



Indian Institute of Technology Delhi

# Audit Pass

Harsh Wardhan, Rishabh Dhiman, Tamajit Banerjee

ACM-ICPC World Finals 2023

18 April 2024

```

1 Time:  $\mathcal{O}(\log N)$  f2b2fb, 16 lines
   #include <bits/extc++.h>
1   using namespace __gnu_pbds;

   template<class T>
4   using ost = tree<T, null_type, less<T>, rb_tree_tag,
       tree_order_statistics_node_update>;

5   void example() {
       ost<int> t, t2; t.insert(8);
9       auto it = t.insert(10).first;
       assert(it == t.lower_bound(9));
       assert(t.order_of_key(10) == 1);
11      assert(t.order_of_key(11) == 2);
       assert(*t.find_by_order(0) == 8);
17      t.join(t2); // assuming  $T < T2$  or  $T > T2$ , merge  $t2$  into  $t$ 
   }

```

## 22 HashMap.h

**Description:** Hash map with mostly the same API as unordered\_map, but  
~3x faster. Uses 1.5x memory. Initial capacity must be a power of 2 (if  
24 provided)

```

Time:  $\mathcal{O}(\log N)$ .
fibc35, 54 lines

template <class S, S (*op)(S, S), S (*e)(), class F,
        S (*mapping)(F, S), F (*composition)(F, F), F (*id)()>
struct lazy_segtree {
    int n; vector<S> t; vector<F> lz;
    lazy_segtree(int n = 0) : n{n} {
        int sz = 1; while (sz < n) sz *= 2;
        t.assign(2 * sz, e()); lz.assign(sz, id());
    }

    void apply_node(int v, F f) {
        if (v < sz(lz)) lz[v] = composition(f, lz[v]);
        t[v] = mapping(f, t[v]);
    }

    void push(int v) {
        if (v < sz(lz)) {
            apply_node(v << 1 | 0, lz[v]);
            apply_node(v << 1 | 1, lz[v]);
            lz[v] = id();
        }
    }

    void pull(int v) { t[v] = op(t[v << 1], t[v << 1 | 1]); }

    S prod(int lo, int hi) { // query [lo, hi)
        auto rec = [&](auto self, int v, int l, int r) -> S {
            if (r <= lo || hi <= l) return e();
            if (lo <= l && r <= hi) return t[v];
            push(v); auto m = l + (r - l) / 2;
            return op(self(self, v << 1 | 0, l, m),
                      self(self, v << 1 | 1, m, r));
        };
        return rec(rec, 1, 0, n);
    }

    void apply(int lo, int hi, F f) { // apply f to [lo, hi)
        auto rec = [&](auto self, int v, int l, int r) -> void {
            if (r <= lo || hi <= l) return;
            if (lo <= l && r <= hi) return apply_node(v, f);
            push(v); auto m = l + (r - l) / 2;
            self(self, v << 1 | 0, l, m);
            self(self, v << 1 | 1, m, r);
            pull(v);
        };
        rec(rec, 1, 0, n);
    }

    void set(int p, S x) {
        auto rec = [&](auto self, int v, int l, int r) -> void {
            if (r - l == 1) return t[v] = x, void();
            push(v); auto m = l + (r - l) / 2;
            if (p < m) self(self, v << 1 | 0, l, m);
            else self(self, v << 1 | 1, m, r);
            pull(v);
        };
        rec(rec, 1, 0, n);
    }
};

```

```
#include <bits/stdc++.h>
using namespace std;

#define rep(i, a, b) for (auto i{a}; i < (b); ++i)
#define per(i, a, b) for (auto i{b}; i-- > (a); )
#define all(x) begin(x), end(x)
#define all(x) (x).rbegin(), (x).rend()
#define sz(x) (int)(x).size()

template <class T> bool uax(T& a, const T& b) { return a < b ?
    a = b, true : false; }
template <class T> bool uin(T& a, const T& b) { return a > b ?
    a = b, true : false; }

using ll = long long;

mt19937 rng(chrono::steady_clock::now().time_since_epoch().
    count());

signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
}
```

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++17 \
-fsanitize=undefined,address'
```

```
# Hashes a file, ignoring all whitespace and comments. Use for
# verifying that code was correctly typed.
cpp -dD -P -fpreprocessed | tr -d '[:space:]' | md5sum | cut -c-6
```

```

const uint64_t C = 1l(4e18 * acos(0)) | 71;
ll operator()(ll x) const { return __builtin_bswap64(x*C); }
};

__gnu_pbds::gp_hash_table<ll,int, chash> h({}, {}, {}, {}, {}, {1<<16});

```

## Segtree.h

**Usage:** using S = int;  
S op(S a, S b) { return max(a, b); }  
S e() { return -inf; }  
segtree<S, op, e> t(n);  
**Time:**  $O(\log N)$

---

d1a9cf, 20 lines

```

template <class S, S (*op)(S, S), S (*e)()>
struct segtree {
    int sz; vector<S> t;
    segtree(int n = 0) : sz{1} { // can replace with just n
        while (sz < n) sz *= 2;
        t.assign(2 * sz, e());
    }
    void set(int pos, S val) {
        for (t[pos += sz] = val; pos /= 2;)
            t[pos] = op(t[pos << 1], t[pos << 1|1]);
    }
    S prod(int l, int r) { // query [l, r)
        S a = e(), b = e();
        for (l += sz, r += sz; l < r; l /= 2, r /= 2) {
            if (l & 1) a = op(a, t[l++]);
            if (r & 1) b = op(t[--r], b);
        }
        return op(a, b);
    }
};

```

```
Usage: using S = int;
S op(S a, S b) { return max(a, b); }
S e() { return -inf; }
using F = int;
S mapping(F f, S x) { return x + f; }
F composition(F f, F g) { return f + g; }
F id() { return 0; }
lazy_seqtree<S, op, e, F, mapping, composition, id> t(n);
```

**Description:** Computes max  $r$  such that  $g[l, r)$  is true for a monotone  $g$ .  $g(e(i))$  must be true. Add to segtree struct, works on both segment trees.

**Usage:** `t.max_right(l, [x](S y) { return y < x; });`  
 Leftmost  $r \geq l$  with value  $\geq x$ , assuming  $t$  stores  $\max$ .

**Time:**  $\mathcal{O}(\log N)$ .

---

"LazySegmentTree.h" 160b8b, 21 lines

```
template <class G> int max_right(int p, G g) {
    assert(g(e(p))); S x(e(p));
    auto rec = [&](auto self, int v, int l, int r) -> int {
        if (r - l == 1) {
            auto nxt_x = op(x, t[v]);
```

```
        return g(nxt_x) ? x = nxt_x, r : l;
    }
    push(v); // remove for non-lazy segtree
    auto m = 1 + (r - l) / 2;
    if (p < m) {
        if (p <= l) {
            auto nxt_x = op(x, t[v]);
            if (g(nxt_x)) return x = nxt_x, r;
        }
        auto idx = self(self, v << 1 | 0, l, m);
        if (idx < m) return idx;
    }
    return self(self, v << 1 | 1, m, r);
};
return rec(rec, l, 0, n); // n -> sz for non-lazy
}
```

SegtreeMinLeft.h

**Description:** Computes min  $l$  such that  $g[l,r]$  is true for a monotone  $g$ .  $g(e())$  must be true. Add to segtree struct, works on both segment trees.  
**Usage:**  $t.min\_left(r, [x](S\ y)\ \{ \text{return } y < x; \})$ ;  
Rightmost  $l \leq r$  with value at  $l-1 \geq x$ , assuming  $t$  stores max.  
**Time:**  $\mathcal{O}(\log N)$ .

"LazySegmentTree.h"	9240cc, 21 lines
<pre>template &lt;class G&gt; int min_left(int p, G g) {     assert(g(e())); S x{e()};     auto rec = [&amp;](auto self, int v, int l, int r) -&gt; int {         if (r - l == 1) {             auto nxt_x = op(t[v], x);             return g(nxt_x) ? x = nxt_x, l : r;         }         push(v); // remove for non-lazy segtree         auto m = 1 + (r - l) / 2;         if (p &gt; m) {             if (p &gt;= r) {                 auto nxt_x = op(t[v], x);                 if (g(nxt_x)) return x = nxt_x, l;             }             auto idx = self(self, v &lt;&lt; 1   1, m, r);             if (idx &gt; m) return idx;         }         return self(self, v &lt;&lt; 1   0, l, m);     };     return rec(rec, l, 0, n); // n -&gt; sz for non-lazy }</pre>	

SegtreeBeats.h

**Description:** For supporting range  $A_i \leftarrow \min(A_i, x)$  operations, break if  $\max < x$  and apply lazy update if second  $\max < x$ .  
**Time:**  $\mathcal{O}(\log N)$  amortized

"LazySegmentTree.h"	
<p>PersistentLazySegtree.h</p> <p><b>Description:</b> Same syntax as lazy segtree. Use bump allocator, SmallPtr or implicit indices for better performance. Remove instances of F to make it non-lazy, don't remove push(), children if empty need to be created. <b>Usage:</b> using T = persistent_lazysegtree&lt;...&gt;; T t(n); T::Node* root{nullptr}; root = t.set(root, i, x); root = t.apply(root, l, r, f); auto res = t.prod(root, l, r); <b>Time:</b> <math>\mathcal{O}(\log N)</math>.</p>	

template <class S, S (*op)(S, S), S (*e)(), class F, S (*mapping)(F, S), F (*composition)(F, F), F (*id)()>	
struct persistent_lazysegtree {	
struct Node {	
Node *l = nullptr, *r = nullptr;	03dd11, 58 lines

