f.p_idx[(j+1)%3]] = true; -----
----- else -----
```

```
----- faces_inv.push_back(f); } -----
- faces.clear();
- for(face &f : faces_inv) { -----
-- for (int j = 0; j < 3; ++j) { -----
-- int a = f.p_idx[j], b = f.p_idx[(j + 1) % 3];
-- if(dead[b][a]) -----
-- add_face(b, a, i); } }
- faces.insert( -----
- faces.end(), faces_inv.begin(), faces_inv.end()); }
- return faces; } -----
```

3.11. *Delaunay Triangulation*. Simply map each point (x,y) to (x,y,x^2+y^2) , find the 3d convex hull, and drop the 3rd dimension.

3.12. *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) { -----
- bool in = false;
- for (int i = 0, j = n - 1; i < n; j = i++) -----
-- in ^= (((p[i].y > q.y) != (p[j].y > q.y)) && -----
-- q.x < (p[j].x - p[i].x) * (q.y - p[i].y) / -----
-- (p[j].y - p[i].y) + p[i].x);
- return in; }
bool onPolygon(point q, point p[], int n) { -----
- for (int i = 0, j = n - 1; i < n; j = i++) -----
- if (abs(dist(p[i], q) + dist(p[j], q) -----
----- dist(p[i], p[j])) < EPS) -----
-- return true;
- return false; } -----
```

3.13. *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) { -----
- vector<point> poly;
- for (int i = 0, j = n - 1; i < n; j = i++) { -----
-- double c1 = cross(a, b, p[j]);
-- double c2 = cross(a, b, p[i]);
-- if (c1 > -EPS) poly.push_back(p[j]);
-- if (c1 * c2 < -EPS)
-- poly.push_back(line_inter(p[j], p[i], a, b));
- } return poly; } -----
```

3.14. *Triangle Centers*.

```
point bary(point A, point B, point C, -----
-- double a, double b, double c) {
- return (A*a + B*b + C*c) / (a + b + c); }
point trilinear(point A, point B, point C, -----
-- double a, double b, double c) {
- return bary(A,B,C,abs(B-C)*a, -----
-- abs(C-A)*b,abs(A-B)*c); }
point centroid(point A, point B, point C) { -----
- return bary(A, B, C, 1, 1, 1); }
point circumcenter(point A, point B, point C) { -----
- double a=norm(B-C), b=norm(C-A), c=norm(A-B);
- return bary(A,B,C,a*(b+c-a),b*(c+a-b),c*(a+b-c)); }
point orthocenter(point A, point B, point C) { -----
- return bary(A,B,C, tan(angle(B,A,C)), -----
```

```
----- tan(angle(A,B,C)), tan(angle(A,C,B))); } -----
point incenter(point A, point B, point C) { -----
- return bary(A,B,C,abs(B-C),abs(A-C),abs(A-B)); } -----
// incircle radius given the side lengths a, b, c -----
double inradius(double a, double b, double c) { -----
- double s = (a + b + c) / 2;
- return sqrt(s * (s-a) * (s-b) * (s-c)) / s; }
point excenter(point A, point B, point C) { -----
- double a = abs(B-C), b = abs(C-A), c = abs(A-B);
- return bary(A, B, C, -a, b, c); } -----
// return bary(A, B, C, a, -b, c); -----
// return bary(A, B, C, a, b, -c); -----
point brocard(point A, point B, point C) { -----
- double a = abs(B-C), b = abs(C-A), c = abs(A-B);
- return bary(A,B,C,c/b*a,c/b*a,c/a*b,a/b*c); // CCW -----
// return bary(A,B,C,b/c*a,c/a*b,a/b*c); // CW } -----
point symmedian(point A, point B, point C) { -----
- return bary(A,B,C,norm(B-C),norm(C-A),norm(A-B)); } -----
```

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

```
std::vector<point> convex_polygon_inter( -----
-- point a[], int an, point b[], int bn) { -----
- point ans[an + bn + an*bn];
- int size = 0;
- for (int i = 0; i < an; ++i) -----
-- if (inPolygon(a[i],b,bn) || onPolygon(a[i],b,bn)) -----
-- ans[size++] = a[i];
- for (int i = 0; i < bn; ++i) -----
-- if (inPolygon(b[i],a,an) || onPolygon(b[i],a,an)) -----
-- ans[size++] = b[i];
- for (int i = 0, I = an - 1; i < an; I = i++) -----
-- for (int j = 0, J = bn - 1; j < bn; J = j++) { -----
-- try { -----
-- point p=line_inter(a[i],a[I],b[j],b[J],true);
-- ans[size++] = p;
-- } catch (exception ex) {} }
- size = convex_hull(ans, size);
- return vector<point>(ans, ans + size); } -----
```

3.16. *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: $\text{Area} = I + B/2 - 1$.

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

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

```
std::pair<point, double> bounding_ball(point p[], int n){ -----
- std::random_shuffle(p, p + n);
- point center(0, 0); double radius = 0;
- for (int i = 0; i < n; ++i) { -----
```

```
--- if (dist(center, p[i]) > radius + EPS) { -----
----- center = p[i]; radius = 0; -----
    for (int j = 0; j < i; ++j) -----
----- if (dist(center, p[j]) > radius + EPS) { -----
----- center.x = (p[i].x + p[j].x) / 2; -----
----- center.y = (p[i].y + p[j].y) / 2; -----
----- // center.z = (p[i].z + p[j].z) / 2; -----
----- radius = dist(center, p[i]); // midpoint -----
    for (int k = 0; k < j; ++k) -----
----- if (dist(center, p[k]) > radius + EPS) { -----
----- center = circumcenter(p[i], p[j], p[k]); -----
----- radius = dist(center, p[i]); } } } -----
- return {center, radius}; } -----
```

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

```
double shamos(point p[], int n) {
- point *h = new point[n+1]; copy(p, p + n, h);
- int k = convex_hull(h, n); if (k <= 2) return 0;
- h[k] = h[0]; double d = HUGE_VAL;
- for (int i = 0, j = 1; i < k; ++i) {
- while (distPtLine(h[j+1], h[i], h[i+1]) >=
- distPtLine(h[j], h[i], h[i+1])) {
- j = (j + 1) % k; }
- d = min(d, distPtLine(h[j], h[i], h[i+1]));
- } return d; }
```

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

```
#define cpoint const point& -----
bool cmpx(cpoint a, cpoint b) {return a.x < b.x;} -----
bool cmpy(cpoint a, cpoint b) {return a.y < b.y;} -----
struct KDTree { -----
- KDTree(point p[],int n): p(p), n(n) {build(0,n);} -----
- priority_queue< pair<double, point*> > pq; -----
- point *p; int n, k; double qx, qy, prune; -----
- void build(int L, int R, bool dvx=false) { -----
- if (L >= R) return; -----
- int M = (L + R) / 2; -----
- nth_element(p + L, p + M, p + R, dvx?cmpx:cmpy); -----
- build(L, M, !dvx); build(M + 1, R, !dvx); } -----
- void dfs(int L, int R, bool dvx) { -----
- if (L >= R) return; -----
- int M = (L + R) / 2; -----
- double dx = qx - p[M].x, dy = qy - p[M].y; -----
- double delta = dvx ? dx : dy; -----
- double D = dx * dx + dy * dy; -----
- if (D<=prune && (pq.size()<k||D<pq.top().first)) { -----
- pq.push(make_pair(D, &p[M])); -----
- if (pq.size() > k) pq.pop(); } -----
- int nL = L, nR = M, fL = M + 1, fR = R; -----
- if (delta > 0) {swap(nL, fL); swap(nR, fR);} -----
- dfs(nL, nR, !dvx); -----
- D = delta * delta; -----
- if (D<=prune && (pq.size()<k||D<pq.top().first)) -----
- dfs(fL, fR, !dvx); } -----
- // returns k nearest neighbors of (x, y) in tree -----
- // usage: vector<point> ans = tree.knn(x, y, 2); -----
```

```
- vector<point> knn(double x, double y, -----
----- int k=1, double r=-1) { -----
- qx=x; qy=y; this->k=k; prune=r<0?HUGE_VAL:r*r; -----
- dfs(0, n, false); vector<point> v; -----
- while (!pq.empty()) { -----
- v.push_back(*pq.top().second); -----
- pq.pop(); -----
- } reverse(v.begin(), v.end()); -----
- return v; } };
```

3.20. **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 k D Tree.

```
bool cmpy(const point& a, const point& b) { return a.y < b.y; }
double closest_pair_sweep(point p[], int n) {
- if (n <= 1) return HUGE_VAL;
- std::sort(p, p + n, cmpy);
- std::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);
- std::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 = std::min(best, std::sqrt(dx*dx + dy*dy));
- ++it; }
- box.insert(p[i]);
- } return best; }
```

3.21. **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.

3.22. **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 $(c_1r_2 - c_2r_1)/(r_1 + r_2)$.

4. GRAPHS

4.1. Single-Source Shortest Paths.

4.1.1. *Dijkstra.*

```
#include "graph_template_adjlist.cpp" -----
// insert inside graph; needs n, dist[], and adj[] -----
void dijkstra(int s) { -----
- for (int u = 0; u < n; ++u) -----
- dist[u] = INF; -----
- dist[s] = 0; -----
- std::priority_queue<ii, vii, std::greater<ii> > pq; -----
- pq.push({0, s}); -----
- while (!pq.empty()) { -----
- int u = pq.top().second; -----
- int d = pq.top().first; -----
- pq.pop(); -----
- if (dist[u] < d) -----
- continue; -----
- dist[u] = d; -----
- for (auto &e : adj[u]) { -----
- int v = e.first; -----
- int w = e.second; -----
- if (dist[v] > dist[u] + w) { -----
- dist[v] = dist[u] + w; -----
- pq.push({dist[v], v}); } } } -----
```

4.1.2. *Bellman-Ford.*

```
#include "graph_template_adjlist.cpp" -----
// insert inside graph; needs n, dist[], and adj[] -----
void bellman_ford(int s) { -----
- for (int u = 0; u < n; ++u) -----
- dist[u] = INF; -----
- dist[s] = 0; -----
- for (int i = 0; i < n-1; ++i) -----
- for (int u = 0; u < n; ++u) -----
- for (auto &e : adj[u]) -----
- if (dist[u] + e.second < dist[e.first]) -----
- dist[e.first] = dist[u] + e.second; } -----
// you can call this after running bellman_ford() -----
bool has_neg_cycle() { -----
- for (int u = 0; u < n; ++u) -----
- for (auto &e : adj[u]) -----
- if (dist[e.first] > dist[u] + e.second) -----
- return true; -----
- return false; }
```

4.1.3. *Shortest Path Faster Algorithm.*

```
#include "graph_template_adjlist.cpp" -----
// insert inside graph; -----
// needs n, dist[], in_queue[], num_vis[], and adj[] -----
bool spfa(int s) { -----
- for (int u = 0; u < n; ++u) { -----
- dist[u] = INF; -----
- in_queue[u] = 0; -----
```

```
-- num_vis[u] = 0; }
-- dist[s] = 0;
-- in_queue[s] = 1;
-- bool has_negative_cycle = false;
-- std::queue<int> q; q.push(s);
-- while (not q.empty()) {
--   int u = q.front(); q.pop(); in_queue[u] = 0;
--   if (++num_vis[u] >= n)
--     dist[u] = -INF, has_negative_cycle = true;
--   for (auto &[v, c] : adj[u])
--     if (dist[v] > dist[u] + c) {
--       dist[v] = dist[u] + c;
--       if (!in_queue[v]) {
--         q.push(v);
--         in_queue[v] = 1; } } }
-- return has_negative_cycle; }
```

4.2. All-Pairs Shortest Paths.

4.2.1. Floyd-Washall.

```
#include "graph_template_adjmat.cpp"
// insert inside graph; needs n and mat[][]
void floyd_warshall() {
  for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        if (mat[i][k] + mat[k][j] < mat[i][j])
          mat[i][j] = mat[i][k] + mat[k][j]; }
```

4.3. Strongly Connected Components.

4.3.1. Kosaraju.