```
S val = e(); F lz = id();

void apply(F f) {
    lz = composition(f, lz);
    val = mapping(f, val);
}
void push() {
    create(l)->apply(lz);
    create(r)->apply(lz);
    lz = id();
}
void pull() { val=op(l ? l->val : e(), r ? r->val : e()); }
};
static Node* create(Node* &x) {
    return x = x ? new Node(*x) : new Node();
}

int n;
persistent_lazysegtree(int n = 0) : n{n} {}

S prod(Node* root, int lo, int hi) { // [lo, hi)
    auto rec = [&](auto self, Node* v, int l, int r) -> S {
        if (r <= lo || hi <= l) return e();
        if (lo <= l && r <= hi) return v->val;
        v->push(); auto m = 1 + (r - l) / 2;
        return op(self(self, v->l, l, m),
                  self(self, v->r, m, r));
    };
    return root ? rec(rec, root, 0, n) : e();
}
Node* apply(Node* root, int lo, int hi, F f) { // [lo, hi)
    auto rec = [&](auto self, Node* v, int l, int r) -> void {
        // return v to get new node of [l, r)
        if (r <= lo || hi <= l) return;
        if (lo <= l && r <= hi) return v->apply(f);
        v->push(); auto m = 1 + (r - l) / 2;
        self(self, v->l, l, m);
        self(self, v->r, m, r);
        v->pull();
    };
    return rec(rec, create(root), 0, n), root;
}
Node* set(Node* root, int p, S x) {
    auto rec = [&](auto self, Node* v, int l, int r) -> void {
        if (r - l == 1) return v->val = x, void();
        v->push(); auto m = 1 + (r - l) / 2;
        if (p < m) self(self, v->l, l, m);
        else self(self, v->r, m, r);
        v->pull();
    };
    return rec(rec, create(root), 0, n), root;
}
};
```

S prod(Node* root, int lo, int hi) { // [lo, hi)	
auto rec = [&](auto self, Node* v, int l, int r) -> S {	
if (r <= lo    hi <= l) return e();	
if (lo <= l && r <= hi) return v->val;	
v->push(); auto m = 1 + (r - l) / 2;	
return op(self(self, v->l, l, m),	
self(self, v->r, m, r));	
};	
return root ? rec(rec, root, 0, n) : e();	
}	
Node* apply(Node* root, int lo, int hi, F f) { // [lo, hi)	
auto rec = [&](auto self, Node* v, int l, int r) -> void {	
// return v to get new node of [l, r)	
if (r <= lo    hi <= l) return;	
if (lo <= l && r <= hi) return v->apply(f);	
v->push(); auto m = 1 + (r - l) / 2;	
self(self, v->l, l, m);	
self(self, v->r, m, r);	
v->pull();	
};	
return rec(rec, create(root), 0, n), root;	
}	
Node* set(Node* root, int p, S x) {	
auto rec = [&](auto self, Node* v, int l, int r) -> void {	
if (r - l == 1) return v->val = x, void();	
v->push(); auto m = 1 + (r - l) / 2;	
if (p < m) self(self, v->l, l, m);	
else self(self, v->r, m, r);	
v->pull();	
};	
return rec(rec, create(root), 0, n), root;	
}	
};	258544, 1 lines

ImplicitLazySegtree.h

**Description:** Replace create() in persistent lazy segtree.  
**Usage:** T t(n); T::Node\* root{nullptr};  
t.create(root); // Important, empty tree otherwise

"PersistentLazySegtree.h"	258544, 1 lines
static Node* create(Node* &x) { return x ? : x = new Node(); }	
<p>RMQ.h</p> <p><b>Description:</b> Returns <math>\min(V[a], V[a+1], \dots, V[b-1])</math> in constant time. <b>Usage:</b> RMQ rmq(values); rmq.query(inclusive, exclusive); <b>Time:</b> <math>\mathcal{O}( V  \log  V  + Q)</math></p>	

template<class T>	
struct RMQ {	510c32, 16 lines

```
vector<vector<T>> jmp;
RMQ(const vector<T>& V) : jmp(1, V) {
    for (int pw = 1, k = 1; pw * 2 <= sz(V); pw *= 2, ++k) {
        jmp.emplace_back(sz(V) - pw * 2 + 1);
        rep(j, 0, sz(jmp[k]))
            jmp[k][j] = min(jmp[k - 1][j], jmp[k - 1][j + pw]);
    }
}
T query(int a, int b) {
    assert(a < b); // or return inf if a == b
    int dep = 31 - __builtin_clz(b - a);
    return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
}
};
```

SubMatrix.h

**Description:** Calculate submatrix sums quickly, given upper-left and lower-right corners (half-open).  
**Usage:** SubMatrix<int> m(matrix);  
m.sum(0, 0, 2, 2); // top left 4 elements  
**Time:**  $\mathcal{O}(N^2 + Q)$

template<class T>	c59ada, 13 lines
struct SubMatrix {	
vector<vector<T>> p;	
SubMatrix(vector<vector<T>>& v) {	
int R = sz(v), C = sz(v[0]);	
p.assign(R+1, vector<T>(C+1));	
rep(r, 0, R) rep(c, 0, C)	
p[r+1][c+1] = v[r][c] + p[r][c+1] + p[r+1][c] - p[r][c];	
}	
T sum(int u, int l, int d, int r) {	
return p[d][r] - p[d][l] - p[u][r] + p[u][l];	
}	
};	

LineContainer.h

**Description:** Container where you can add lines of the form  $kx+m$ , and query maximum values at points  $x$ . Useful for dynamic programming (“convex hull trick”).  
**Time:**  $\mathcal{O}(\log N)$

struct Line {	a0102d, 30 lines
mutable ll k, m, p;	
bool operator<(const Line& o) const { return k < o.k; }	
bool operator<(ll x) const { return p < x; }	
};	
struct LineContainer : multiset<Line, less<>> {	
// (for doubles, use inf = 1/.0, div(a,b) = a/b)	
static constexpr ll inf = LLONG_MAX;	
ll div(ll a, ll b) { // floored division	
return a / b - ((a ^ b) < 0 && a % b); }	
bool isect(iterator x, iterator y) {	
if (y == end()) return x->p = inf, 0;	
if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;	
else x->p = div(y->m - x->m, x->k - y->k);	
return x->p >= y->p;	
}	
void add(ll k, ll m) {	
auto z = insert({k, m, 0}), y = z++, x = y;	
while (isect(y, z)) z = erase(z);	
if (x != begin() && isect(--x, y)) isect(x, y = erase(y));	
while ((y = x) != begin() && (--x)->p >= y->p)	
isect(x, erase(y));	
}	
ll query(ll x) {	
assert(!empty());	
auto l = *lower_bound(x);	

```
        return l.k * x + l.m;
    }
};
```

UnionFindRollback.h

**Description:** Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().  
**Usage:** int t = uf.time(); ...; uf.rollback(t);  
**Time:**  $\mathcal{O}(\log N)$

```
struct RollbackUF {
    vi e; vector<pii> st;
    RollbackUF(int n) : e(n, -1) {}
    int size(int x) { return -e[find(x)]; }
    int find(int x) { return e[x] < 0 ? x : find(e[x]); }
    int time() { return sz(st); }
    void rollback(int t) {
        for (int i = time(); i --> t;)
            e[st[i].first] = st[i].second;
        st.resize(t);
    }
    bool join(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (e[a] > e[b]) swap(a, b);
        st.push_back({a, e[a]});
        st.push_back({b, e[b]});
        e[a] += e[b]; e[b] = a;
        return true;
    }
};
```

Matrix.h

**Description:** Basic operations on square matrices.  
**Usage:** Matrix<int, 3> A;  
A.d = {{{{1,2,3}}, {{4,5,6}}, {{7,8,9}}}}};  
vector<int> vec = {1,2,3};  
vec = (A^N) \* vec;

```
template<class T, int N> struct Matrix {
    typedef Matrix M;
    array<array<T, N>, N> d{};
    M operator*(const M& m) const {
        M a;
        rep(i,0,N) rep(j,0,N)
            rep(k,0,N) a.d[i][j] += d[i][k]*m.d[k][j];
        return a;
    }
    vector<T> operator*(const vector<T>& vec) const {
        vector<T> ret(N);
        rep(i,0,N) rep(j,0,N) ret[i] += d[i][j] * vec[j];
        return ret;
    }
    M pow(ll p) const {
        assert(p >= 0);
        M a, b(*this);
        rep(i,0,N) a.d[i][i] = 1;
        while (p) {
            if (p&1) a = a*b;
            b = b*b;
            p >>= 1;
        }
        return a;
    }
};
```

Treap.h

**Description:** A short self-balancing tree. It acts as a sequential container with log-time splits/joins, and is easy to augment with additional data.

```
Time:  $\mathcal{O}(\log N)$ 
91bb92, 43 lines

struct Node {
    Node *l = nullptr, *r = nullptr;
    int val, y, c = 1;
    Node(int val) : val(val), y(rng()) {}
    void push();
    void pull();
};

int cnt(Node* n) { return n ? n->c : 0; }
void Node::push() {}
void Node::pull() { c = cnt(l) + cnt(r) + 1; }

pair<Node*, Node*> split(Node* n, int k) {
    if (!n) return {};
    n->push();
    if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
        auto pa = split(n->l, k);
        n->l = pa.second;
        n->pull();
        return {pa.first, n};
    } else {
        auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
        n->r = pa.first;
        n->pull();
        return {n, pa.second};
    }
}

Node* merge(Node* l, Node* r) {
    if (!l) return r;
    if (!r) return l;
    if (l->y > r->y) {
        l->push();
        l->r = merge(l->r, r);
        l->pull();
        return l;
    } else {
        r->push();
        r->l = merge(l, r->l);
        r->pull();
        return r;
    }
}
```

FenwickTree.h

**Description:** Computes partial sums  $a[0] + a[1] + \dots + a[\text{pos} - 1]$ , and updates single elements  $a[i]$ , taking the difference between the old and new value.  
**Time:** Both operations are  $\mathcal{O}(\log N)$ .