```
struct kosaraju_graph {
  int n, *vis;
  vi **adj;
  std::vector<vi> sccs;
  kosaraju_graph(int n) {
    this->n = n;
    vis = new int[n];
    adj = new vi*[2];
    for (int dir = 0; dir < 2; ++dir)
      adj[dir] = new vi[n];
    void add_edge(int u, int v) {
      adj[0][u].push_back(v);
      adj[1][v].push_back(u);
    }
    void dfs(int u, int p, int dir, vi &topo) {
      vis[u] = 1;
      for (int v : adj[dir][u])
        if (!vis[v] && v != p) dfs(v, u, dir, topo);
      topo.push_back(u);
    }
    void kosaraju() {
      vi topo;
      for (int u = 0; u < n; ++u) vis[u] = 0;
      for (int u = 0; u < n; ++u) if(!vis[u]) dfs(u, -1, 0, topo);
      for (int u = 0; u < n; ++u) vis[u] = 0;
      for (int i = n-1; i >= 0; --i) {
        if (!vis[topo[i]]) {
```

```
        sccs.push_back({});
        dfs(topo[i], -1, 1, sccs.back()); } } };
```

4.3.2. Tarjan's Offline Algorithm.

```
int n, id[N], low[N], st[N], in[N], TOP, ID;
int scc[N], SCC_SIZE; // 0 <= scc[u] < SCC_SIZE
vector<int> adj[N]; // 0-based adjlist
void dfs(int u) {
  id[u] = low[u] = ID++;
  st[TOP++] = u; in[u] = 1;
  for (int v : adj[u]) {
    if (id[v] == -1) {
      dfs(v);
      low[u] = min(low[u], low[v]);
    } else if (in[v] == 1)
      low[u] = min(low[u], id[v]);
    if (id[u] == low[u]) {
      int sid = SCC_SIZE++;
      do {
        int v = st[--TOP];
        in[v] = 0; scc[v] = sid;
      } while (st[TOP] != u); }
  }
}
void tarjan() { // call tarjan() to load SCC
  memset(id, -1, sizeof(int) * n);
  SCC_SIZE = ID = TOP = 0;
  for (int i = 0; i < n; ++i)
    if (id[i] == -1) dfs(i); }
```

4.4. 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;
  vector<vector<double>> arr(n+1, vector<double>(n, mn));
  arr[0][0] = 0;
  rep(k,1,n+1) rep(j,0,n) iter(it,adj[j])
    arr[k][it->first] = min(arr[k][it->first],
                           it->second + arr[k-1][j]);
  rep(k,0,n) {
    double mx = -INFINITY;
    rep(i,0,n) mx = max(mx, (arr[n][i]-arr[k][i])/(n-k));
    mn = min(mn, mx);
  }
  return mn; }
```

4.5. Biconnected Components.

4.5.1. Bridges and Articulation Points.

```
struct graph {
  int n, *disc, *low, TIME;
  vi *adj, stk, articulation_points;
  std::set<ii> bridges;
  vvi comps;
  graph(int n) : n(n) {
    adj = new vi[n];
    disc = new int[n];
    low = new int[n];
    void add_edge(int u, int v) {
      adj[u].push_back(v);
```

```
      adj[v].push_back(u); }
  void bridges_artics(int u, int p) {
    disc[u] = low[u] = TIME++;
    stk.push_back(u);
    int children = 0;
    bool has_low_child = false;
    for (int v : adj[u]) {
      if (disc[v] == -1) {
        _bridges_artics(v, u);
        children++;
        if (disc[u] < low[v])
          bridges.insert({std::min(u, v), std::max(u, v)});
        if (disc[u] <= low[v]) {
          has_low_child = true;
          comps.push_back({u});
          while (comps.back().back() != v and !stk.empty()) {
            comps.back().push_back(stk.back());
            stk.pop_back(); }
          low[u] = std::min(low[u], low[v]);
        } else if (v != p)
          low[u] = std::min(low[u], disc[v]);
      }
      if ((p == -1 && children >= 2) ||
          (p != -1 && has_low_child))
        articulation_points.push_back(u);
    }
    void bridges_artics() {
      for (int u = 0; u < n; ++u) disc[u] = -1;
      stk.clear();
      articulation_points.clear();
      bridges.clear();
      comps.clear();
      TIME = 0;
      for (int u = 0; u < n; ++u) if (disc[u] == -1)
        _bridges_artics(u, -1); } }
```

4.5.2. Block Cut Tree.

```
// insert inside code for finding articulation points
graph build_block_cut_tree() {
  int bct_n = articulation_points.size() + comps.size();
  vi block_id(n), is_art(n, 0);
  graph tree(bct_n);
  for (int i = 0; i < articulation_points.size(); ++i) {
    block_id[articulation_points[i]] = i;
    is_art[articulation_points[i]] = 1;
  }
  for (int i = 0; i < comps.size(); ++i) {
    int id = i + articulation_points.size();
    for (int u : comps[i])
      if (is_art[u]) tree.add_edge(block_id[u], id);
    else
      block_id[u] = id;
  }
  return tree; }
```

4.5.3. Bridge Tree.

```
// insert inside code for finding bridges
// requires union_find and hasher
graph build_bridge_tree() {
  union_find uf(n);
  for (int u = 0; u < n; ++u) {
    for (int v : adj[u]) {
```



```
----- ii uv = { std::min(u, v), std::max(u, v) }; -----
----- if (bridges.find(uv) == bridges.end()) -----
----- uf.unite(u, v); } } -----
- hasher h; -----
- for (int u = 0; u < n; ++u)
-- if (u == uf.find(u)) h.get_hash(u);
- int tn = h.h.size(); -----
- graph tree(tn);
- for (int i = 0; i < M; ++i) { -----
-- int ui = h.get_hash(uf.find(u));
-- int vi = h.get_hash(uf.find(v));
-- if (ui != vi) tree.add_edge(ui, vi); }
- return tree; }
```

4.6. Minimum Spanning Tree.

4.6.1. Kruskal.

```
#include "graph_template_edgelist.cpp" -----
#include "union_find.cpp" -----
// insert inside graph; needs n, and edges
void kruskal(viii &res) { -----
- viii().swap(res); // or use res.clear(); -----
- std::priority_queue<iii, viii, std::greater<iii> > pq; -----
- for (auto &edge : edges) -----
-- pq.push(edge);
- union_find uf(n); -----
- while (!pq.empty()) { -----
-- auto node = pq.top(); pq.pop();
-- int u = node.second.first;
-- int v = node.second.second;
-- if (uf.unite(u, v))
-- res.push_back(node); } } -----
```

4.6.2. Prim.

```
#include "graph_template_adjlist.cpp" -----
// insert inside graph; needs n, vis[], and adj[]
void prim(viii &res, int s=0) { -----
- viii().swap(res); // or use res.clear(); -----
- std::priority_queue<ii, vii, std::greater<ii> > pq; -----
- pq.push({0, s});
- vis[s] = true;
- while (!pq.empty()) { -----
-- int u = pq.top().second; pq.pop();
-- vis[u] = true;
-- for (auto &[v, w] : adj[u]) { -----
--- if (v == u) continue;
--- if (vis[v]) continue;
--- res.push_back({w, {u, v}});
--- pq.push({w, v}); } } } -----
```

4.7. Euler Path/Cycle .

4.7.1. Euler Path/Cycle in a Directed Graph .

```
#define MAXV 1000 -----
#define MAXE 5000 -----
vi adj[MAXV];
int n, m, indeg[MAXV], outdeg[MAXV], res[MAXE + 1];
```

```
ii start_end() { -----
- int start = -1, end = -1, any = 0, c = 0;
- rep(i,0,n) { -----
-- if (outdeg[i] > 0) any = i;
-- if (indeg[i] + 1 == outdeg[i]) start = i, c++;
-- else if (indeg[i] == outdeg[i] + 1) end = i, c++;
-- else if (indeg[i] != outdeg[i]) return ii(-1,-1); }
- if ((start == -1) != (end == -1) || (c != 2 && c != 0)) -----
-- return ii(-1,-1);
- if (start == -1) start = end = any;
- return ii(start, end); }
bool euler_path() { -----
- ii se = start_end();
- int cur = se.first, at = m + 1;
- if (cur == -1) return false;
- stack<int> s;
- while (true) { -----
-- if (outdeg[cur] == 0) { -----
--- res[--at] = cur;
--- if (s.empty()) break;
--- cur = s.top(); s.pop();
--- } else s.push(cur), cur = adj[cur][--outdeg[cur]]; }
- return at == 0; }
```

4.7.2. Euler Path/Cycle in an Undirected Graph .

```
multiset<int> adj[1010];
list<int> L;
list<int>::iterator euler(int at, int to,
list<int>::iterator it) { -----
- if (at == to) return it;
- L.insert(it, at), --it;
- while (!adj[at].empty()) { -----
-- int nxt = *adj[at].begin();
-- adj[at].erase(adj[at].find(nxt));
-- adj[nxt].erase(adj[nxt].find(at));
-- if (to == -1) { -----
--- it = euler(nxt, at, it);
--- L.insert(it, at);
--- --it;
-- } else { -----
--- it = euler(nxt, to, it);
--- to = -1; } }
- return it; }
// euler(0,-1,L.begin()) -----
```

4.8. Bipartite Matching .

4.8.1. Alternating Paths Algorithm .

```
vi* adj;
bool* done;
int* owner;
int alternating_path(int left) { -----
- if (done[left]) return 0;
- done[left] = true;
- rep(i,0,size(adj[left])) { -----
-- int right = adj[left][i];
```

```
--- if (owner[right] == -1 || alternating_path(owner[right])) {
--- owner[right] = left; return 1; } } -----
- return 0; } -----
```

4.8.2. Hopcroft-Karp Algorithm .

```
#define MAXN 5000 -----
int dist[MAXN+1], q[MAXN+1];
#define dist(v) dist[v == -1 ? MAXN : v] -----
struct bipartite_graph { -----
- int N, M, *L, *R; vi *adj; -----
- bipartite_graph(int _N, int _M) : N(_N), M(_M), -----
-- L(new int[N]), R(new int[M]), adj(new vi[N]) {}
- ~bipartite_graph() { delete[] adj; delete[] L; delete[] R; }
- bool bfs() { -----
-- int l = 0, r = 0; -----
-- rep(v,0,N) if(L[v] == -1) dist(v) = 0, q[r++] = v;
-- else dist(v) = INF;
-- dist(-1) = INF;
-- while(l < r) { -----
--- int v = q[l++];
--- if(dist(v) < dist(-1)) { -----
---- iter(u, adj[v]) if(dist(R[*u]) == INF)
---- dist(R[*u]) = dist(v) + 1, q[r++] = R[*u]; } }
-- return dist(-1) != INF; }
- bool dfs(int v) { -----
-- if(v != -1) { -----
--- iter(u, adj[v])
--- if(dist(R[*u]) == dist(v) + 1)
--- if(dfs(R[*u])) { -----
---- R[*u] = v, L[v] = *u;
--- return true; }
--- dist(v) = INF;
--- return false; }
-- return true; }
- void add_edge(int i, int j) { adj[i].push_back(j); }
- int maximum_matching() { -----
-- int matching = 0;
-- memset(L, -1, sizeof(int) * N);
-- memset(R, -1, sizeof(int) * M);
-- while(bfs()) rep(i,0,N)
-- matching += L[i] == -1 && dfs(i);
-- return matching; } };
```

4.8.3. Minimum Vertex Cover in Bipartite Graphs .

```
#include "hopcroft_karp.cpp" -----
vector<bool> alt;
void dfs(bipartite_graph &g, int at) { -----
- alt[at] = true;
- iter(it,g.adj[at]) { -----
-- alt[*it + g.N] = true;
-- if (g.R[*it] != -1 && !alt[g.R[*it]])
-- dfs(g, g.R[*it]); } }
vi mvc_bipartite(bipartite_graph &g) { -----
- vi res; g.maximum_matching();
- alt.assign(g.N + g.M,false);
- rep(i,0,g.N) if (g.L[i] == -1) dfs(g, i);
```

```
- rep(i,0,g.N) if (!alt[i]) res.push_back(i);
- rep(i,0,g.M) if (alt[g.N + i]) res.push_back(g.N + i);
- return res; }
```