```
struct FT {
    vector<ll> s;
    FT(int n) : s(n) {}
    void update(int pos, ll dif) { // a[pos] += dif
        for (; pos < sz(s); pos |= pos + 1) s[pos] += dif;
    }
    ll query(int pos) { // sum of values in [0, pos)
        ll res = 0;
        for (; pos > 0; pos &= pos - 1) res += s[pos-1];
        return res;
    }
    int lower_bound(ll sum) { // min pos st sum of [0, pos] >= sum
        // Returns n if no sum is >= sum, or -1 if empty sum is.
        if (sum <= 0) return -1;
        int pos = 0;
        for (int pw = 1 << 25; pw; pw >>= 1) {
            if (pos + pw <= sz(s) && s[pos + pw-1] < sum)
                pos += pw, sum -= s[pos-1];
        }
    }
};
```

```
    }
    return pos;
};

FenwickTree2d.h
Description: Computes sums a[i,j] for all i<I, j<J, and increases single elements a[i,j]. Requires that the elements to be updated are known in advance (call fakeUpdate() before init()).
Time:  $\mathcal{O}(\log^2 N)$ . (Use persistent segment trees for  $\mathcal{O}(\log N)$ .)
"FenwickTree.h"
157f07, 22 lines

struct FT2 {
    vector<vi> ys; vector<FT> ft;
    FT2(int limx) : ys(limx) {}
    void fakeUpdate(int x, int y) {
        for (; x < sz(ys); x |= x + 1) ys[x].push_back(y);
    }
    void init() {
        for (vi& v : ys) sort(all(v)), ft.emplace_back(sz(v));
    }
    int ind(int x, int y) {
        return (int)(lower_bound(all(ys[x]), y) - ys[x].begin()); }
    void update(int x, int y, ll dif) {
        for (; x < sz(ys); x |= x + 1)
            ft[x].update(ind(x, y), dif);
    }
    ll query(int x, int y) {
        ll sum = 0;
        for (; x; x &= x - 1)
            sum += ft[x-1].query(ind(x-1, y));
        return sum;
    }
};
```

MoQueries.h

**Description:** Answer interval or tree path queries by finding an approximate TSP through the queries, and moving from one query to the next by adding/removing points at the ends. If values are on tree edges, change step to add/remove the edge (a,c) and remove the initial add call (but keep in).  
**Time:**  $\mathcal{O}(N\sqrt{Q})$

```
a12ef4, 49 lines

void add(int ind, int end) { ... } // add a[ind] (end = 0 or 1)
void del(int ind, int end) { ... } // remove a[ind]
int calc() { ... } // compute current answer

vi mo(vector<pii> Q) {
    int L = 0, R = 0, blk = 350; // ~N/sqrt(Q)
    vi s(sz(Q)), res = s;
    #define K(x) pii(x.first/blk, x.second ^ -(x.first/blk & 1))
    iota(all(s), 0);
    sort(all(s), [&](int s, int t){ return K(Q[s]) < K(Q[t]); });
    for (int qi : s) {
        pii q = Q[qi];
        while (L > q.first) add(--L, 0);
        while (R < q.second) add(R++, 1);
        while (L < q.first) del(L++, 0);
        while (R > q.second) del(--R, 1);
        res[qi] = calc();
    }
    return res;
}
```

```
vi moTree(vector<array<int, 2>> Q, vector<vi& ed, int root=0){
    int N = sz(ed), pos[2] = {}, blk = 350; // ~N/sqrt(Q)
    vi s(sz(Q)), res = s, I(N), L(N), R(N), in(N), par(N);
    add(0, 0), in[0] = 1;
    auto dfs = [&](int x, int p, int dep, auto& f) -> void {
        par[x] = p;
        L[x] = N;
```

```
    if (dep) I[x] = N++;
    for (int y : ed[x]) if (y != p) f(y, x, !dep, f);
    if (!dep) I[x] = N++;
    R[x] = N;
};

dfs(root, -1, 0, dfs);
#define K(x) pii(I[x[0]] / blk, I[x[1]] ^ -(I[x[0]] / blk & 1))
iota(all(s), 0);
sort(all(s), [&](int s, int t){ return K(Q[s]) < K(Q[t]); });
for (int qi : s) rep(end,0,2) {
    int &a = pos[end], b = Q[qi][end], i = 0;
#define step(c) { if (in[c]) { del(a, end); in[a] = 0; } \
                  else { add(c, end); in[c] = 1; } a = c; }
    while (!(L[b] <= L[a] && R[a] <= R[b]))
        I[i++] = b, b = par[b];
    while (a != b) step(par[a]);
    while (i--) step(I[i]);
    if (end) res[qi] = calc();
}
return res;
}
```

**FoldStack.h**  
**Description:** Compute right fold of op of elements in stack. Error on querying empty stack.  
**Usage:** int op(int a, int b) { return min(a, b); }  
FoldStack<int, op> st;  
**Time:**  $\mathcal{O}(1)$

```
template <class S, S (*op)(S, S)>
struct FoldStack {
    stack<pair<S, S>> st;
    void push(S x) {
        st.emplace(x, empty() ? x : op(x, st.top().second));
    }
    S pop() {
        auto res = st.top().first;
        return st.pop(), res;
    }
    S top() { return st.top().second; }
    bool empty() { return st.empty(); }
};
```

**AssocQueue.h**  
**Description:** Compute op of all values in queue for an associative binary operator. Error on querying empty queue.  
**Usage:** int op(int a, int b) { return min(a, b); }  
AssocQueue<int, op> st;  
**Time:**  $\mathcal{O}(1)$  amortized

```
"FoldStack.h"
template <class S, S (*op)(S, S)>
struct AssocQueue {
    static S op2(S a, S b) { return op(b, a); } // can remove for commutative functions
    FoldStack<S, op> s1; // swap op, op2 to flip order of
    FoldStack<S, op2> s2; // evaluation, in front() too

    void push(S x) { s1.push(x); }
    S pop() {
        if (s2.empty())
            while (!s1.empty()) s2.push(s1.pop());
        return s2.pop();
    }
    S front() {
        if (s1.empty()) return s2.top();
        if (s2.empty()) return s1.top();
        return op(s1.top(), s2.top()); // swap here too
    }
    bool empty() { return s1.empty() && s2.empty(); }
```

```
};

XorBasis.h
92207d, 36 lines

struct xor_basis {
    static constexpr int MAXN = 505;
    array<pair<bitset<MAXN>, bitset<MAXN>>, MAXN> basis;
    bool insert(bitset<MAXN> a, int ind){
        bitset<MAXN> cur; cur[ind] = 1;
        per(i, 0, MAXN)
            if (a[i]) {
                if (basis[i].F.any())
                    a ^= basis[i].F, cur ^= basis[i].S;
                else {
                    basis[i].F = a, basis[i].S = cur;
                    return true;
                }
            }
        return false;
    }
    // insert stores min range to achieve that val
    // if (lb[i].val[j] == 0) {
    //     lb[i].val[j] = tmp, lb[i].pos[j] = po;
    //     break;
    // } else {
    //     if (po > lb[i].pos[j])
    //         swap(tmp, lb[i].val[j]),
    //         swap(po, lb[i].pos[j]);
    //     tmp ^= lb[i].val[j];
    // }
    bool belongs(bitset<MAXN> a, bitset<MAXN> &swit){
        per(i, 0, MAXN)
            if (a[i]) {
                if (basis[i].S != 0)
                    a ^= basis[i].F, swit ^= basis[i].S;
                else return false;
            }
        return true;
    }
};
```

Combinatorial (3)

3.1 Permutations

3.1.1 Factorial

<i>n</i>	1	2	3	4	5	6	7	8	9	10
<i>n</i> !	1	2	6	24	120	720	5040	40320	362880	3628800
<i>n</i>	11	12	13	14	15	16	17			
<i>n</i> !	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14			
<i>n</i>	20	25	30	40	50	100	150	171		
<i>n</i> !	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX		

**factorial.h**  
**Description:** Computes  $i!$  and  $(i!)^{-1}$  for all  $0 \leq i < N$  modulo a prime.  
"../number-theory/ModPow.h" 109316, 7 lines

```
constexpr int N = 1e6, mod = 1e9 + 7;
int fac[N], ifac[N];

fac[0] = 1;
rep(i, 1, N) fac[i] = (1ll)i * fac[i - 1] % mod;
ifac[N - 1] = modpow(fac[N - 1], mod - 2);
per(i, 1, N) ifac[i - 1] = (1ll)i * ifac[i] % mod;
```

**IntPerm.h**  
**Description:** Permutation  $\mapsto$  integer conversion. (Not order preserving.)  
Integer  $\mapsto$  permutation can use a lookup table.  
**Time:**  $\mathcal{O}(n)$

```
int permToInt(vi& v) {
    int use = 0, i = 0, r = 0;
    for(int x:v) r = r * ++i + __builtin_popcount(use & -(1<<x)),
        use |= 1 << x; // (note: minus, not ~!)
    return r;
}
```

3.1.2 Cycles

Let  $g_S(n)$  be the number of  $n$ -permutations whose cycle lengths all belong to the set  $S$ . Then

$$\sum_{n=0}^{\infty} g_S(n) \frac{x^n}{n!} = \exp \left( \sum_{n \in S} \frac{x^n}{n} \right)$$

3.1.3 Derangements

Permutations of a set such that none of the elements appear in their original position.

$$D(n) = (n-1)(D(n-1) + D(n-2)) = nD(n-1) + (-1)^n = \left\lfloor \frac{n!}{e} \right\rfloor$$

3.1.4 Burnside’s lemma

Given a group  $G$  of symmetries and a set  $X$ , the number of elements of  $X$  up to symmetry equals

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

where  $X^g$  are the elements fixed by  $g$  ( $g.x = x$ ).

If  $f(n)$  counts “configurations” (of some sort) of length  $n$ , we can ignore rotational symmetry using  $G = \mathbb{Z}_n$  to get

$$g(n) = \frac{1}{n} \sum_{k=0}^{n-1} f(\gcd(n, k)) = \frac{1}{n} \sum_{k|n} f(k) \phi(n/k).$$

3.2 Partitions and subsets

3.2.1 Partition function

Number of ways of writing  $n$  as a sum of positive integers, disregarding the order of the summands.

$$p(0) = 1, \quad p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k - 1)/2)$$

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

<i>n</i>	0	1	2	3	4	5	6	7	8	9	20	50	100
<i>p</i> ( <i>n</i> )	1	1	2	3	5	7	11	15	22	30	627	~2e5	~2e8

3.2.2 Lucas’ Theorem

Let  $n, m$  be non-negative integers and  $p$  a prime. Write  $n = n_k p^k + \dots + n_1 p + n_0$  and  $m = m_k p^k + \dots + m_1 p + m_0$ . Then  $\binom{n}{m} \equiv \prod_{i=0}^k \binom{n_i}{m_i} \pmod{p}$ .

3.2.3 Binomials

binomial.h  
**Description:** Computes  $\binom{n}{r}$  modulo a prime

```
"factorial.h"
5e8c9a, 4 lines

int C(int n, int r) {
    if (r < 0 || r > n) return 0;
    return (1ll)fac[n] * ifac[r] % mod * ifac[n - r] % mod;
}
```

multinomial.h

**Description:** Computes  $\binom{k_1 + \dots + k_n}{k_1, k_2, \dots, k_n} = \frac{(\sum k_i)!}{k_1!k_2! \dots k_n!}$ .

```
a0a312, 6 lines

ll multinomial(vi& v) {
    ll c = 1, m = v.empty() ? 1 : v[0];
    rep(i, 1, sz(v)) rep(j, 0, v[i])
        c = c * ++m / (j+1);
    return c;
}
```

3.3 General purpose numbers

3.3.1 Bernoulli numbers

EGF of Bernoulli numbers is  $B(t) = \frac{t}{e^t - 1}$  (FFT-able).  
 $B[0, \dots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \dots]$

Sums of powers:

$$\sum_{i=1}^n n^m = \frac{1}{m+1} \sum_{k=0}^m \binom{m+1}{k} B_k \cdot (n+1)^{m+1-k}$$

Euler-Maclaurin formula for infinite sums:

$$\sum_{i=m}^\infty f(i) = \int_m^\infty f(x)dx - \sum_{k=1}^\infty \frac{B_k}{k!} f^{(k-1)}(m) \\ \approx \int_m^\infty f(x)dx + \frac{f(m)}{2} - \frac{f'(m)}{12} + \frac{f'''(m)}{720} + O(f^{(5)}(m))$$

3.3.2 Stirling numbers of the first kind

Number of permutations on  $n$  items with  $k$  cycles.

$$c(n, k) = c(n-1, k-1) + (n-1)c(n-1, k), \quad c(0, 0) = 1 \\ \sum_{k=0}^n c(n, k)x^k = x(x+1) \dots (x+n-1)$$

$c(8, k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$   
 $c(n, 2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \dots$

3.3.3 Eulerian numbers

Number of permutations  $\pi \in S_n$  in which exactly  $k$  elements are greater than the previous element.  $k$   $j$ :s s.t.  $\pi(j) > \pi(j+1)$ ,  $k+1$   $j$ :s s.t.  $\pi(j) \geq j$ ,  $k$   $j$ :s s.t.  $\pi(j) > j$ .

$$E(n, k) = (n-k)E(n-1, k-1) + (k+1)E(n-1, k)$$

$$E(n, 0) = E(n, n-1) = 1$$

$$E(n, k) = \sum_{j=0}^k (-1)^j \binom{n+1}{j} (k+1-j)^n$$

binomial multinomial PolyRoots

3.3.4 Stirling numbers of the second kind

Partitions of  $n$  distinct elements into exactly  $k$  groups.

$$S(n, k) = S(n-1, k-1) + kS(n-1, k)$$

$$S(n, 1) = S(n, n) = 1$$

$$S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$$

3.3.5 Bell numbers

Total number of partitions of  $n$  distinct elements.  $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$  For  $p$  prime,

$$B(p^m + n) \equiv mB(n) + B(n+1) \pmod{p}$$

3.3.6 Labeled unrooted trees

# on  $n$  vertices:  $n^{n-2}$   
# on  $k$  existing trees of size  $n_i$ :  $n_1 n_2 \dots n_k n^{k-2}$   
# with degrees  $d_i$ :  $(n-2)! / ((d_1-1)! \dots (d_n-1)!)$

3.3.7 Catalan numbers

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

$$C_0 = 1, \quad C_{n+1} = \frac{2(2n+1)}{n+2} C_n, \quad C_{n+1} = \sum C_i C_{n-i}$$

$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots$

- sub-diagonal monotone paths in an  $n \times n$  grid.
- strings with  $n$  pairs of parenthesis, correctly nested.
- binary trees with with  $n+1$  leaves (0 or 2 children).
- ordered trees with  $n+1$  vertices.
- ways a convex polygon with  $n+2$  sides can be cut into triangles by connecting vertices with straight lines.
- permutations of  $[n]$  with no 3-term increasing subseq.

Paths from from  $(0, 0)$  to  $(x, y)$  not crossing  $x = y$ ,  $\binom{x+y}{x} - \binom{x+y}{x+1}$ .

3.4 Power Series Operations

3.4.1 Lagrange Inversion

To solve for  $g(x)$  in  $f(g(x)) = x$ ,

$$k[x^k]h(g(x)) = [t^{-1}]h'(t)f(t)^{-k}.$$

In particular, for  $f(t) = t/\phi(t)$ ,  $h(x) = x$ ,

$$k[x^k]g(x) = [t^{k-1}]\phi(t)^k.$$

3.4.2 Combinatorial Species

$$\mathcal{A} = \text{Pointing}(\mathcal{B}) \implies A(z) = z \frac{d}{dz} B(z)$$

Operations for OGFs, be careful about  $\mathcal{B}_0 \neq \emptyset$ ,

$$\mathcal{A} = \text{Mset}(\mathcal{B}) \implies A(z) = \frac{\exp\left(\sum_{k \geq 1} B(z^k)/k\right)}{\prod_{k \geq 1} (1 - z^k)^{-B_k}}$$

$$\mathcal{A} = \text{Pset}(\mathcal{B}) \implies A(z) = \frac{\exp\left(\sum_{k \geq 1} (-1)^{k-1} B(z^k)/k\right)}{\prod_{k \geq 1} (1 + z^k)^{B_k}}$$

$$\mathcal{A} = \text{Cyc}(\mathcal{B}) \implies A(z) = \sum_{k \geq 1} \frac{\varphi(k)}{k} \log \frac{1}{1 - B(z^k)}.$$

Operations for EGFs, be careful about  $\mathcal{B}_0 \neq \emptyset$ ,

$$\mathcal{A} = \text{Seq}(\mathcal{B}) \implies A(z) = \frac{1}{1 - B(z)}$$

$$\mathcal{A} = \text{Set}(\mathcal{B}) \implies A(z) = \exp(B(z))$$

$$\mathcal{A} = \text{Cyc}(\mathcal{B}) \implies A(z) = \log \frac{1}{1 - B(z)}$$

Numerical (4)

4.1 Polynomials and recurrences

```
PolyRoots.h
Description: Finds the real roots to a polynomial.
Usage: polyRoots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0
Time:  $\mathcal{O}(n^2 \log(1/\epsilon))$ 
"Polynomial.h"
b00bfe, 23 lines

vector<double> polyRoots(Poly p, double xmin, double xmax) {
    if (sz(p.a) == 2) { return {-p.a[0]/p.a[1]}; }
    vector<double> ret;
    Poly der = p;
    der.diff();
    auto dr = polyRoots(der, xmin, xmax);
    dr.push_back(xmin-1);
    dr.push_back(xmax+1);
    sort(all(dr));
    rep(i, 0, sz(dr)-1) {
        double l = dr[i], h = dr[i+1];
        bool sign = p(l) > 0;
        if (sign ^ (p(h) > 0)) {
            rep(it, 0, 60) { // while (h - l > 1e-8)
                double m = (l + h) / 2, f = p(m);
                if ((f <= 0) ^ sign) l = m;
                else h = m;
            }
            ret.push_back((l + h) / 2);
        }
    }
    return ret;
}
```

## PolyInterpolate.h

**Description:** Given  $n$  points  $(x[i], y[i])$ , computes an  $n-1$ -degree polynomial  $p$  that passes through them:  $p(x) = a[0] * x^0 + \dots + a[n-1] * x^{n-1}$ . For numerical precision, pick  $x[k] = c * \cos(k / (n-1) * \pi)$ ,  $k = 0 \dots n-1$ .

**Time:**  $\mathcal{O}(n^2)$  a058de, 13 lines

```
using vd = vector<double>;
vd interpolate(vd x, vd y, int n) {
    vd res(n), temp(n);
    rep(k,0,n-1) rep(i,k+1,n)
        y[i] = (y[i] - y[k]) / (x[i] - x[k]);
    double last = 0; temp[0] = 1;
    rep(k,0,n) rep(i,0,n) {
        res[i] += y[k] * temp[i];
        swap(last, temp[i]);
        temp[i] -= last * x[k];
    }
    return res;
}
```

To interpolate in  $O(n \log^2 n)$ , let  $Q(x) = \prod_j (x - x_j)$ ,  $Q'(x_i)$  gives  $\prod_{j \neq i} (x_i - x_j)$ . Then divide and conquer to compute the polynomial.

## BerlekampMassey.h

**Description:** Recovers any  $n$ -order linear recurrence relation from the first  $2n$  terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size  $\leq n$ .

**Usage:** `berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}`

**Time:**  $\mathcal{O}(N^2)$

```

../number-theory/ModPow.h" 96548b, 20 lines
vector<ll> berlekampMassey(vector<ll> s) {
    int n = sz(s), L = 0, m = 0;
    vector<ll> C(n), B(n), T;
    C[0] = B[0] = 1;

    ll b = 1;
    rep(i,0,n) { ++m;
        ll d = s[i] % mod;
        rep(j,1,L+1) d = (d + C[j] * s[i - j]) % mod;
        if (!d) continue;
        T = C; ll coef = d * modpow(b, mod-2) % mod;
        rep(j,m,n) C[j] = (C[j] - coef * B[j - m]) % mod;
        if (2 * L > i) continue;
        L = i + 1 - L; B = T; b = d; m = 0;
    }

    C.resize(L + 1); C.erase(C.begin());
    for (ll& x : C) x = (mod - x) % mod;
    return C;
}

```

## LinearRecurrence.h

**Description:** Generates the  $k$ 'th term of an  $n$ -order linear recurrence  $S[i] = \sum_j S[i-j-1]tr[j]$ , given  $S[0 \dots n-1]$  and  $tr[0 \dots n-1]$ . Faster than matrix multiplication. Useful together with Berlekamp-Massey.

**Usage:** `linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number`

**Time:**  $\mathcal{O}(n^2 \log k)$ , can be improved to  $\mathcal{O}(n \log n \log k)$ , replace combine (which is a polynomial modulo operation) with FFT.

```
using poly = vector<ll>;
ll linearRec(poly S, poly tr, ll k) {
    int n = sz(tr);

    auto combine = [&](poly a, poly b) {
        poly res(n * 2 + 1);
        rep(i, 0, n+1) rep(j, 0, n+1)
```

```

    res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
    for (int i = 2 * n; i > n; --i) rep(j, 0, n)
        res[i - 1 - j] = (res[i - 1 - j] + res[i] * tr[j]) % mod;
    res.resize(n + 1);
    return res;
};

poly pol(n + 1), e(pol);
pol[0] = e[1] = 1;

for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
}

ll res = 0;
rep(i, 0, n) res = (res + pol[i + 1] * S[i]) % mod;
return res;
}

```

Newton-Raphson method,  $F$  is a polynomial operation.

$$F(Q) = 0 \implies Q_{k+1} = Q_k - F(Q_k)/F'(Q_k) \pmod{x^{2^{k+1}}}$$

Useful examples,

$$P^{-1} : Q_{k+1} = Q_k(2 - PQ_k) \pmod{x^{2^{k+1}}}$$

$$\exp(P) : Q_{k+1} = Q_k(1 + P - \ln Q_k) \pmod{x^{2^{k+1}}}$$

$$P^{1/m} : Q_{k+1} = ((m-1)Q_k + PQ_k^{-(m-1)})/m \pmod{x^{2^{k+1}}}$$

If  $P^R(x)$  is the reverse of  $P(x)$ , compute mod like this.

$$A = BD + R \implies D^R(x) = A^R(x)(B^R(x))^{-1} \pmod{x^{n-m+1}}$$

## 4.2 Optimization

## GoldenSectionSearch.h

**Description:** Finds the argument minimizing the function  $f$  in the interval  $[a, b]$  assuming  $f$  is unimodal on the interval, i.e. has only one local minimum and no local maximum. The maximum error in the result is  $eps$ . Works equally well for maximization with a small change in the code. See Ternary-Search.h in the Various chapter for a discrete version.