4.9. Maximum Flow.

4.9.1. Edmonds-Karp. $O(VE^2)$

```
struct flow_network {
- int n, s, t, *par, **c, **f;
- vi adj;
- flow_network(int n, int s, int t) : n(n), s(s), t(t) {
- adj = new std::vector<int>[n];
- par = new int[n];
- c = new int*[n];
- f = new int*[n];
- for (int i = 0; i < n; ++i) {
- c[i] = new int[n];
- f[i] = new int[n];
- for (int j = 0; j < n; ++j)
- c[i][j] = f[i][j] = 0; } }
- void add_edge(int u, int v, int w) {
- adj[u].push_back(v);
- adj[v].push_back(u);
- c[u][v] += w; }
- int res(int i, int j) { return c[i][j] - f[i][j]; }
- bool bfs() {
- std::queue<int> q;
- q.push(this->s);
- while (!q.empty()) {
- int u = q.front(); q.pop();
- for (int v : adj[u]) {
- if (res(u, v) > 0 and par[v] == -1) {
- par[v] = u;
- if (v == this->t)
- return true;
- q.push(v); } } }
- return false; }
- bool aug_path() {
- for (int u = 0; u < n; ++u) par[u] = -1;
- par[s] = s;
- return bfs(); }
- int calc_max_flow() {
- int ans = 0;
- while (aug_path()) {
- int flow = INF;
- for (int u = t; u != s; u = par[u])
- flow = std::min(flow, res(par[u], u));
- for (int u = t; u != s; u = par[u])
- f[par[u]][u] += flow, f[u][par[u]] -= flow;
- ans += flow; }
- return ans; } }
```

4.9.2. Dinic. $O(V^2E)$

```
struct edge {
- int u, v;
- ll cap, flow;
- edge(int u, int v, ll cap) :
```

```
-- u(u), v(v), cap(cap), flow(0) {} };
struct flow_network {
- int n, s, t, *adj_ptr, *par;
- ll *dist;
- std::vector<edge> edges;
- std::vector<int> *adj;
- flow_network(int n, int s, int t) : n(n), s(s), t(t) {
- adj = new std::vector<int>[n];
- adj_ptr = new int[n];
- par = new int[n];
- dist = new ll[n]; }
- void add_edge(int u, int v, ll cap, bool bi=false) {
- adj[u].push_back(edges.size());
- edges.push_back(edge(u, v, cap));
- adj[v].push_back(edges.size());
- edges.push_back(edge(v, u, (bi ? cap : 0LL))); }
- ll res(edge &e) { return e.cap - e.flow; }
- bool make_level_graph() {
- for (int u = 0; u < n; ++u) dist[u] = -1;
- dist[s] = 0;
- std::queue<int> q; q.push(s);
- while (!q.empty()) {
- int u = q.front(); q.pop();
- for (int i : adj[u]) {
- edge &e = edges[i];
- if (dist[e.v] < 0 and res(e)) {
- dist[e.v] = dist[u] + 1;
- q.push(e.v); } } }
- return dist[t] != -1; }
- bool is_next(int u, int v) {
- return dist[v] == dist[u] + 1; }
- bool dfs(int u) {
- if (u == t) return true;
- for (int &ii = adj_ptr[u]; ii < adj[u].size(); ++ii) {
- int i = adj[u][ii];
- edge &e = edges[i];
- if (is_next(u, e.v) and res(e) > 0 and dfs(e.v)) {
- par[e.v] = i;
- return true; } }
- return false; }
- bool aug_path() {
- for (int u = 0; u < n; ++u) par[u] = -1;
- return dfs(s); }
- ll calc_max_flow() {
- ll total_flow = 0;
- while (make_level_graph()) {
- for (int u = 0; u < n; ++u) adj_ptr[u] = 0;
- while (aug_path()) {
- ll flow = INF;
- for (int i = par[t]; i != -1; i = par[edges[i].u])
- flow = std::min(flow, res(edges[i]));
- for (int i = par[t]; i != -1; i = par[edges[i].u]) {
- edges[i].flow += flow;
- edges[i^1].flow -= flow; }
- total_flow += flow; } }
- return total_flow; } }
```

4.9.3. Push-relabel. $\omega(VE + V^2\sqrt{E})$, $O(V^3)$

```
int n;
std::vector<vi> capacity, flow;
vi height, excess;
void push(int u, int v) {
- int d = min(excess[u], capacity[u][v] - flow[u][v]);
- flow[u][v] += d; flow[v][u] -= d;
- excess[u] -= d; excess[v] += d; }
void relabel(int u) {
- int d = INF;
- for (int i = 0; i < n; i++)
- if (capacity[u][i] - flow[u][i] > 0)
- d = min(d, height[i]);
- if (d < INF) height[u] = d + 1; }
vi find_max_height_vertices(int s, int t) {
- vi max_height;
- for (int i = 0; i < n; i++) {
- if (i != s && i != t && excess[i] > 0) {
- if (!max_height.empty() && height[i] > height[max_height[0]])
- max_height.clear();
- if (max_height.empty() || height[i] == height[max_height[0]])
- max_height.push_back(i); } }
- return max_height; }
int max_flow(int s, int t) {
- flow.assign(n, vi(n, 0));
- height.assign(n, 0); height[s] = n;
- excess.assign(n, 0); excess[s] = INF;
- for (int i = 0; i < n; i++) if (i != s) push(s, i);
- vi current;
- while (!(current = find_max_height_vertices(s, t)).empty()) {
- for (int i : current) {
- bool pushed = false;
- for (int j = 0; j < n && excess[i]; j++) {
- if (capacity[i][j] - flow[i][j] > 0 &&
- height[i] == height[j] + 1) {
- push(i, j);
- pushed = true; } }
- if (!pushed) relabel(i), break; } }
- int max_flow = 0;
- for (int i = 0; i < n; i++) max_flow += flow[i][t];
- return max_flow; }
```

4.9.4. Gomory-Hu (All-pairs Maximum Flow).

```
#define MAXV 2000
int q[MAXV], d[MAXV];
struct flow_network {
- struct edge { int v, nxt, cap;
- edge(int _v, int _cap, int _nxt)
- : v(_v), nxt(_nxt), cap(_cap) {} };
- int n, *head, *curh; vector<edge> e, e_store;
- flow_network(int _n) : n(_n) {
- curh = new int[n];
- memset(head = new int[n], -1, n*sizeof(int)); }
- void reset() { e = e_store; }
- void add_edge(int u, int v, int uv, int vu=0) {
- 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) {
--- if (v == t) return f;
--- for (int &i = curh[v], ret; i != -1; i = e[i].nxt)
----- if (e[i].cap > 0 && d[e[i].v] + 1 == d[v])
----- 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; }
- int max_flow(int s, int t, bool res=true) {
--- e_store = e;
--- int l, r, f = 0, x;
--- while (true) {
----- memset(d, -1, n*sizeof(int));
----- l = r = 0, d[q[r++] = t] = 0;
----- while (l < r)
----- 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)
----- d[q[r++] = e[i].v] = d[v]+1;
----- if (d[s] == -1) break;
----- memcpy(curh, head, n * sizeof(int));
----- while ((x = augment(s, t, INF)) != 0) f += x; }
--- if (res) reset();
--- return f; } };
bool same[MAXV];
pair<vii, vvi> construct_gh_tree(flow_network &g) {
--- int n = g.n, v;
--- vii par(n, ii(0, 0)); vvi cap(n, vi(n, -1));
--- rep(s,1,n) {
--- int l = 0, r = 0;
--- par[s].second = g.max_flow(s, par[s].first, false);
--- memset(d, 0, n * sizeof(int));
--- memset(same, 0, n * sizeof(bool));
--- d[q[r++] = s] = 1;
--- while (l < r) {
----- same[v = q[l++]] = true;
----- for (int i = g.head[v]; i != -1; i = g.e[i].nxt)
----- if (g.e[i].cap > 0 && d[g.e[i].v] == 0)
----- d[q[r++] = g.e[i].v] = 1; }
--- rep(i,s+1,n)
----- if (par[i].first == par[s].first && same[i])
----- par[i].first = s;
--- g.reset(); }
--- rep(i,0,n) {
--- int mn = INF, cur = i;
--- while (true) {
----- cap[cur][i] = mn;
----- if (cur == 0) break;
----- mn = min(mn, par[cur].second), cur = par[cur].first; } }
--- return make_pair(par, cap); }
int compute_max_flow(int s, int t, const pair<vii, vvi> &gh) {
- int cur = INF, at = s;
- while (gh.second[at][t] == -1)
--- cur = min(cur, gh.first[at].second),
--- at = gh.first[at].first;
- return min(cur, gh.second[at][t]); }

```

4.10. Minimum Cost Maximum Flow.

```

struct edge {
- int u, v; ll cost, cap, flow;
- edge(int u, int v, ll cap, ll cost) :
--- u(u), v(v), cap(cap), cost(cost), flow(0) {} };
struct flow_network {
- int n, s, t, *par, *in_queue, *num_vis; ll *dist, *pot;
- std::vector<edge> edges;
- std::vector<int> adj;
- std::map<std::pair<int, int>, std::vector<int>> edge_idx;
- flow_network(int n, int s, int t) : n(n), s(s), t(t) {
--- adj = new std::vector<int>(n);
--- par = new int[n];
--- in_queue = new int[n];
--- num_vis = new int[n];
--- dist = new ll[n];
--- pot = new ll[n];
--- for (int u = 0; u < n; ++u) pot[u] = 0; }
- void add_edge(int u, int v, ll cap, ll cost) {
--- adj[u].push_back(edges.size());
--- edge_idx[{u, v}].push_back(edges.size());
--- edges.push_back(edge(u, v, cap, cost));
--- adj[v].push_back(edges.size());
--- edge_idx[{v, u}].push_back(edges.size());
--- edges.push_back(edge(v, u, 0LL, -cost)); }
- ll get_flow(int u, int v) {
--- ll f = 0;
--- for (int i : edge_idx[{u, v}]) f += edges[i].flow;
--- return f; }
- ll res(edge &e) { return e.cap - e.flow; }
- void bellman_ford() {
--- for (int u = 0; u < n; ++u) pot[u] = INF;
--- pot[s] = 0;
--- for (int it = 0; it < n-1; ++it)
--- for (auto e : edges)
----- if (res(e) > 0)
----- pot[e.v] = std::min(pot[e.v], pot[e.u] + e.cost); }
- bool spfa () {
--- std::queue<int> q; q.push(s);
--- while (not q.empty()) {
--- int u = q.front(); q.pop(); in_queue[u] = 0;
--- if (++num_vis[u] >= n) {
----- dist[u] = -INF;
----- return false; }
--- for (int i : adj[u]) {
--- edge e = edges[i];
--- if (res(e) <= 0) continue;
--- ll nd = dist[u] + e.cost + pot[u] - pot[e.v];
--- if (dist[e.v] > nd) {
----- dist[e.v] = nd;
----- par[e.v] = i;
----- if (not in_queue[e.v]) {
----- q.push(e.v);
----- in_queue[e.v] = 1; } } } }
--- return dist[t] != INF; }
- bool aug_path() {
--- for (int u = 0; u < n; ++u) {