```

Usage: double func(double x) { return 4+x+.3*x*x; }
double xmin = gss(-1000,1000,func);
Time:  $\mathcal{O}(\log((b-a)/\epsilon))$ 

```

31d45b, 14 lines

```
double gss(double a, double b, double (*f)(double)) {
    double r = (sqrt(5)-1)/2, eps = 1e-7;
    double x1 = b - r*(b-a), x2 = a + r*(b-a);
    double f1 = f(x1), f2 = f(x2);
    while (b-a > eps)
        if (f1 < f2) { //change to > to find maximum
            b = x2; x2 = x1; f2 = f1;
            x1 = b - r*(b-a); f1 = f(x1);
        } else {
            a = x1; x1 = x2; f1 = f2;
            x2 = a + r*(b-a); f2 = f(x2);
        }
    return a;
}
```

## HillClimbing.h

```

Description: Poorman's optimization for unimodal functions.
using P = array<double, 2>;

template<class F> pair<double, P> hillClimb(P start, F f) {
    pair<double, P> cur(f(start), start);
    for (double jmp = 1e9; jmp > 1e-20; jmp /= 2) {
        rep(j,0,100) rep(dx,-1,2) rep(dy,-1,2) {
            P p = cur.second;
            p[0] += dx*jmp;
            p[1] += dy*jmp;
            cur = min(cur, make_pair(f(p), p));
        }
    }
    return cur;
}

```

## Integrate.h

**Description:** Simple integration of a function over an interval using Simpson's rule. The error should be proportional to  $h^4$ , although in practice you will want to verify that the result is stable to desired precision when epsilon changes.

```
template<class F>
double quad(double a, double b, F f, const int n = 1000) {
    double h = (b - a) / 2 / n, v = f(a) + f(b);
    rep(1, l, n*2)
        v += f(a + i*h) * (i&1 ? 4 : 2);
    return v * h / 3;
}
```

## IntegrateAdaptive.h

**Description:** Fast integration using an adaptive Simpson's rule.

**Usage:** double sphereVolume = quad(-1, 1, [])(double x) {  
 return quad(-1, 1, [&](double y) {  
 return quad(-1, 1, [&](double z) {  
 return x\*x + y\*y + z\*z < 1; } } } } };

f57493, 15 lines

```
using d = double;
#define S(a,b) (f(a) + 4*f((a+b) / 2) + f(b)) * (b-a) / 6
```

```
template <class F>
d rec(F& f, d a, d b, d eps, d S) {
    d c = (a + b) / 2;
    d S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
    if (abs(T - S) <= 15 * eps || b - a < 1e-10)
        return T + (T - S) / 15;
    return rec(f, a, c, eps / 2, S1) + rec(f, c, b, eps / 2, S2);
}

template<class F>
d quad(d a, d b, F f, d eps = 1e-8) {
    return rec(f, a, b, eps, S(a, b));
}
```

# Simplex.h

**Description:** Solves a general linear maximization problem: maximize  $c^T x$  subject to  $Ax \leq b$ ,  $x \geq 0$ . Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of  $c^T x$  otherwise. The input vector is set to an optimal  $x$  (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that  $x = 0$  is viable.

**Usage:** vvd A = {{1,-1}, {-1,1}}, c = {{-1,-1}, {-1,-2}};  
vvd b = {1,1,-4}, c = {-1,-1}, x;  
T val = LPSolver(A, b, c).solve(x);

**Time:**  $O(NM \times \#pivots)$ , where a pivot may be e.g. an edge relaxation.  
 $O(2^n)$  in the general case.

```
using T = double; // long double, Rational, double + mod<P>...
using vd = vector<T>;
using vvd = vector<vd>;
```

```
constexpr T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s])) s=j

struct LPSolver {
    int m, n;
    vi N, B;
    vvd D;

    LPSolver(const vvd& A, const vd& b, const vd& c) :
        m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)) {
        rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
        rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }
        rep(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m+1][n] = 1;
    }

    void pivot(int r, int s) {
        T *a = D[r].data(), inv = 1 / a[s];
        rep(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
            T *b = D[i].data(), inv2 = b[s] * inv;
            rep(j,0,n+2) b[j] -= a[j] * inv2;
            b[s] = a[s] * inv2;
        }
        rep(j,0,n+2) if (j != s) D[r][j] *= inv;
        rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
    }

    bool simplex(int phase) {
        int x = m + phase - 1;
        for (;;) {
            int s = -1;
            rep(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
            if (D[x][s] >= -eps) return true;
            int r = -1;
            rep(i,0,m) {
                if (D[i][s] <= eps) continue;
                if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                    < MP(D[r][n+1] / D[r][s], B[r])) r = i;
            }
            if (r == -1) return false;
            pivot(r, s);
        }
    }

    T solve(vd &x) {
        int r = 0;
        rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
        if (D[r][n+1] < -eps) {
            pivot(r, n);
            if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
            rep(i,0,m) if (B[i] == -1) {
                int s = 0;
                rep(j,1,n+1) ltj(D[i]);
                pivot(i, s);
            }
        }
        bool ok = simplex(1); x = vd(n);
        rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
        return ok ? D[m][n+1] : inf;
    }
};
```

4.3 Matrices

Determinant.h

**Description:** Calculates determinant of a matrix. Destroys the matrix.  
**Time:**  $\mathcal{O}(N^3)$

```
double det(vector<vector<double>>& a) {
    int n = sz(a); double res = 1;
    rep(i,0,n) {
        int b = i;
        rep(j,i+1,n) if (fabs(a[j][i]) > fabs(a[b][i])) b = j;
        if (i != b) swap(a[i], a[b]), res *= -1;
        res *= a[i][i];
        if (res == 0) return 0;
        rep(j,i+1,n) {
            double v = a[j][i] / a[i][i];
            if (v != 0) rep(k,i+1,n) a[j][k] -= v * a[i][k];
        }
    }
    return res;
}
```

IntDeterminant.h

**Description:** Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.  
**Time:**  $\mathcal{O}(N^3)$

```
constexpr ll mod = 12345;
ll det(vector<vector<ll>>& a) {
    int n = sz(a); ll ans = 1;
    rep(i,0,n) {
        rep(j,i+1,n) {
            while (a[j][i] != 0) { // gcd step
                ll t = a[i][i] / a[j][i];
                if (t) rep(k,i,n)
                    a[i][k] = (a[i][k] - a[j][k] * t) % mod;
                swap(a[i], a[j]);
                ans *= -1;
            }
            ans = ans * a[i][i] % mod;
            if (!ans) return 0;
        }
    }
    return (ans + mod) % mod;
}
```

SolveLinear.h

**Description:** Solves  $A * x = b$ . If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in  $A$  and  $b$  is lost.  
**Time:**  $\mathcal{O}(n^2m)$

```
using vd = vector<double>;
constexpr double eps = 1e-12;

int solveLinear(vector<vd>& A, vd& b, vd& x) {
    int n = sz(A), m = sz(x), rank = 0, br, bc;
    if (n) assert(sz(A[0]) == m);
    vi col(m); iota(all(col), 0);

    rep(i,0,n) {
        double v, bv = 0;
        rep(r,i,n) rep(c,i,m)
            if ((v = fabs(A[r][c])) > bv)
                br = r, bc = c, bv = v;
        if (bv <= eps) {
            rep(j,i,n) if (fabs(b[j]) > eps) return -1;
            break;
        }
        swap(A[i], A[br]);
        swap(b[i], b[br]);
```

```
        swap(col[i], col[bc]);
        rep(j,0,n) swap(A[j][i], A[j][bc]);
        bv = 1/A[i][i];
        rep(j,i+1,n) {
            double fac = A[j][i] * bv;
            b[j] -= fac * b[i];
            rep(k,i+1,m) A[j][k] -= fac*A[i][k];
        }
        rank++;
    }

    x.assign(m, 0);
    for (int i = rank; i--;) {
        b[i] /= A[i][i];
        x[col[i]] = b[i];
        rep(j,0,i) b[j] -= A[j][i] * b[i];
    }
    return rank; // (multiple solutions if rank < m)
}
```

SolveLinear2.h

**Description:** To get all uniquely determined values of  $x$  back from SolveLinear, make the following changes:

```
"SolveLinear.h"
rep(j,0,n) if (j != i) // instead of rep(j,i+1,n)
// ... then at the end:
x.assign(m, undefined);
rep(i,0,rank) {
    rep(j,rank,m) if (fabs(A[i][j]) > eps) goto fail;
    x[col[i]] = b[i] / A[i][i];
fail;; }
```

SolveLinearBinary.h

**Description:** Solves  $Ax = b$  over  $\mathbb{F}_2$ . If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys  $A$  and  $b$ .  
**Time:**  $\mathcal{O}(n^2m)$

```
using bs = bitset<1000>;

int solveLinear(vector<bs>& A, vi& b, bs& x, int m) {
    int n = sz(A), rank = 0, br;
    assert(m <= sz(x));
    vi col(m); iota(all(col), 0);
    rep(i,0,n) {
        for (br=i; br<n; ++br) if (A[br].any()) break;
        if (br == n) {
            rep(j,i,n) if(b[j]) return -1;
            break;
        }
        int bc = (int)A[br]._Find_next(i-1);
        swap(A[i], A[br]);
        swap(b[i], b[br]);
        swap(col[i], col[bc]);
        rep(j,0,n) if (A[j][i] != A[j][bc]) {
            A[j].flip(i); A[j].flip(bc);
        }
        rep(j,i+1,n) if (A[j][i]) {
            b[j] ^= b[i];
            A[j] ^= A[i];
        }
        rank++;
    }

    x = bs();
    for (int i = rank; i--;) {
        if (!b[i]) continue;
        x[col[i]] = 1;
        rep(j,0,i) b[j] ^= A[j][i];
    }
}
```