```

```

--- par[u] = -1;
--- in_queue[u] = 0;
--- num_vis[u] = 0;
--- dist[u] = INF; }
--- dist[s] = 0;
--- in_queue[s] = 1;
--- return spfa();
- }
- pll calc_max_flow(bool do_bellman_ford=false) {
--- ll total_cost = 0, total_flow = 0;
--- if (do_bellman_ford)
--- bellman_ford();
--- while (aug_path()) {
--- ll f = INF;
--- for (int i = par[t]; i != -1; i = par[edges[i].u])
--- f = std::min(f, res(edges[i]));
--- for (int i = par[t]; i != -1; i = par[edges[i].u]) {
--- edges[i].flow += f;
--- edges[i^1].flow -= f; }
--- total_cost += f * (dist[t] + pot[t] - pot[s]);
--- total_flow += f;
--- for (int u = 0; u < n; ++u)
--- if (par[u] != -1) pot[u] += dist[u]; }
--- return {total_cost, total_flow}; } };

```

4.10.1. Hungarian Algorithm.

```

int n, m; // size of A, size of B
int cost[N+1][N+1]; // input cost matrix, 1-indexed
int way[N+1], p[N+1]; // p[i]=j: Ai is matched to Bj
int minv[N+1], A[N+1], B[N+1]; bool used[N+1];
int hungarian() {
- for (int i = 0; i <= N; ++i)
--- A[i] = B[i] = p[i] = way[i] = 0; // init
- for (int i = 1; i <= n; ++i) {
--- p[0] = i; int R = 0;
--- for (int j = 0; j <= m; ++j)
--- minv[j] = INF, used[j] = false;
--- do {
----- int L = p[R], dR = 0;
----- int delta = INF;
----- used[R] = true;
----- for (int j = 1; j <= m; ++j)
----- if (!used[j]) {
----- int c = cost[L][j] - A[L] - B[j];
----- if (c < minv[j]) minv[j] = c, way[j] = R;
----- if (minv[j] < delta) delta = minv[j], dR = j;
----- }
----- for (int j = 0; j <= m; ++j)
----- if (used[j]) A[p[j]] += delta, B[j] -= delta;
----- else minv[j] -= delta;
----- R = dR;
--- } while (p[R] != 0);
--- for (; R != 0; R = way[R])
--- p[R] = p[way[R]]; }
- return -B[0]; }

```

4.11. **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 i th element is the edge for the i th vertex. The answer for the root is undefined!

```
#include "../data-structures/union_find.cpp"
struct arborescence {
    int n; union_find uf;
    vector<vector<pair<ii,int>>> adj;
    arborescence(int _n) : n(_n), uf(n), adj(n) {}
    void add_edge(int a, int b, int c) {
        adj[b].push_back(make_pair(ii(a,b),c));
    }
    vii find_min(int r) {
        vi vis(n,-1), mn(n,INF); vii par(n);
        rep(i,0,n) {
            if (uf.find(i) != i) continue;
            int at = i;
            while (at != r && vis[at] == -1) {
                vis[at] = i;
                iter(it,adj[at]) if (it->second < mn[at] &&
                    uf.find(it->first.first) != at)
                    mn[at] = it->second, par[at] = it->first;
                if (par[at] == ii(0,0)) return vii();
                at = uf.find(par[at].first);
            }
            if (at == r || vis[at] != i) continue;
            union_find tmp = uf; vi seq;
            do { seq.push_back(at); at = uf.find(par[at].first);
            } while (at != seq.front());
            iter(it,seq) uf.unite(*it,seq[0]);
            int c = uf.find(seq[0]);
            vector<pair<ii,int>> nw;
            iter(it,seq) iter(jt,adj[*it])
                nw.push_back(make_pair(jt->first,
                    jt->second - mn[*it]));
            adj[c] = nw;
            vii rest = find_min(r);
            if (size(rest) == 0) return rest;
            ii use = rest[c];
            rest[at = tmp.find(use.second)] = use;
            iter(it,seq) if (*it != at)
                rest[*it] = par[*it];
            return rest;
        }
        return par;
    }
};
```

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

```
#define MAXV 300
bool marked[MAXV], emarked[MAXV][MAXV];
int S[MAXV];
vi find_augmenting_path(const vector<vi> &adj, const vi &m){
    int n = size(adj), s = 0;
    vi par(n,-1), height(n), root(n,-1), q, a, b;
    memset(marked,0,sizeof(marked));
    memset(emarked,0,sizeof(emarked));
    rep(i,0,n) if (m[i] >= 0) emarked[i][m[i]] = true;
    else root[i] = i, S[s++] = i;
```

```
    while (s) {
        int v = S[--s];
        iter(wt,adj[v]) {
            int w = *wt;
            if (emarked[v][w]) continue;
            if (root[w] == -1) {
                int x = S[s++] = m[w];
                par[w]=v, root[w]=root[v], height[w]=height[v]+1;
                par[x]=w, root[x]=root[w], height[x]=height[w]+1;
            } else if (height[w] % 2 == 0) {
                if (root[v] != root[w]) {
                    while (v != -1) q.push_back(v), v = par[v];
                    reverse(q.begin(), q.end());
                    while (w != -1) q.push_back(w), w = par[w];
                    return q;
                } else {
                    int c = v;
                    while (c != -1) a.push_back(c), c = par[c];
                    c = w;
                    while (c != -1) b.push_back(c), c = par[c];
                    while (!a.empty() && !b.empty() && a.back() == b.back())
                        c = a.back(), a.pop_back(), b.pop_back();
                    memset(marked,0,sizeof(marked));
                    fill(par.begin(), par.end(), 0);
                    iter(it,a) par[*it] = 1; iter(it,b) par[*it] = 1;
                    par[c] = s = 1;
                    rep(i,0,n) root[par[i] = par[i] ? 0 : s++] = i;
                    vector<vi> adj2(s);
                    rep(i,0,n) iter(it,adj[i]) {
                        if (par[*it] == 0) continue;
                        if (par[i] == 0) {
                            if (!marked[par[*it]]) {
                                adj2[par[i]].push_back(par[*it]);
                                adj2[par[*it]].push_back(par[i]);
                                marked[par[*it]] = true;
                            }
                            else adj2[par[i]].push_back(par[*it]);
                        }
                    }
                    vi m2(s, -1);
                    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)
                        m2[par[i]] = par[m[i]];
                    vi p = find_augmenting_path(adj2, m2);
                    int t = 0;
                    while (t < size(p) && p[t]) t++;
                    if (t == size(p)) {
                        rep(i,0,size(p)) p[i] = root[p[i]];
                        return p;
                    }
                    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]]);
                    iter(it,adj[root[p[t-1]]) {
                        if (par[*it] != (s = 0)) continue;
                        a.push_back(c), reverse(a.begin(), a.end());
                        iter(jt,b) a.push_back(*jt);
                        while (a[s] != *it) s++;
                        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);
                    rep(i,t+1,size(p)) q.push_back(root[p[i]]);
                    return q;
                }
            }
            emarked[v][w] = emarked[w][v] = true;
            marked[v] = true;
        }
        return q;
    }
}
vii max_matching(const vector<vi> &adj) {
    vi m(size(adj), -1), ap; vii res, es;
    rep(i,0,size(adj)) iter(it,adj[i]) es.emplace_back(i,*it);
    random_shuffle(es.begin(), es.end());
    iter(it,es) if (m[it->first] == -1 && m[it->second] == -1)
        m[it->first] = it->second, m[it->second] = it->first;
    do { ap = find_augmenting_path(adj, m);
        rep(i,0,size(ap)) m[m[ap[i^1]] = ap[i]] = ap[i^1];
    } while (!ap.empty());
    rep(i,0,size(m)) if (i < m[i]) res.emplace_back(i, m[i]);
    return res;
}
```

4.13. **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).

4.14. **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.

4.15. **Maximum Weighted Ind. 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.

4.16. **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.

4.17. **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.

4.18. **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.

4.19. Heavy Light Decomposition.

```
#include "segment_tree.cpp"
struct heavy_light_tree {
- int n, *par, *heavy, *dep, *path_root, *pos;
- std::vector<int> *adj;
- segtree *segment_tree;
- heavy_light_tree(int n) : n(n) {
-- this->adj = new std::vector<int>(n);
-- segment_tree = new segtree(0, n-1);
-- par = new int[n];
-- heavy = new int[n];
-- dep = new int[n];
-- path_root = new int[n];
-- pos = new int[n]; }
- void add_edge(int u, int v) {
-- adj[u].push_back(v);
-- adj[v].push_back(u); }
- void build(int root) {
-- for (int u = 0; u < n; ++u)
---- heavy[u] = -1;
-- par[root] = root;
-- dep[root] = 0;
-- dfs(root);
-- for (int u = 0, p = 0; u < n; ++u) {
---- if (par[u] == -1 or heavy[par[u]] != u) {
----- for (int v = u; v != -1; v = heavy[v]) {
----- -- path_root[v] = u;
----- -- pos[v] = p++; } } }
- int dfs(int u) {
-- int sz = 1;
-- int max_subtree_sz = 0;
-- for (int v : adj[u]) {
---- if (v != par[u]) {
----- par[v] = u;
----- dep[v] = dep[u] + 1;
----- int subtree_sz = dfs(v);
----- if (max_subtree_sz < subtree_sz) {
----- -- max_subtree_sz = subtree_sz;
----- -- heavy[u] = v; }
----- sz += subtree_sz; } }
-- return sz; }
- int query(int u, int v) {
-- int res = 0;
-- while (path_root[u] != path_root[v]) {
---- if (dep[path_root[u]] > dep[path_root[v]])
----- std::swap(u, v);
---- res += segment_tree->sum(pos[path_root[v]], pos[v]);
---- v = par[path_root[v]]; }
-- res += segment_tree->sum(pos[u], pos[v]);
-- return res; }
- void update(int u, int v, int c) {
-- for (; path_root[u] != path_root[v]; v = par[path_root[v]]){
---- if (dep[path_root[u]] > dep[path_root[v]])
```

```
----- std::swap(u, v);
---- segment_tree->increase(pos[path_root[v]], pos[v], c); }
-- segment_tree->increase(pos[u], pos[v], c); } };
```

4.20. Centroid Decomposition.

```
#define MAXV 100100
#define LGMAXV 20
int jmp[MAXV][LGMAXV],
- path[MAXV][LGMAXV],
- sz[MAXV], seph[MAXV],
- shortest[MAXV];
struct centroid_decomposition {
- int n; vvi adj;
- centroid_decomposition(int _n) : n(_n), adj(n) { }
- void add_edge(int a, int b) {
-- adj[a].push_back(b); adj[b].push_back(a); }
- int dfs(int u, int p) {
-- sz[u] = 1;
-- rep(i,0,size(adj[u]))
---- if (adj[u][i] != p) sz[u] += dfs(adj[u][i], u);
-- return sz[u]; }
- void makepaths(int sep, int u, int p, int len) {
-- jmp[u][seph[sep]] = sep, path[u][seph[sep]] = len;
-- int bad = -1;
-- rep(i,0,size(adj[u])) {
---- if (adj[u][i] == p) bad = i;
---- else makepaths(sep, adj[u][i], u, len + 1); }
-- if (p == sep)
---- swap(adj[u][bad], adj[u].back()), adj[u].pop_back(); }
- void separate(int h=0, int u=0) {
-- dfs(u,-1); int sep = u;
-- down: iter(nxt,adj[sep])
---- if (sz[*nxt] < sz[sep] && sz[*nxt] > sz[u]/2) {
-- sep = *nxt; goto down; }
-- seph[sep] = h, makepaths(sep, sep, -1, 0);
-- rep(i,0,size(adj[sep])) separate(h+1, adj[sep][i]); }
- void paint(int u) {
-- rep(h,0,seph[u]+1)
---- shortest[jmp[u][h]] = min(shortest[jmp[u][h]],
----- path[u][h]); }
- int closest(int u) {
-- int mn = INF/2;
-- rep(h,0,seph[u]+1)
---- mn = min(mn, path[u][h] + shortest[jmp[u][h]]);
-- return mn; } };
```

4.21. Least Common Ancestor.

4.21.1. Binary Lifting.

```
struct graph {
- int n, logn, *dep, **par;
- std::vector<int> *adj;
- graph(int n, int logn=20) : n(n), logn(logn) {
-- adj = new std::vector<int>(n);
-- dep = new int[n];
-- par = new int*[n];
-- for (int i = 0; i < n; ++i) par[i] = new int[logn]; }
```

```
- void dfs(int u, int p, int d) {
-- dep[u] = d;
-- par[u][0] = p;
-- for (int v : adj[u])
---- if (v != p) dfs(v, u, d+1); }
- int ascend(int u, int k) {
-- for (int i = 0; i < logn; ++i)
---- if (k & (1 << i)) u = par[u][i];
-- return u; }
- int lca(int u, int v) {
-- if (dep[u] > dep[v]) u = ascend(u, dep[u] - dep[v]);
-- if (dep[v] > dep[u]) v = ascend(v, dep[v] - dep[u]);
-- if (u == v) return u;
-- for (int k = logn-1; k >= 0; --k) {
---- if (par[u][k] != par[v][k]) {
----- u = par[u][k]; v = par[v][k]; } }
-- return par[u][0]; }
- bool is_anc(int u, int v) {
-- if (dep[u] < dep[v]) std::swap(u, v);
-- return ascend(u, dep[u] - dep[v]) == v; }
- void prep_lca(int root=0) {
-- dfs(root, root, 0);
-- for (int k = 1; k < logn; ++k)
---- for (int u = 0; u < n; ++u)
----- par[u][k] = par[par[u][k-1]][k-1]; }
```

4.21.2. Euler Tour Sparse Table.

```
struct graph {
- int n, logn, *par, *dep, *first, *lg, **spt;
- vi *adj, euler; // spt size should be ~ 2n
- graph(int n, int logn=20) : n(n), logn(logn) {
-- adj = new vi[n];
-- par = new int[n];
-- dep = new int[n];
-- first = new int[n]; }
- void add_edge(int u, int v) {
-- adj[u].push_back(v); adj[v].push_back(u); }
- void dfs(int u, int p, int d) {
-- dep[u] = d; par[u] = p;
-- first[u] = euler.size();
-- euler.push_back(u);
-- for (int v : adj[u])
---- if (v != p) {
----- dfs(v, u, d+1);
----- euler.push_back(u); } }
- void prep_lca(int root=0) {
-- dfs(root, root, 0);
-- int en = euler.size();
-- lg = new int[en+1];
-- lg[0] = lg[1] = 0;
-- for (int i = 2; i <= en; ++i)
---- lg[i] = lg[i >> 1] + 1;
-- spt = new int*[en];
-- for (int i = 0; i < en; ++i) {
---- spt[i] = new int[lg[en]];
---- spt[i][0] = euler[i]; }
```

```
--- for (int k = 0; (2 << k) <= en; ++k) -----
----- for (int i = 0; i + (2 << k) <= en; ++i) -----
----- if (dep[spt[i][k]] < dep[spt[i+(1<<k)][k]]) -----
----- spt[i][k+1] = spt[i][k]; -----
----- else -----
----- spt[i][k+1] = spt[i+(1<<k)][k]; }
- int lca(int u, int v) {
--- int a = first[u], b = first[v];
--- if (a > b) std::swap(a, b);
--- int k = lg[b-a+1], ba = b - (1 << k) + 1;
--- if (dep[spt[a][k]] < dep[spt[ba][k]]) return spt[a][k];
--- return spt[ba][k]; } };
```

4.21.3. Tarjan Off-line LCA.

```
#include "../data-structures/union-find.cpp"
struct tarjan_lca {
- int *ancestor;
- vi *adj, answers;
- vii *queries;
- bool *colored;
- union_find uf;
- tarjan_lca(int n, vi *_adj) : adj(_adj), uf(n) {
--- colored = new bool[n];
--- ancestor = new int[n];
--- queries = new vii[n];
--- memset(colored, 0, n); }
- void query(int x, int y) {
--- queries[x].push_back(ii(y, size(answers)));
--- queries[y].push_back(ii(x, size(answers)));
--- answers.push_back(-1); }
- void process(int u) {
--- ancestor[u] = u;
--- rep(i,0,size(adj[u])) {
--- int v = adj[u][i];
--- process(v);
--- uf.unite(u,v);
--- ancestor[uf.find(u)] = u; }
--- colored[u] = true;
--- rep(i,0,size(queries[u])) {
--- int v = queries[u][i].first;
--- if (colored[v])
--- answers[queries[u][i].second] = ancestor[uf.find(v)];
} } };
```

4.22. 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 = $|\text{cofactor}(D - A)|$

4.23. Erdős-Gallai Theorem. A sequence of non-negative integers $d_1 \geq \dots \geq d_n$ can be represented as the degree sequence of finite simple

graph on n vertices if and only if $d_1 + \dots + d_n$ is even and the following holds for $1 \leq k \leq n$:

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

4.24. Tree Isomorphism.

```
// REQUIREMENT: list of primes pr[], see prime sieve
typedef long long LL;
int pre[N], q[N], path[N]; bool vis[N];
// perform BFS and return the last node visited
int bfs(int u, vector<int> adj[]) {
--- memset(vis, 0, sizeof(vis));
--- int head = 0, tail = 0;
--- q[tail++] = u; vis[u] = true; pre[u] = -1;
--- while (head != tail) {
--- u = q[head]; if (++head == N) head = 0;
--- for (int i = 0; i < adj[u].size(); ++i) {
--- int v = adj[u][i];
--- if (!vis[v]) {
--- vis[v] = true; pre[v] = u;
--- q[tail++] = v; if (tail == N) tail = 0;
--- } } }
--- return u;
} // returns the list of tree centers
vector<int> tree_centers(int r, vector<int> adj[]) {
--- int size = 0;
--- for (int u=bfs(bfs(r, adj), adj); u!=-1; u=pre[u])
--- path[size++] = u;
--- vector<int> med(1, path[size/2]);
--- if (size % 2 == 0) med.push_back(path[size/2-1]);
--- return med;
} // returns "unique hashcode" for tree with root u
LL rootcode(int u, vector<int> adj[], int p=-1, int d=15){
--- vector<LL> k; int nd = (d + 1) % primes;
--- for (int i = 0; i < adj[u].size(); ++i)
--- if (adj[u][i] != p)
--- k.push_back(rootcode(adj[u][i], adj, u, nd));
--- sort(k.begin(), k.end());
--- LL h = k.size() + 1;
--- for (int i = 0; i < k.size(); ++i)
--- h = h * pr[d] + k[i];
--- return h;
} // returns "unique hashcode" for the whole tree
LL treecode(int root, vector<int> adj[]) {
--- vector<int> c = tree_centers(root, adj);
--- if (c.size()==1)
--- return rootcode(c[0], adj) << 1 | 1;
--- return (rootcode(c[0],adj)*rootcode(c[1],adj))<<1;
} // checks if two trees are isomorphic
bool isomorphic(int r1, vector<int> adj1[], int r2,
--- vector<int> adj2[], bool rooted = false) {
--- if (rooted)
--- return rootcode(r1, adj1) == rootcode(r2, adj2);
--- return treecode(r1, adj1) == treecode(r2, adj2); }
```

5.1. Generating Function Manager.

```
const int DEPTH = 19;
const int ARR_DEPTH = (1<<DEPTH); //approx 5x10^5
const int SZ = 12;
ll temp[SZ][ARR_DEPTH+1];
const ll MOD = 998244353;
struct GF_Manager {
- int tc = 0;
- std::stack<int> to_be_freed;
- const static ll DEPTH = 23;
- ll prim[DEPTH+1], prim_inv[DEPTH+1], two_inv[DEPTH+1];
- ll mod_pow(ll base, ll exp) {
--- if(exp==0) return 1;
--- if(exp&1) return (base*mod_pow(base,exp-1))%MOD;
--- else return mod_pow((base*base)%MOD, exp/2); }
- void set_up_primitives() {
--- prim[DEPTH] = 31;
--- prim_inv[DEPTH] = mod_pow(prim[DEPTH], MOD-2);
--- two_inv[DEPTH] = mod_pow(1<<DEPTH,MOD-2);
--- for(int n = DEPTH-1; n >= 0; n--) {
--- prim[n] = (prim[n+1]*prim[n+1])%MOD;
--- prim_inv[n] = mod_pow(prim[n],MOD-2);
--- two_inv[n] = mod_pow(1<<n,MOD-2); } }
- GF_Manager(){ set_up_primitives(); }
- void start_claiming(){ to_be_freed.push(0); }
- ll* claim()
--- ++to_be_freed.top(); assert(tc < SZ); return temp[tc++]; }
- void end_claiming(){tc-=to_be_freed.top(); to_be_freed.pop();}
- void NTT(ll A[], int n, ll t[],
--- bool is_inverse=false, int offset=0) {
--- if (n==0) return;
--- //Put the evens first, then the odds
--- for (int i = 0; i < (1<<(n-1)); i++) {
--- t[i] = A[offset+2*i];
--- t[i+(1<<(n-1))] = A[offset+2*i+1]; }
--- for(int i = 0; i < (1<<n); i++)
--- A[offset+i] = t[i];
--- NTT(A, n-1, t, is_inverse, offset);
--- NTT(A, n-1, t, is_inverse, offset+(1<<(n-1)));
--- ll w1 = (is_inverse ? prim_inv[n] : prim[n]), w = 1;
--- for (int i = 0; i < (1<<(n-1)); i++, w=(w*w1)%MOD) {
--- t[i] = (A[offset+i] + w*A[offset+(1<<(n-1))+i])%MOD;
--- t[i+(1<<(n-1))] = (A[offset+i]
--- w*A[offset+(1<<(n-1))+i])%MOD; }
--- for (int i = 0; i < (1<<n); i++) A[offset+i] = t[i];
- }
- int add(ll A[], int an, ll B[], int bn, ll C[]) {
--- int cn = 0;
--- for(int i = 0; i < max(an,bn); i++) {
--- C[i] = A[i]+B[i];
--- if(C[i] <= -MOD) C[i] += MOD;
--- if(MOD <= C[i]) C[i] -= MOD;
--- if(C[i]!=0) cn = i; }
- return cn; }
```

```
- int subtract(ll A[], int an, ll B[], int bn, ll C[]) {
-- int cn = 0;
-- for(int i = 0; i < max(an,bn); i++) {
--     C[i] = A[i]-B[i];
--     if(C[i] <= -MOD) C[i] += MOD;
--     if(MOD <= C[i]) C[i] -= MOD;
--     if(C[i]!=0) cn = i; }
-- return cn+1; }
- int scalar_mult(ll v, ll A[], int an, ll C[]) {
-- for(int i = 0; i < an; i++) C[i] = (v*A[i])%MOD;
-- return v==0 ? 0 : an; }
- int mult(ll A[], int an, ll B[], int bn, ll C[]) {
-- start_claiming();
-- // make sure you've called setup prim first
-- // note: an and bn refer to the *number of items in
-- // each array*, NOT the degree of the largest term
-- int n, degree = an+bn-1;
-- for(n=0; (1<n) < degree; n++);
-- ll *tA = claim(), *tB = claim(), *t = claim();
-- copy(A,A+an,tA); fill(tA+an,tA+(1<n),0);
-- copy(B,B+bn,tB); fill(tB+bn,tB+(1<n),0);
-- NTT(tA,n,t);
-- NTT(tB,n,t);
-- for(int i = 0; i < (1<n); i++)
--     tA[i] = (tA[i]*tB[i])%MOD;
-- NTT(tA,n,t,true);
-- scalar_mult(two_inv[n],tA,degree,C);
-- end_claiming();
-- return degree; }
- int reciprocal(ll F[], int fn, ll R[]) {
-- start_claiming();
-- ll *tR = claim(), *tempR = claim();
-- int n; for(n=0; (1<n) < fn; n++);
-- fill(tempR,tempR+(1<n),0);
-- tempR[0] = mod_pow(F[0],MOD-2);
-- for (int i = 1; i <= n; i++) {
--     mult(tempR,1<i,F,1<i,tR);
--     tR[0] -= 2;
--     scalar_mult(-1,tR,1<i,tR);
--     mult(tempR,1<i,tR,1<i,tempR); }
-- copy(tempR,tempR+fn,R);
-- end_claiming();
-- return n; }
- int quotient(ll F[], int fn, ll G[], int gn, ll Q[]) {
-- start_claiming();
-- ll* revF = claim();
-- ll* revG = claim();
-- ll* tempQ = claim();
-- copy(F,F+fn,revF); reverse(revF,revF+fn);
-- copy(G,G+gn,revG); reverse(revG,revG+gn);
-- int qn = fn-gn+1;
-- reciprocal(revG,qn,revG);
-- mult(revF,qn,revG,qn,tempQ);
-- reverse(tempQ,tempQ+qn);
-- copy(tempQ,tempQ+qn,Q);
-- end_claiming();
-- return qn; }
- int mod(ll F[], int fn, ll G[], int gn, ll R[]) {
-- start_claiming();
-- ll *Q = claim(), *GQ = claim();
-- int qn = quotient(F, fn, G, gn, Q);
-- int gqn = mult(G, gn, Q, qn, GQ);
-- int rn = subtract(F, fn, GQ, gqn, R);
-- end_claiming();
-- return rn; }
- ll horners(ll F[], int fn, ll xi) {
-- ll ans = 0;
-- for(int i = fn-1; i >= 0; i--)
--     ans = (ans*xi+F[i]) % MOD;
-- return ans; } };
GF_Manager gfManager;
ll split[DEPTH+1][2*(ARR_DEPTH)+1];
ll Fi[DEPTH+1][2*(ARR_DEPTH)+1];
int bin_splitting(ll a[], int l, int r, int s=0, int offset=0) {
- if(l == r) {
-- split[s][offset] = -a[l]; //x^0
-- split[s][offset+1] = 1; //x^1
-- return 2; }
- int m = (l+r)/2;
- int sz = m-l+1;
- int da = bin_splitting(a, l, m, s+1, offset);
- int db = bin_splitting(a, m+1, r, s+1, offset+(sz<<1));
-- return gfManager.mult(split[s+1]+offset, da,
-- split[s+1]+offset+(sz<<1), db, split[s]+offset); }
void multipoint_eval(ll a[], int l, int r, ll F[], int fn,
- ll ans[], int s=0, int offset=0) {
-- if(l == r) {
--     ans[l] = gfManager.horners(F,fn,a[l]);
--     return; }
- int m = (l+r)/2;
- int sz = m-l+1;
- int da = gfManager.mod(F, fn, split[s+1]+offset,
-- sz+1, Fi[s]+offset);
- int db = gfManager.mod(F, fn, split[s+1]+offset+(sz<<1),
-- r-m+1, Fi[s]+offset+(sz<<1));
-- multipoint_eval(a,l,m,Fi[s]+offset,da,ans,s+1,offset);
-- multipoint_eval(a,m+1,r,Fi[s]+offset+(sz<<1),
-- db,ans,s+1,offset+(sz<<1));
-- }
};
```