```
void fft_2d(poly_2d& a) { // returns transpose
    const int n = sz(a), Ln = 31 - __builtin_clz(n);
    const int m = sz(a[0]), Lm = 31 - __builtin_clz(m);
    assert(m == 1 << Ln);
    for (auto& v : a) assert(sz(v) == (1 << Lm)), fft(v);
```

```
poly_2d at(m, poly(n, 0)); a.resize(n, poly(m, 0));
rep(i, 0, m) rep(j, 0, n) at[i][j] = a[j][i];
for (auto& v : at) fft(v);
a = move(at);
}
```

```
poly_2d conv_2d(poly_2d a, poly_2d b) {
    if (a.empty() || b.empty()) return {};
    const int Ln=32-__builtin_clz(sz(a)+sz(b)-1), n=1<<Ln;
    const int Lm=32-__builtin_clz(sz(a[0])+sz(b[0])-1), m=1<<Lm;

    a.resize(n), b.resize(n); // make power of 2
    rep(i, 0, n) a[i].resize(m), b[i].resize(m);
    fft_2d(a), fft_2d(b); // FT
    rep(i, 0, m) rep(j, 0, n) a[i][j] *= b[i][j];
    fft_2d(a); // inverse FT
    for (auto& row : a) reverse(1 + all(row));
    reverse(1 + all(a));
    const double inv = 1.0 / (n * m);
    for (auto& row : a) for (auto& x : row) x *= inv;
    return a;
}
```

### FastFourierTransformMod.h

**Description:** Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as  $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$  (in practice  $10^{16}$  or higher). Inputs must be in  $[0, \text{mod})$ .

**Time:**  $\mathcal{O}(N \log N)$ , where  $N = |A| + |B|$  (twice as slow as NTT or FFT)

"FastFourierTransform.h"	6c05a9, 22 lines
<pre><b>using</b> vl = vector&lt;ll&gt;; <b>template</b>&lt;<b>int</b> M&gt; vl convMod(<b>const</b> vl &amp;a, <b>const</b> vl &amp;b) {     <b>if</b> (a.empty()    b.empty()) <b>return</b> {};     vl res(sz(a) + sz(b) - 1);     <b>int</b> B=32-__builtin_clz(sz(res)), n=1&lt;&lt;B, cut=<b>int</b>(sqrt(M));     vector&lt;C&gt; L(n), R(n), outs(n), outl(n);     rep(i,0,sz(a)) L[i] = C((<b>int</b>)a[i] / cut, (<b>int</b>)a[i] % cut);     rep(i,0,sz(b)) R[i] = C((<b>int</b>)b[i] / cut, (<b>int</b>)b[i] % cut);     fft(L), fft(R);     rep(i,0,n) {         <b>int</b> j = -i &amp; (n - 1);         outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);         outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / li;     }     fft(outl), fft(outs);     rep(i,0,sz(res)) {         ll av = ll(real(outl[i])+.5), cv = ll(imag(outs[i])+.5);         ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5);         res[i] = ((av % M * cut + bv) % M * cut + cv) % M;     }     <b>return</b> res; }</pre>	

### NumberTheoreticTransform.h

**Description:** ntt(a) computes  $\hat{f}(k) = \sum_x a[x]g^{xk}$  for all  $k$ , where  $g = \text{root}^{(\text{mod}-1)/N}$ .  $N$  must be a power of 2. Useful for convolution modulo specific nice primes of the form  $2^a b + 1$ , where the convolution result has size at most  $2^a$ . For arbitrary modulo, see FFTMod. conv(a, b) = c, where  $c[x] = \sum a[i]b[x - i]$ . For manual convolution: NTT the inputs, multiply pointwise, divide by n, reverse(start+1, end), NTT back. Inputs must be in  $[0, \text{mod})$ .

**Time:**  $\mathcal{O}(N \log N)$

"../number-theory/ModPow.h"	a26fdd, 35 lines
<pre>constexpr ll mod = (119 &lt;&lt; 23) + 1, root = 62; // = 998244353 // For p &lt; 2^30 there is also e.g. 5 &lt;&lt; 25, 7 &lt;&lt; 26, 479 &lt;&lt; 21 // and 483 &lt;&lt; 21 (same root). The last two are &gt; 10^9. <b>using</b> vl = vector&lt;ll&gt;; <b>void</b> ntt(vl &amp;a) {     <b>int</b> n = sz(a), L = 31 - __builtin_clz(n);</pre>	

```
static vl rt(2, 1);
for (static int k = 2, s = 2; k < n; k *= 2, s++) {
    rt.resize(n);
    ll z[] = {1, modpow(root, mod >> s)};
    rep(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1] % mod;
}
vi rev(n);
rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
        ll z = rt[j + k] * a[i + j + k] % mod, &ai = a[i + j];
        a[i + j + k] = ai - z + (z > ai ? mod : 0);
        ai += (ai + z >= mod ? z - mod : z);
    }
}
vl conv(const vl &a, const vl &b) {
    if (a.empty() || b.empty()) return {};
    int s = sz(a) + sz(b) - 1, B = 32 - __builtin_clz(s),
        n = 1 << B;
    int inv = modpow(n, mod - 2);
    vl L(a), R(b), out(n);
    L.resize(n), R.resize(n);
    ntt(L), ntt(R);
    rep(i,0,n)
        out[-i & (n - 1)] = (ll)L[i] * R[i] % mod * inv % mod;
    ntt(out);
    return {out.begin(), out.begin() + s};
}
```

### FastSubsetTransform.h

**Description:** Transform to a basis with fast convolutions of the form  $c[z] = \sum_{z=x \oplus y} a[x] \cdot b[y]$ , where  $\oplus$  is one of AND, OR, XOR. The size of  $a$  must be a power of two.

**Time:**  $\mathcal{O}(N \log N)$

	741958, 16 lines
<pre><b>void</b> FST(vi&amp; a, <b>bool</b> inv) {     <b>for</b> (<b>int</b> n = sz(a), step = 1; step &lt; n; step *= 2) {         <b>for</b> (<b>int</b> i = 0; i &lt; n; i += 2 * step) rep(j,i,i+step) {             <b>auto</b> &amp;u = a[j], &amp;v = a[j + step]; tie(u, v) =                 inv ? pii(v - u, u) : pii(v, u + v); // AND                 inv ? pii(v, u - v) : pii(u + v, u); // OR                 // XOR         }     }     <b>if</b> (inv) <b>for</b> (<b>auto</b>&amp; x : a) x /= sz(a); // XOR only } vi conv(vi a, vi b) {     FST(a, 0); FST(b, 0);     rep(i,0,sz(a)) a[i] *= b[i];     FST(a, 1); <b>return</b> a; }</pre>	

### SubsetConvolution.h

**Description:** Compute  $c(S) = \sum_{T \subseteq S} a(T)b(S \setminus T)$ .

**Time:**  $\mathcal{O}(N \log^2 N)$

```
vi conv(vi a, vi b) {
    int n = 31 - __builtin_clz(sz(a));
    vector f(n + 1, vi(1 << n, 0)); auto g = f, h = f;
    rep(S, 0, 1 << n) {
        auto i = __builtin_popcount(S);
        f[i][S] = a[S]; g[i][S] = b[S];
    }
    rep(i, 0, n + 1) rep(j, 0, n) rep(S, 0, 1 << n)
        if (S >> j & 1) { // subset sum
            f[i][S] += f[i][S ^ (1 << j)];
            g[i][S] += g[i][S ^ (1 << j)];
        }
```

```
    }
    rep(i, 0, n + 1) rep(j, 0, i + 1) rep(S, 0, 1 << n)
        h[i][S] += f[j][S] * g[i - j][S];
    rep(i, 0, n + 1) rep(j, 0, n) rep(S, 0, 1 << n)
        if (S >> j & 1) h[i][S] -= h[i][S ^ (1 << j)];
    vi c(1 << n);
    rep(S, 0, 1 << n) c[S] = h[__builtin_popcount(S)][S];
    return c;
}
```

## Number theory (5)

### 5.1 Modular arithmetic

#### ModLog.h

**Description:** Returns the smallest  $x > 0$  s.t.  $a^x = b \pmod m$ , or  $-1$  if no such  $x$  exists. modLog(a,1,m) can be used to calculate the order of  $a$ . Doesn't require gcd( $a, m$ ) = 1.

**Time:**  $\mathcal{O}(\sqrt{m})$

	d7e01b, 11 lines
<pre>ll modLog(ll a, ll b, ll m) {     ll n = (ll) sqrt(m) + 1, e = 1, f = 1, j = 1;     unordered_map&lt;ll, ll&gt; A;     <b>while</b> (j &lt;= n &amp;&amp; (e = f = e * a % m) != b % m)         A[e * b % m] = j++;     <b>if</b> (e == b % m) <b>return</b> j;     <b>if</b> (gcd(m, e) == gcd(m, b))         rep(i,2,n+2) <b>if</b> (A.count(e = e * f % m))             <b>return</b> n * i - A[e];     <b>return</b> -1; }</pre>	

#### DivSum.h

**Description:** Sums of div'ed arithmetic progressions.

divsum(to, c, k, m) =  $\sum_{i=0}^{\text{to}-1} \lfloor \frac{ki+c}{m} \rfloor$ .

**Time:**  $\log(m)$ , with a large constant.

	d5f4b5, 9 lines
<pre>ull sumsq(ull to) { <b>return</b> to / 2 * ((to-1)   1); }</pre>	
<pre>ull divsum(ull to, ull c, ull k, ull m) {     ull res = k / m * sumsq(to) + c / m * to;     k %= m; c %= m;     <b>if</b> (!k) <b>return</b> res;     ull to2 = (to * k + c) / m;     <b>return</b> res + (to - 1) * to2 - divsum(to2, m-1 - c, m, k); }</pre>	

#### ModMulLL.h

**Description:** Calculate  $a \cdot b \pmod c$  (or  $a^b \pmod c$ ) for  $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$ .

**Time:**  $\mathcal{O}(1)$  for modmul,  $\mathcal{O}(\log b)$  for modpow

	4c7c4a, 11 lines
<pre><b>using</b> ull = unsigned long long; ull modmul(ull a, ull b, ull M) {     ll ret = a * b - M * ull(1.L / M * a * b);     <b>return</b> ret + M * (ret &lt; 0) - M * (ret &gt;= (ll)M); } ull modpow(ull b, ull e, ull mod) {     ull ans = 1;     <b>for</b> (; e; b = modmul(b, b, mod), e /= 2)         <b>if</b> (e &amp; 1) ans = modmul(ans, b, mod);     <b>return</b> ans; }</pre>	

#### ModSqrt.h

**Description:** Tonelli-Shanks algorithm for modular square roots. Finds  $x$  s.t.  $x^2 = a \pmod p$  ( $-x$  gives the other solution).

**Time:**  $\mathcal{O}(\log^2 p)$  worst case,  $\mathcal{O}(\log p)$  for most  $p$

"ModPow.h"	7483eb, 24 lines
------------	------------------

$(p, k) = (962592769, 21), (754974721, 24), (167772161, 25),$   
 $(469762049, 26)$  are such that  $2^k \mid p - 1$ .

### 5.7 Estimates

$\sum_{d|n} d = O(n \log \log n)$ .

The number of divisors of  $n$  is at most around 100 for  $n < 5e4$ , 500 for  $n < 1e7$ , 2000 for  $n < 1e10$ , 200 000 for  $n < 1e19$ .

### 5.8 Mobius Function

Mobius Inversion:

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu(d)g(n/d)$$

Other useful formulas/forms:

$$\sum_{d|n} \mu(d) = [n = 1] \text{ (very useful)}$$

$$g(n) = \sum_{n|d} f(d) \Leftrightarrow f(n) = \sum_{n|d} \mu(d/n)g(d)$$

$$g(n) = \sum_{1 \leq m \leq n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum_{1 \leq m \leq n} \mu(m)g(\lfloor \frac{n}{m} \rfloor)$$

## Graph (6)

### 6.1 Fundamentals

BellmanFord.h

**Description:** Calculates shortest paths from  $s$  in a graph that might have negative edge weights. Unreachable nodes get dist = inf; nodes reachable through negative-weight cycles get dist = -inf. Assumes  $V^2 \max |w_i| < \sim 2^{63}$ .  
**Time:**  $O(VE)$