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

```
struct poly {
-- double a, b;
-- poly(double a=0, double b=0): a(a), b(b) {}
-- poly operator+(const poly& p) const {
--     return poly(a + p.a, b + p.b);}
-- poly operator-(const poly& p) const {
--     return poly(a - p.a, b - p.b);}
-- poly operator*(const poly& p) const {
--     return poly(a*p.a - b*p.b, a*p.b + b*p.a);}
};
```

```
void fft(poly in[], poly p[], int n, int s) {
-- if (n < 1) return;
-- if (n == 1) {p[0] = in[0]; return;}
-- n >>= 1; fft(in, p, n, s << 1);
-- fft(in + s, p + n, n, s << 1);
-- poly w(1), wn(cos(M_PI/n), sin(M_PI/n));
-- for (int i = 0; i < n; ++i) {
--     poly even = p[i], odd = p[i + n];
--     p[i] = even + w * odd;
--     p[i + n] = even - w * odd;
--     w = w * wn;
-- }
}
void fft(poly p[], int n) {
-- poly *f = new poly[n]; fft(p, f, n, 1);
-- copy(f, f + n, p); delete[] f;
-- }
void inverse_fft(poly p[], int n) {
-- for(int i=0; i<n; i++) {p[i].b *= -1; fft(p, n);
-- for(int i=0; i<n; i++) {p[i].a/=n; p[i].b/= -1*n;}}
-- }
```

5.3. 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)
int mult(int a[],int an,int b[],int bn,int c[]) {
-- int n, degree = an + bn - 1;
-- for (n = 1; n < degree; n <= 1); // power of 2
-- poly *A = new poly[n], *B = new poly[n];
-- copy(a, a + an, A); fill(A + an, A + n, 0);
-- copy(b, b + bn, B); fill(B + bn, B + n, 0);
-- fft(A, n); fft(B, n);
-- for (int i = 0; i < n; i++) A[i] = A[i] * B[i];
-- inverse_fft(A, n);
-- for (int i = 0; i < degree; i++)
--     c[i] = int(A[i].a + 0.5); // same as round(A[i].a)
-- delete[] A, B; return degree;
-- }
```

5.4. Number Theoretic Transform. Other possible moduli: 2113929217(2^{25}), 2013265920268435457(2^{28} , withg = 5)

```
#include "../mathematics/primitive_root.cpp"
int mod = 998244353, g = primitive_root(mod),
- ginv = mod_pow<ll>(g, mod-2, mod),
- inv2 = mod_pow<ll>(2, mod-2, mod);
#define MAXN (1<<22)
struct Num {
-- int x;
-- Num(ll _x=0) { x = (_x%mod+mod)%mod; }
-- Num operator +(const Num &b) { return x + b.x; }
-- Num operator -(const Num &b) const { return x - b.x; }
-- Num operator *(const Num &b) const { return (ll)x * b.x; }
-- Num operator /(const Num &b) const {
--     return (ll)x * b.inv().x; }
-- Num inv() const { return mod_pow<ll>((ll)x, mod-2, mod); }
-- Num pow(int p) const { return mod_pow<ll>((ll)x, p, mod); }
```

```

} T1[MAXN], T2[MAXN];
void ntt(Num x[], int n, bool inv = false) {
- Num z = inv ? ginv : g;
- z = z.pow((mod - 1) / n);
- for (ll i = 0, j = 0; i < n; i++) {
--- if (i < j) swap(x[i], x[j]);
--- ll k = n>>1;
--- while (1 <= k && k <= j) j -= k, k >>= 1;
--- j += k; }
- for (int mx = 1, p = n/2; mx < n; mx <= 1, p >= 1) {
--- Num wp = z.pow(p), w = 1;
--- for (int k = 0; k < mx; k++, w = w*wp) {
----- for (int i = k; i < n; i += mx << 1) {
----- Num t = x[i + mx] * w;
----- x[i + mx] = x[i] - t;
----- x[i] = x[i] + t; } }
- if (inv) {
--- Num ni = Num(n).inv();
--- rep(i,0,n) { x[i] = x[i] * ni; } }
void inv(Num x[], Num y[], int l) {
- if (l == 1) { y[0] = x[0].inv(); return; }
- inv(x, y, l>>1);
- // NOTE: maybe l<<2 instead of l<<1
- rep(i,l>>1,l<<1) T1[i] = y[i] = 0;
- rep(i,0,l) T1[i] = x[i];
- ntt(T1, l<<1); ntt(y, l<<1);
- rep(i,0,l<<1) y[i] = y[i]*2 - T1[i] * y[i] * y[i];
- ntt(y, l<<1, true); }
void sqrt(Num x[], Num y[], int l) {
- if (l == 1) { assert(x[0].x == 1); y[0] = 1; return; }
- sqrt(x, y, l>>1);
- inv(y, T2, l>>1);
- rep(i,l>>1,l<<1) T1[i] = T2[i] = 0;
- rep(i,0,l) T1[i] = x[i];
- ntt(T2, l<<1); ntt(T1, l<<1);
- rep(i,0,l<<1) T2[i] = T1[i] * T2[i];
- ntt(T2, l<<1, true);
- rep(i,0,l) y[i] = (y[i] + T2[i]) * inv2; }

```

5.5. **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;
Poly Q, R; // quotient and remainder
void trim(Poly& A) { // remove trailing zeroes
--- while (!A.empty() && abs(A.back()) < EPS)
--- A.pop_back();
}
void divide(Poly A, Poly B) {
--- if (B.size() == 0) throw exception();
--- if (A.size() < B.size()) {Q.clear(); R=A; return;}
--- Q.assign(A.size() - B.size() + 1, 0);
--- Poly part;
--- while (A.size() >= B.size()) {
----- int As = A.size(), Bs = B.size();
----- part.assign(As, 0);
----- for (int i = 0; i < Bs; i++)

```

```

--- part[As-Bs+i] = B[i];
--- double scale = Q[As-Bs] = A[As-1] / part[As-1];
--- for (int i = 0; i < As; i++)
--- A[i] -= part[i] * scale;
--- trim(A);
--- } R = A; trim(Q); }

```

5.6. **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[][]) {
--- int p = A.length, q = A[0].length, r = B[0].length;
--- // if(q != B.length) throw new Exception(":((");
--- long AB[][] = new long[p][r];
--- for (int i = 0; i < p; i++)
--- for (int j = 0; j < r; j++)
--- for (int k = 0; k < q; k++)
--- (AB[i][j] += A[i][k] * B[k][j]) %= MOD;
--- return AB; }

```

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

```

long[][] power(long B[][], long e) {
--- int n = B.length;
--- long ans[][] = new long[n][n];
--- for (int i = 0; i < n; i++) ans[i][i] = 1;
--- while (e > 0) {
--- if (e % 2 == 1) ans = multiply(ans, B);
--- B = multiply(B, B); e /= 2;
--- } return ans; }

```

5.8. **Fibonacci Matrix.** Fast computation for n th Fibonacci $\{F_1, F_2, \dots, F_n\}$ in $O(\log n)$:

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

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

```

boolean gaussJordan(double A[][]) {
--- int n = A.length, m = A[0].length;
--- boolean singular = false;
--- // double determinant = 1;
--- for (int i=0, p=0; i<n && p<m; i++, p++) {
--- for (int k = i + 1; k < n; k++) {
--- if (Math.abs(A[k][p]) > EPS) { // swap
--- // determinant *= -1;
--- double t[] = A[i]; A[i] = A[k]; A[k] = t;
--- break;
--- }
--- }
--- // determinant *= A[i][p];
--- if (Math.abs(A[i][p]) < EPS)
--- { singular = true; i--; continue; }
--- for (int j = m-1; j >= p; j--) A[i][j] /= A[i][p];
--- for (int k = 0; k < n; k++) {
--- if (i == k) continue;
--- for (int j = m-1; j >= p; j--)
--- A[k][j] -= A[k][p] * A[i][j];

```

```

}
} return !singular; }

```

6. MATH II - COMBINATORICS

6.1. **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
LL lucas(LL n, LL k, int p) {
--- if (k == 0) return 1;
--- if (n < p && k < p) {
--- if (lid != p) {
--- lid = p; f[0] = 1;
--- for (int i = 0; i < p; ++i) f[i] = f[i-1] * i % p;
--- }
--- return f[n] * modpow(f[n-k] * f[k] % p, p-2, p) % p; }
--- return lucas(n/p, k/p, p) * lucas(n%p, k%p, p) % p; }

```

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

```

def fprime(n, p):
--- # counts the number of prime divisors of n!
--- pk, ans = p, 0
--- while pk <= n:
--- ans += n // pk
--- pk *= p
--- return ans
def granville(n, k, p, E):
--- # n choose k (mod p^E)
--- prime_pow = fprime(n,p)-fprime(k,p)-fprime(n-k,p)
--- if prime_pow >= E: return 0
--- e = E - prime_pow
--- pe = p ** e
--- r, f = n - k, [1]*pe
--- for i in range(1, pe):
--- x = i
--- if x % p == 0:
--- x = 1
--- f[i] = f[i-1] * x % pe
--- numer, denom, negate, ptr = 1, 1, 0, 0
--- while n:
--- if f[-1] != 1 and ptr >= e:
--- negate ^= (n&1) ^ (k&1) ^ (r&1)
--- numer = numer * f[n%pe] % pe
--- denom = denom * f[k%pe] % pe * f[r%pe] % pe
--- n, k, r = n//p, k//p, r//p
--- ptr += 1
--- ans = numer * modinv(denom, pe) % pe
--- if negate and (p != 2 or e < 3):
--- ans = (pe - ans) % pe
--- return mod(ans * p**prime_pow, p**E)
def choose(n, k, m): # generalized (n choose k) mod m
--- factors, x, p = [], m, 2
--- while p*p <= x:
--- e = 0
--- while x % p == 0:
--- e += 1

```



```
----- x // = p -----
----- if e: factors.append((p, e)) -----
----- p += 1 -----
--- if x > 1: factors.append((x, 1)) -----
--- crt_array = [granville(n,k,p,e) for p, e in factors] -----
--- mod_array = [p**e for p, e in factors] -----
--- return chinese_remainder(crt_array, mod_array)[0] -----
```

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

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

6.4. **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 -----
typedef long long LL;
void factoradic(int arr[], int n) { // 0 to n-1 -----
--- for (int i = 0; i <= n; i++) fen[i] = 0; -----
--- for (int i = 1; i < n; i++) add(i, 1); -----
--- for (int i = 0; i < n; i++) { -----
--- int s = sum(arr[i]); -----
--- add(arr[i], -1); arr[i] = s; -----
--- } -----
void permute(int arr[], int n) { // factoradic to perm -----
--- for (int i = 0; i <= n; i++) fen[i] = 0; -----
--- for (int i = 1; i < n; i++) add(i, 1); -----
--- for (int i = 0; i < n; i++) { -----
--- arr[i] = low(arr[i] - 1); -----
--- add(arr[i], -1); -----
--- }
```

6.5. **k th Permutation.** Get the next k th 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) { -----
--- factoradic(arr, n); // values from 0 to n-1 -----
--- for (int i = n-1; i >= 0 && k > 0; --i){ -----
--- LL temp = arr[i] + k; -----
--- arr[i] = temp % (n - i); -----
--- k = temp / (n - i); -----
--- } -----
--- permute(arr, n); -----
--- return k == 0; }
```

6.6. **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 $\{1, \dots, n\}$ without a 3-term increasing subsequence

- (8) The number of ways to form a mountain range with n ups and n downs

6.7. **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

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

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

6.8. **Partition Function.** Pregenerate the number of partitions of positive integer n with n positive addends.

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

7. MATH III - NUMBER THEORY

7.1. **Number/Sum of Divisors.** If a number n is prime factorized where $n = p_1^{e_1} \times p_2^{e_2} \times \dots \times p_k^{e_k}$, where σ_0 is the number of divisors while σ_1 is the sum of divisors:

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

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

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

```
std::bitset<N> is; int mu[N]; -----
void mobiusSieve() { -----
- for (int i = 1; i < N; ++i) mu[i] = 1; -----
- for (int i = 2; i < N; ++i) if (!is[i]) { -----
--- for (int j = i; j < N; j += i) { is[j] = 1; mu[j] *= -1; } -----
--- for (long long j = 1LL*i*i; j < N; j += i*i) mu[j] = 0; } }
```

7.3. **Möbius Inversion.** Given arithmetic functions f and g :

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

7.4. **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 -----
long long gcnt[MX+1]; // gcnt[G]: answer when gcd==G -----
long long C(int f) {return (1LL << f) - 1;} -----
// f: frequency count -----
// C(f): # of subsets of f elements (YOU CAN EDIT) -----
// Usage: int subsets_with_gcd_1 = gcnt[1]; -----
void gcd_counter(int a[], int n) { -----
- memset(f, 0, sizeof f); -----
- memset(gcnt, 0, sizeof gcnt); -----
```

```
- int mx = 0; -----
- for (int i = 0; i < n; ++i) { -----
--- f[a[i]] += 1; -----
--- mx = max(mx, a[i]); } -----
- for (int i = mx; i >= 1; --i) { -----
--- int add = f[i]; -----
--- long long sub = 0; -----
--- for (int j = 2*i; j <= mx; j += i) { -----
--- add += f[j]; -----
--- sub += gcnt[j]; } -----
--- gcnt[i] = C(add) - sub; }
```

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

```
ll totient(ll n) { -----
- if (n <= 1) return 1; -----
- ll tot = n; -----
- for (int i = 2; i * i <= n; i++) { -----
--- if (n % i == 0) tot -= tot / i; -----
--- while (n % i == 0) n /= i; } -----
- if (n > 1) tot -= tot / n; -----
- return tot; }
```

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

```
ll mod(ll x, ll m) { // use this instead of x % m -----
- if (m == 0) return 0; -----
- if (m < 0) m *= -1; -----
- return (x%m + m) % m; // always nonnegative -----
} -----
ll extended_euclid(ll a, ll b, ll &x, ll &y) { -----
- if (b==0) {x = 1; y = 0; return a;} -----
- ll g = extended_euclid(b, a%b, x, y); -----
- ll z = x - a/b*y; -----
- x = y; y = z; return g; -----
}
```

7.7. **Modular Exponentiation.** Find $b^e \pmod m$ in $O(\log e)$ time.

```
template <class T> -----
T mod_pow(T b, T e, T m) { -----
- T res = T(1); -----
- while (e) { -----
--- if (e & T(1)) res = smod(res * b, m); -----
--- b = smod(b * b, m), e >>= T(1); } -----
- return res; }
```

7.8. **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) { -----
- ll x, y; ll g = extended_euclid(a, m, x, y); -----
- if (g == 1 || g == -1) return mod(x * g, m); -----
- return 0; // 0 if invalid }
```

7.9. **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 $x \pmod M$.

```
pll modsolver(ll a, ll b, ll m) {
- ll x, y; ll g = extended_euclid(a, m, x, y);
- if (b % g != 0) return {-1, -1};
- return {mod(x*b/g, m/g), abs(m/g)}; }
```

7.10. **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.

```
pll null(-1, -1); // needs extended euclidean
pll diophantine(ll a, ll b, ll c) {
- if (!a && !b) return c ? null : {0, 0};
- if (!a) return c % b ? null : {0, c / b};
- if (!b) return c % a ? null : {c / a, 0};
- ll x, y; ll g = extended_euclid(a, b, x, y);
- if (c % g) return null;
- y = mod(y * (c/g), a/g);
- if (y == 0) y += abs(a/g); // prefer positive sol.
- return {(c - b*y)/a, y}; }
```

7.11. **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 $x \bmod M$.

```
pll chinese(ll b1, ll m1, ll b2, ll m2) {
- ll x, y; ll g = extended_euclid(m1, m2, x, y);
- if (b1 % g != b2 % g) return ii(-1, -1);
- ll M = abs(m1 / g * m2);
- return {mod(mod(x*b2*m1+y*b1*m2, M*g)/g, M), M}; }
ii chinese_remainder(ll b[], ll m[], int n) {
- ii ans(0, 1);
- for (int i = 0; i < n; ++i) {
--- ans = chinese(b[i], m[i], ans.first, ans.second);
--- if (ans.second == -1) break; }
- return ans; }
```

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

```
pll super_chinese(ll a[], ll b[], ll m[], int n) {
- pll ans(0, 1);
- for (int i = 0; i < n; ++i) {
--- pll two = modsolver(a[i], b[i], m[i]);
--- if (two.second == -1) return two;
--- ans = chinese(ans.first, ans.second,
--- two.first, two.second);
--- if (ans.second == -1) break; }
- return ans; }
```

7.12. **Primitive Root.**

```
#include "mod_pow.cpp"
ll primitive_root(ll m) {
- vector<ll> div;
- for (ll i = 1; i*i <= m-1; i++) {
--- if ((m-1) % i == 0) {
----- if (i < m) div.push_back(i);
----- if (m/i < m) div.push_back(m/i); } }
- rep(x, 2, m) {
--- bool ok = true;
--- iter(it, div) if (mod_pow<ll>(x, *it, m) == 1) {
```

```
----- ok = false; break; }
--- if (ok) return x; }
- return -1; }
```

7.13. **Josephus.** Last man standing out of n if every k th is killed. Zero-based, and does not kill 0 on first pass.

```
int J(int n, int k) {
- if (n == 1) return 0;
- if (k == 1) return n-1;
- if (n < k) return (J(n-1, k)+k)%n;
- int np = n - n/k;
- return k*((J(np, k)+np-n*k*np)%np) / (k-1); }
```

7.14. **Number of Integer Points under a Lines.** Count the number of integer solutions to $Ax + By \leq C$, $0 \leq x \leq n$, $0 \leq y$. In other words, evaluate the sum $\sum_{x=0}^n \left\lfloor \frac{C-Ax}{B} + 1 \right\rfloor$. To count all solutions, let $n = \left\lfloor \frac{C}{A} \right\rfloor$. In any case, it must hold that $C - nA \geq 0$. Be very careful about overflows.

8. MATH IV - NUMERICAL METHODS

8.1. **Fast Square Testing.** An optimized test for square integers.

```
long long M;
void init_is_square() {
- rep(i, 0, 64) M |= 1ULL << (63-(i*i)%64); }
inline bool is_square(ll x) {
- if (x == 0) return true; // XXX
- if ((M << x) >= 0) return false;
- int c = __builtin_ctz(x);
- if (c & 1) return false;
- x >>= c;
- if ((x&7) - 1) return false;
- ll r = sqrt(x);
- return r*r == x; }
```

8.2. **Simpson Integration.** Use to numerically calculate integrals

```
const int N = 1000 * 1000; // number of steps
double simpson_integration(double a, double b){
- double h = (b - a) / N;
- double s = f(a) + f(b); // a = x_0 and b = x_2n
- for (int i = 1; i <= N - 1; ++i) {
--- double x = a + h * i;
--- s += f(x) * ((i & 1) ? 4 : 2); }
- s *= h / 3;
- return s; }
```

9. STRINGS

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

```
int par[N]; // parent table
void buildKMP(string& f) {
- par[0] = -1, par[1] = 0;
- int i = 2, j = 0;
- while (i <= f.length()) {
--- if (f[i-1] == f[j]) par[i++] = ++j;
--- else if (j > 0) j = par[j];
```

```
--- else par[i++] = 0; } }
std::vector<int> KMP(string& s, string& f) {
- buildKMP(f); // call once if f is the same
- int i = 0, j = 0; vector<int> ans;
- while (i + j < s.length()) {
--- if (s[i + j] == f[j]) {
----- if (++j == f.length()) {
----- ans.push_back(i);
----- i += j - par[j];
----- if (j > 0) j = par[j]; }
--- } else {
--- i += j - par[j];
--- if (j > 0) j = par[j]; }
- } return ans; }
```

9.2. **Trie.**

```
template <class T>
struct trie {
- struct node {
--- map<T, node*> children;
--- int prefixes, words;
--- node() { prefixes = words = 0; } };
- node* root;
- trie() : root(new node()) { }
- template <class I>
- void insert(I begin, I end) {
--- node* cur = root;
--- while (true) {
----- cur->prefixes++;
----- if (begin == end) { cur->words++; break; }
----- else {
----- T head = *begin;
----- typename map<T, node*>::const_iterator it;
----- it = cur->children.find(head);
----- if (it == cur->children.end()) {
----- pair<T, node*> nw(head, new node());
----- it = cur->children.insert(nw).first;
----- } begin++, cur = it->second; } } }
- template <class I>
- int countMatches(I begin, I end) {
--- node* cur = root;
--- while (true) {
--- if (begin == end) return cur->words;
--- else {
----- T head = *begin;
----- typename map<T, node*>::const_iterator it;
----- it = cur->children.find(head);
----- if (it == cur->children.end()) return 0;
----- begin++, cur = it->second; } } }
- template <class I>
- int countPrefixes(I begin, I end) {
--- node* cur = root;
--- while (true) {
--- if (begin == end) return cur->prefixes;
--- else {
----- T head = *begin;
```

```
----- typename map<T, node*>::const_iterator it;
----- it = cur->children.find(head);
----- if (it == cur->children.end()) return 0;
----- begin++, cur = it->second; } } };
```