```
constexpr ll inf = LLONG_MAX;
struct Ed { int a, b, w, s() { return a < b ? a : -a; }};
struct Node { ll dist = inf; int prev = -1; };

void bellmanFord(vector<Node>& nodes, vector<Ed>& eds, int s) {
    nodes[s].dist = 0;
    sort(all(eds), [](Ed a, Ed b) { return a.s() < b.s(); });

    int lim = sz(nodes) / 2 + 2; // /3+100 with shuffled vertices
    rep(i,0,lim) for (Ed ed : eds) {
        Node cur = nodes[ed.a], &dest = nodes[ed.b];
        if (abs(cur.dist) == inf) continue;
        ll d = cur.dist + ed.w;
        if (d < dest.dist) {
            dest.prev = ed.a;
            dest.dist = (i < lim-1 ? d : -inf);
        }
    }
    rep(i,0,lim) for (Ed e : eds) {
        if (nodes[e.a].dist == -inf)
            nodes[e.b].dist = -inf;
    }
}
```

FloydWarshall.h

**Description:** Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix  $m$ , where  $m[i][j] = \text{inf}$  if  $i$  and  $j$  are not adjacent. As output,  $m[i][j]$  is set to the shortest distance between  $i$  and  $j$ , inf if no path, or -inf if the path goes through a negative-weight cycle.  
**Time:**  $O(N^3)$

```
constexpr ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll>>& m) {
```

```
int n = sz(m);
rep(i,0,n) m[i][i] = min(m[i][i], 0LL);
rep(k,0,n) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) {
        auto newDist = max(m[i][k] + m[k][j], -inf);
        m[i][j] = min(m[i][j], newDist);
    }
rep(k,0,n) if (m[k][k] < 0) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

### 6.2 Network flow

PushRelabel.h

**Description:** Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.  
**Time:**  $O(V^2\sqrt{E})$

```
struct PushRelabel {
    struct Edge {
        int dest, back;
        ll f, c;
    };
    vector<vector<Edge>> g;
    vector<ll> ec;
    vector<Edge*> cur;
    vector<vi> hs; vi H;
    PushRelabel(int n) : g(n), ec(n), cur(n), hs(2*n), H(n) {}

    void addEdge(int s, int t, ll cap, ll rcap=0) {
        if (s == t) return;
        g[s].push_back({t, sz(g[t]), 0, cap});
        g[t].push_back({s, sz(g[s])-1, 0, rcap});
    }

    void addFlow(Edge& e, ll f) {
        Edge &back = g[e.dest][e.back];
        if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
        e.f += f; e.c -= f; ec[e.dest] += f;
        back.f -= f; back.c += f; ec[back.dest] -= f;
    }

    ll calc(int s, int t) {
        int v = sz(g); H[s] = v; ec[t] = 1;
        vi co(2*v); co[0] = v-1;
        rep(i,0,v) cur[i] = g[i].data();
        for (Edge& e : g[s]) addFlow(e, e.c);

        for (int hi = 0;;) {
            while (hs[hi].empty()) if (!hi--) return -ec[s];
            int u = hs[hi].back(); hs[hi].pop_back();
            while (ec[u] > 0) // discharge u
                if (cur[u] == g[u].data() + sz(g[u])) {
                    H[u] = 1e9;
                    for (Edge& e : g[u]) if (e.c && H[u] > H[e.dest]+1)
                        H[u] = H[e.dest]+1, cur[u] = &e;
                    if (++co[H[u]],!--co[hi] && hi < v)
                        rep(i,0,v) if (hi < H[i] && H[i] < v)
                            --co[H[i]], H[i] = v + 1;
                    hi = H[u];
                } else if (cur[u]->c && H[u] == H[cur[u]->dest]+1)
                    addFlow(*cur[u], min(ec[u], cur[u]->c));
                else ++cur[u];
        }
    }

    bool leftOfMinCut(int a) { return H[a] >= sz(g); }
};
```

MinCostMaxFlow.h

**Description:** Min-cost max-flow. If costs can be negative, call setpi before maxflow, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.  
**Time:**  $O(FE \log(V))$  where F is max flow.  $O(VE)$  for setpi.

```
constexpr ll INF = numeric_limits<ll>::max() / 4;

struct MCMF {
    struct edge {
        int from, to, rev;
        ll cap, cost, flow;
    };
    int N;
    vector<vector<edge>> ed;
    vi seen;
    vector<ll> dist, pi;
    vector<edge*> par;

    MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N), par(N) {}

    void addEdge(int from, int to, ll cap, ll cost) {
        if (from == to) return;
        ed[from].push_back(edge{ from,to,sz(ed[to]),cap,cost,0 });
        ed[to].push_back(edge{ to,from,sz(ed[from])-1,0,-cost,0 });
    }

    void path(int s) {
        fill(all(seen), 0);
        fill(all(dist), INF);
        dist[s] = 0; ll di;

        __gnu_pbds::priority_queue<pair<ll, int>> q;
        vector<decltype(q)::point_iterator> its(N);
        q.push({ 0, s });

        while (!q.empty()) {
            s = q.top().second; q.pop();
            seen[s] = 1; di = dist[s] + pi[s];
            for (edge& e : ed[s]) if (!seen[e.to]) {
                ll val = di - pi[e.to] + e.cost;
                if (e.cap - e.flow > 0 && val < dist[e.to]) {
                    dist[e.to] = val;
                    par[e.to] = &e;
                    if (its[e.to] == q.end())
                        its[e.to] = q.push({ -dist[e.to], e.to });
                    else
                        q.modify(its[e.to], { -dist[e.to], e.to });
                }
            }
        }
        rep(i,0,N) pi[i] = min(pi[i] + dist[i], INF);
    }

    pair<ll, ll> maxflow(int s, int t) {
        ll totflow = 0, totcost = 0;
        while (path(s), seen[t]) {
            ll fl = INF;
            for (edge* x = par[t]; x; x = par[x->from])
                fl = min(fl, x->cap - x->flow);

            totflow += fl;
            for (edge* x = par[t]; x; x = par[x->from]) {
                x->flow += fl;
                ed[x->to][x->rev].flow -= fl;
            }
        }
    }
};
```

```
    rep(i,0,N) for(edge& e : ed[i]) totcost += e.cost * e.flow;
    return {totflow, totcost/2};
}

// If some costs can be negative, call this before maxflow:
void setpi(int s) { // (otherwise, leave this out)
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--){
        rep(i,0,N) if (pi[i] != INF)
            for (edge& e : ed[i]) if (e.cap)
                if ((v = pi[i] + e.cost) < pi[e.to])
                    pi[e.to] = v, ch = 1;
        assert(it >= 0); // negative cost cycle
    }
};
```

EdmondsKarp.h

**Description:** Flow algorithm with guaranteed complexity  $O(VE^2)$ . To get edge flow values, compare capacities before and after, and take the positive values only.

482fe0, 36 lines

```
template<class T> T edmondsKarp(vector<unordered_map<int, T>&&
    graph, int source, int sink) {
    assert(source != sink);
    T flow = 0;
    vi par(sz(graph)), q = par;

    for (;;) {
        fill(all(par), -1);
        par[source] = 0;
        int ptr = 1;
        q[0] = source;

        rep(i,0,ptr) {
            int x = q[i];
            for (auto e : graph[x]) {
                if (par[e.first] == -1 && e.second > 0) {
                    par[e.first] = x;
                    q[ptr++] = e.first;
                    if (e.first == sink) goto out;
                }
            }
        }
        return flow;
    out:
        T inc = numeric_limits<T>::max();
        for (int y = sink; y != source; y = par[y])
            inc = min(inc, graph[par[y]][y]);

        flow += inc;
        for (int y = sink; y != source; y = par[y]) {
            int p = par[y];
            if ((graph[p][y] -= inc) <= 0) graph[p].erase(y);
            graph[y][p] += inc;
        }
    }
};
```

Dinic.h

**Description:** Flow algorithm with complexity  $O(VE \log U)$  where  $U = \max[\text{cap}]$ .  $O(\min(E^{1/2}, V^{2/3})E)$  if  $U = 1$ ;  $O(\sqrt{VE})$  for bipartite matching and unit networks ( $U = 1$  and  $\min(\text{indeg}(v), \text{outdeg}(v)) \leq 1$  for each node  $v$ ).

d7f0f1, 42 lines

```
struct Dinic {
    struct Edge {
        int to, rev;
        ll c, oc;
```

```
        ll flow() { return max(oc - c, 0LL); } // if you need flows
    };
    vi lvl, ptr, q;
    vector<vector<Edge>> adj;
    Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
    void addEdge(int a, int b, ll c, ll rcap = 0) {
        adj[a].push_back({b, sz(adj[b]), c, c});
        adj[b].push_back({a, sz(adj[a]) - 1, rcap, rcap});
    }
    ll dfs(int v, int t, ll f) {
        if (v == t || !f) return f;
        for (int& i = ptr[v]; i < sz(adj[v]); i++) {
            Edge& e = adj[v][i];
            if (lvl[e.to] == lvl[v] + 1)
                if (ll p = dfs(e.to, t, min(f, e.c))) {
                    e.c -= p, adj[e.to][e.rev].c += p;
                    return p;
                }
        }
        return 0;
    }
    ll calc(int s, int t) {
        ll flow = 0; q[0] = s;
        rep(L,0,31) do { // 'int L=30' maybe faster for random data
            lvl = ptr = vi(sz(q));
            int qi = 0, qe = lvl[s] = 1;
            while (qi < qe && !lvl[t]) {
                int v = q[qi++];
                for (Edge e : adj[v])
                    if (!lvl[e.to] && e.c >> (30 - L))
                        q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
            }
            while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
        } while (lvl[t]);
        return flow;
    }
    bool leftOfMinCut(int a) { return lvl[a] != 0; }
};
```

MinCut.h

**Description:** After running max-flow, the left side of a min-cut from  $s$  to  $t$  is given by all vertices reachable from  $s$ , only traversing edges with positive residual capacity.