9.2.1. Persistent Trie.

```
const int MAX_KIDS = 2;
const char BASE = '0'; // 'a' or 'A'
struct trie {
- int val, cnt;
- std::vector<trie*> kids;
- trie () : val(-1), cnt(0), kids(MAX_KIDS, NULL) {}
- trie (int val) : val(val), cnt(0), kids(MAX_KIDS, NULL) {}
- trie (int val, int cnt, std::vector<trie*> &n_kids) :
-- val(val), cnt(cnt), kids(n_kids) {}
- trie *insert(std::string &s, int i, int n) {
-- trie *n_node = new trie(val, cnt+1, kids);
-- if (i == n) return n_node;
-- if (!n_node->kids[s[i]-BASE])
-- n_node->kids[s[i]-BASE] = new trie(s[i]);
-- n_node->kids[s[i]-BASE] =
---- n_node->kids[s[i]-BASE]->insert(s, i+1, n);
-- return n_node; } };
// max xor on a binary trie from version `a+1` to `b` (b > a):
int get_max_xor(trie *a, trie *b, int x) {
- int ans = 0;
- for (int i = MAX_BITS; i >= 0; --i) {
-- // don't flip the bit for min xor
-- int u = ((x & (1 << i)) > 0) ^ 1;
-- int res_cnt = (b and b->kids[u] ? b->kids[u]->cnt : 0)
-- (a and a->kids[u] ? a->kids[u]->cnt : 0);
-- if (res_cnt == 0) u ^= 1;
-- ans ^= (u << i);
-- if (a) a = a->kids[u];
-- if (b) b = b->kids[u]; }
- return ans; }
```

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

```
int n, equiv[N+1], suffix[N+1];
ii equiv_pair[N+1];
string T;
void make_suffix_array(string& s) {
- if (s.back()!='$') s += '$';
- n = s.length();
- for (int i = 0; i < n; i++)
-- suffix[i] = i;
- sort(suffix, suffix+n, [&s](int i, int j){return s[i] < s[j]});
- int sz = 0;
- for(int i = 0; i < n; i++){
-- if(i==0 || s[suffix[i]]!=s[suffix[i-1]])
-- ++sz;
-- equiv[suffix[i]] = sz; }
- for (int t = 1; t < n; t<=1) {
-- for (int i = 0; i < n; i++)
-- equiv_pair[i] = {equiv[i],equiv[(i+t)%n]};
-- sort(suffix, suffix+n, [](int i, int j) {
```

```
----- return equiv_pair[i] < equiv_pair[j]});
----- int sz = 0;
----- for (int i = 0; i < n; i++) {
----- if(i==0 || equiv_pair[suffix[i]]!=equiv_pair[suffix[i-1]])
----- ++sz;
----- equiv[suffix[i]] = sz; } } }
int count_occurrences(string& G) { // in string T
- int L = 0, R = n-1;
- for (int i = 0; i < G.length(); i++){
-- // lower/upper = first/last time G[i] is
-- // the ith character in suffixes from [L,R]
-- std::tie(L,R) = {lower(G[i],i,L,R), upper(G[i],i,L,R)};
-- if (L==-1 && R==-1) return 0; }
- return R-L+1; }
```

9.4. 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]:])
void buildLCP(std::string s) { // build suffix array first
- for (int i = 0, k = 0; i < n; i++) {
-- if (pos[i] != n - 1) {
---- for(int j = sa[pos[i]+1]; s[i+k]==s[j+k];k++);
---- lcp[pos[i]] = k; if (k > 0) k--;
-- } else { lcp[pos[i]] = 0; } }
```

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

```
class Node {
- HashMap<Character, Node> next = new HashMap<>();
- Node fail = null;
- long count = 0;
- public void add(String s) { // adds string to trie
-- Node node = this;
-- for (char c : s.toCharArray()) {
--- if (!node.contains(c))
---- node.next.put(c, new Node());
--- node = node.get(c);
-- } node.count++; }
- public void prepare() {
-- // prepares fail links of Aho-Corasick Trie
-- Node root = this; root.fail = null;
-- Queue<Node> q = new ArrayDeque<Node>();
-- for (Node child : next.values()) // BFS
-- { child.fail = root; q.offer(child); }
-- while (!q.isEmpty()) {
-- Node head = q.poll();
-- for (Character letter : head.next.keySet()) {
--- // traverse upwards to get nearest fail link
--- Node p = head;
--- Node nextNode = head.get(letter);
--- do { p = p.fail; }
--- while(p != root && !p.contains(letter));
--- if (p.contains(letter)) { // fail link found
---- p = p.get(letter);
---- nextNode.fail = p;
---- nextNode.count += p.count;
```

```
----- } else { nextNode.fail = root; }
----- q.offer(nextNode); } } }
public BigInteger search(String s) {
-- // counts the words added in trie present in s
-- Node root = this, p = this;
-- BigInteger ans = BigInteger.ZERO;
-- for (char c : s.toCharArray()) {
--- while (p != root && !p.contains(c)) p = p.fail;
--- if (p.contains(c)) {
---- p = p.get(c);
---- ans = ans.add(BigInteger.valueOf(p.count)); }
-- } return ans; }
- private Node get(char c) { return next.get(c); }
- private boolean contains(char c) {
-- return next.containsKey(c); }
// Usage: Node trie = new Node();
// for (String s : dictionary) trie.add(s);
// trie.prepare(); BigInteger m = trie.search(str);
```

9.6. Palindromes.

9.6.1. 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];
int len[N*2+1], node[N*2+1], cs[N*2+1], size;
long long cnt[N + 2]; // count can be very large
int newNode(int p = -1) {
- cnt[size] = 0; par[size] = p;
- len[size] = (p == -1 ? 0 : len[p] + 2);
- memset(child[size], -1, sizeof child[size]);
- return size++; }
int get(int i, char c) {
- if (child[i][c] == -1) child[i][c] = newNode(i);
- return child[i][c]; }
void manachers(char s[]) {
- int n = strlen(s), cn = n * 2 + 1;
- for (int i = 0; i < n; i++) {
-- cs[i * 2] = -1; cs[i * 2 + 1] = s[i]; }
- size = n * 2;
- int odd = newNode(), even = newNode();
- int cen = 0, rad = 0, L = 0, R = 0;
- size = 0; len[odd] = -1;
- for (int i = 0; i < cn; i++)
-- node[i] = (i % 2 == 0 ? even : get(odd, cs[i]));
- for (int i = 1; i < cn; i++) {
-- if (i > rad) { L = i - 1; R = i + 1; }
-- else {
--- int M = cen * 2 - i; // retrieve from mirror
--- node[i] = node[M];
--- if (len[node[M]] < rad - i) L = -1;
--- else {
---- R = rad + 1; L = i * 2 - R;
---- while (len[node[i]] > rad - i)
----- node[i] = par[node[i]]; } } // expand palindrome
-- while (L >= 0 && R < cn && cs[L] == cs[R]) {
```

```
----- if (cs[L] != -1) node[i] = get(node[i],cs[L]);
----- L--, R++; }
cnt[node[i]]++;
if (i + len[node[i]] > rad) {
rad = i + len[node[i]]; cen = i; } }
for (int i = size - 1; i >= 0; --i)
cnt[par[i]] += cnt[i]; // update parent count }
int countUniquePalindromes(char s[]) {
manachers(s); return size; }
int countAllPalindromes(char s[]) {
manachers(s); int total = 0;
for (int i = 0; i < size; i++) total += cnt[i];
return total; }
// longest palindrome substring of s
std::string longestPalindrome(char s[]) {
manachers(s);
int n = strlen(s), cn = n * 2 + 1, mx = 0;
for (int i = 1; i < cn; i++)
if (len[node[mx]] < len[node[i]])
mx = i;
int pos = (mx - len[node[mx]]) / 2;
return std::string(s + pos, s + pos + len[node[mx]]); }
```

9.6.2. *Eertree*.

```
struct node {
int start, end, len, back_edge, *adj;
node() {
adj = new int[26];
for (int i = 0; i < 26; ++i) adj[i] = 0; }
node(int start, int end, int len, int back_edge) :
start(start), end(end), len(len), back_edge(back_edge) {
adj = new int[26];
for (int i = 0; i < 26; ++i) adj[i] = 0; } };
struct eertree {
int ptr, cur_node;
std::vector<node> tree;
eertree () {
tree.push_back(node());
tree.push_back(node(0, 0, -1, 1));
tree.push_back(node(0, 0, 0, 1));
cur_node = 1;
ptr = 2; }
int get_link(int temp, std::string &s, int i) {
while (true) {
int cur_len = tree[temp].len;
// don't return immediately if you want to
// get all palindromes; not recommended
if (i-cur_len-1 >= 0 and s[i] == s[i-cur_len-1])
return temp;
temp = tree[temp].back_edge; }
return temp; }
void insert(std::string &s, int i) {
int temp = cur_node;
temp = get_link(temp, s, i);
if (tree[temp].adj[s[i] - 'a'] != 0) {
cur_node = tree[temp].adj[s[i] - 'a'];
```

```
return; }
ptr++;
tree[temp].adj[s[i] - 'a'] = ptr;
int len = tree[temp].len + 2;
tree.push_back(node(i-len+1, i, len, 0));
temp = tree[temp].back_edge;
cur_node = ptr;
if (tree[cur_node].len == 1) {
tree[cur_node].back_edge = 2;
return; }
temp = get_link(temp, s, i);
tree[cur_node].back_edge = tree[temp].adj[s[i]-'a']; }
void insert(std::string &s) {
for (int i = 0; i < s.size(); ++i)
insert(s, i); }
```

9.7. **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:])
void computeZ(string s) {
int n = s.length(), L = 0, R = 0; z[0] = n;
for (int i = 1; i < n; i++) {
if (i > R) {
L = R = i;
while (R < n && s[R - L] == s[R]) R++;
z[i] = R - L; R--;
} else {
int k = i - L;
if (z[k] < R - i + 1) z[i] = z[k];
else {
L = i;
while (R < n && s[R - L] == s[R]) R++;
z[i] = R - L; R--; } } }
```

9.8. **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];
int booth(string S) {
S.append(S); // concatenate itself
int n = S.length(), i, j, k = 0;
memset(f, -1, sizeof(int) * n);
for (j = 1; j < n; j++) {
i = f[j-k-1];
while (i != -1 && S[j] != S[k + i + 1]) {
if (S[j] < S[k + i + 1]) k = j - i - 1;
i = f[i];
} if (i == -1 && S[j] != S[k + i + 1]) {
if (S[j] < S[k + i + 1]) k = j;
f[j - k] = -1;
} else f[j - k] = i + 1;
} return k; }
```

9.9. **Hashing**.

9.9.1. *Rolling Hash*.

```
int MAXN = 1e5+1, MOD = 1e9+7;
struct hasher {
```

```
int n;
std::vector<ll> *p_pow, *h_ans;
hash(vi &s, vi primes) : n(primes.size()) {
p_pow = new std::vector<ll>[n];
h_ans = new std::vector<ll>[n];
for (int i = 0; i < n; ++i) {
p_pow[i] = std::vector<ll>(MAXN);
p_pow[i][0] = 1;
for (int j = 0; j+1 < MAXN; ++j)
p_pow[i][j+1] = (p_pow[i][j] * primes[i]) % MOD;
h_ans[i] = std::vector<ll>(MAXN);
h_ans[i][0] = 0;
for (int j = 0; j < s.size(); ++j)
h_ans[i][j+1] = (h_ans[i][j] +
s[j] * p_pow[i][j]) % MOD; } } ;
```