GlobalMinCut.h

**Description:** Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.

**Time:**  $O(V^3)$

8b0e19, 21 lines

```
pair<int, vi> globalMinCut(vector<vi> mat) {
    pair<int, vi> best = {INT_MAX, {}};
    int n = sz(mat);
    vector<vi> co(n);
    rep(i,0,n) co[i] = {i};
    rep(ph,1,n) {
        vi w = mat[0];
        size_t s = 0, t = 0;
        rep(it,0,n-ph) { // O(V^2) -> O(E log V) with prio. queue
            w[t] = INT_MIN;
            s = t, t = max_element(all(w)) - w.begin();
            rep(i,0,n) w[i] += mat[t][i];
        }
        best = min(best, {w[t] - mat[t][t], co[t]});
        co[s].insert(co[s].end(), all(co[t]));
        rep(i,0,n) mat[s][i] += mat[t][i];
        rep(i,0,n) mat[i][s] = mat[s][i];
        mat[0][t] = INT_MIN;
    }
};
```

```
    return best;
};
```

GomoryHu.h

**Description:** Given a list of edges representing an undirected flow graph, returns edges of the Gomory-Hu tree. The max flow between any pair of vertices is given by minimum edge weight along the Gomory-Hu tree path.

**Time:**  $O(V)$  Flow Computations

"PushRelabel.h" 9efedc, 13 lines

```
using Edge = array<ll, 3>;
vector<Edge> gomoryHu(int N, vector<Edge> ed) {
    vector<Edge> tree;
    vi par(N);
    rep(i,1,N) {
        PushRelabel D(N); // Dinic also works
        for (Edge t : ed) D.addEdge(t[0], t[1], t[2], t[2]);
        tree.push_back({i, par[i], D.calc(i, par[i])});
        rep(j,i+1,N)
            if (par[j] == par[i] && D.leftOfMinCut(j)) par[j] = i;
    }
    return tree;
};
```

6.3 Matching

hopcroftKarp.h

**Description:** Fast bipartite matching algorithm. Graph  $g$  should be a list of neighbors of the left partition, and  $btoa$  should be a vector full of -1's of the same size as the right partition. Returns the size of the matching.  $btoa[i]$  will be the match for vertex  $i$  on the right side, or  $-1$  if it's not matched.

**Usage:**  $vi\ btoa(m, -1);$  hopcroftKarp( $g, btoa$ );

**Time:**  $O(\sqrt{VE})$

f612e4, 42 lines

```
bool dfs(int a, int L, vector<vi>& g, vi& btoa, vi& A, vi& B) {
    if (A[a] != L) return 0;
    A[a] = -1;
    for (int b : g[a]) if (B[b] == L + 1) {
        B[b] = 0;
        if (btoa[b] == -1 || dfs(btoa[b], L + 1, g, btoa, A, B))
            return btoa[b] = a, 1;
    }
    return 0;
};
```

```
int hopcroftKarp(vector<vi>& g, vi& btoa) {
    int res = 0;
    vi A(g.size()), B(btoa.size()), cur, next;
    for (;;) {
        fill(all(A), 0);
        fill(all(B), 0);
        cur.clear();
        for (int a : btoa) if(a != -1) A[a] = -1;
        rep(a,0,sz(g)) if(A[a] == 0) cur.push_back(a);
        for (int lay = 1;; lay++) {
            bool islast = 0;
            next.clear();
            for (int a : cur) for (int b : g[a]) {
                if (btoa[b] == -1) {
                    B[b] = lay;
                    islast = 1;
                }
            }
            else if (btoa[b] != a && !B[b]) {
                B[b] = lay;
                next.push_back(btoa[b]);
            }
        }
        if (islast) break;
        if (next.empty()) return res;
        for (int a : next) A[a] = lay;
```

```
        cur.swap(next);
    }
    rep(a,0,sz(g))
        res += dfs(a, 0, g, btoa, A, B);
}
}
```

DFSMatching.h

**Description:** Simple bipartite matching algorithm. Graph  $g$  should be a list of neighbors of the left partition, and  $btoa$  should be a vector full of -1's of the same size as the right partition. Returns the size of the matching.  $btoa[i]$  will be the match for vertex  $i$  on the right side, or  $-1$  if it's not matched.  
**Usage:** `vi btoa(m, -1); dfsMatching(g, btoa);`  
**Time:**  $\mathcal{O}(VE)$

522b98, 22 lines

```
bool find(int j, vector<vi>& g, vi& btoa, vi& vis) {
    if (btoa[j] == -1) return 1;
    vis[j] = 1; int di = btoa[j];
    for (int e : g[di])
        if (!vis[e] && find(e, g, btoa, vis)) {
            btoa[e] = di;
            return 1;
        }
    return 0;
}
int dfsMatching(vector<vi>& g, vi& btoa) {
    vi vis;
    rep(i,0,sz(g)) {
        vis.assign(sz(btoa), 0);
        for (int j : g[i])
            if (find(j, g, btoa, vis)) {
                btoa[j] = i;
                break;
            }
    }
    return sz(btoa) - (int)count(all(btoa), -1);
}
```

MinimumVertexCover.h

**Description:** Finds a minimum vertex cover in a bipartite graph. The size is the same as the size of a maximum matching, and the complement is a maximum independent set.

"DFSMatching.h" da4196, 20 lines

```
vi cover(vector<vi>& g, int n, int m) {
    vi match(m, -1);
    int res = dfsMatching(g, match);
    vector<bool> lfound(n, true), seen(m);
    for (int it : match) if (it != -1) lfound[it] = false;
    vi g, cover;
    rep(i,0,n) if (lfound[i]) q.push_back(i);
    while (!q.empty()) {
        int i = q.back(); q.pop_back();
        lfound[i] = 1;
        for (int e : g[i]) if (!seen[e] && match[e] != -1) {
            seen[e] = true;
            q.push_back(match[e]);
        }
    }
    rep(i,0,n) if (!lfound[i]) cover.push_back(i);
    rep(i,0,m) if (seen[i]) cover.push_back(n+i);
    assert(sz(cover) == res);
    return cover;
}
```

WeightedMatching.h

**Description:** Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes `cost[N][M]`, where `cost[i][j] = cost` for  $L[i]$  to be matched with  $R[j]$  and returns  $(\min \text{cost}, \text{match})$ , where  $L[i]$  is matched with  $R[\text{match}[i]]$ . Negate costs for max cost. Requires  $N \leq M$ .  
**Time:**  $\mathcal{O}(N^2M)$

1e0fe9, 31 lines

```
pair<int, vi> hungarian(const vector<vi> &a) {
    if (a.empty()) return {0, {}};
    int n = sz(a) + 1, m = sz(a[0]) + 1;
    vi u(n), v(m), p(m), ans(n - 1);
    rep(i,1,n) {
        p[0] = i;
        int j0 = 0; // add "dummy" worker 0
        vi dist(m, INT_MAX), pre(m, -1);
        vector<bool> done(m + 1);
        do { // dijkstra
            done[j0] = true;
            int i0 = p[j0], j1, delta = INT_MAX;
            rep(j,1,m) if (!done[j]) {
                auto cur = a[i0 - 1][j - 1] - u[i0] - v[j];
                if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
                if (dist[j] < delta) delta = dist[j], j1 = j;
            }
            rep(j,0,m) {
                if (done[j]) u[p[j]] += delta, v[j] -= delta;
                else dist[j] -= delta;
            }
            j0 = j1;
        } while (p[j0]);
        while (j0) { // update alternating path
            int j1 = pre[j0];
            p[j0] = p[j1], j0 = j1;
        }
    }
    rep(j,1,m) if (p[j]) ans[p[j] - 1] = j - 1;
    return {-v[0], ans}; // min cost
}
```

GeneralMatching.h

**Description:** Matching for general graphs. Fails with probability  $N/mod$ .  
**Time:**  $\mathcal{O}(N^3)$

"../numerical/MatrixInverse-mod.h" 4b1448, 40 lines

```
vector<pii> generalMatching(int N, vector<pii>& ed) {
    vector<vector<ll>> mat(N, vector<ll>(N)), A;
    for (pii pa : ed) {
        int a = pa.first, b = pa.second, r = rng() % mod;
        mat[a][b] = r, mat[b][a] = (mod - r) % mod;
    }

    int r = matInv(A = mat), M = 2*N - r, fi, fj;
    assert(r % 2 == 0);

    if (M != N) do {
        mat.resize(M, vector<ll>(M));
        rep(i,0,N) {
            mat[i].resize(M);
            rep(j,N,M) {
                int r = rng() % mod;
                mat[i][j] = r, mat[j][i] = (mod - r) % mod;
            }
        }
    } while (matInv(A = mat) != M);

    vi has(M, 1); vector<pii> ret;
    rep(it,0,M/2) {
        rep(i,0,M) if (has[i])
            rep(j,i+1,M) if (A[i][j] && mat[i][j]) {
                fi = i; fj = j; goto done;
            }
    }
```

```
    } assert(0); done;
    if (fj < N) ret.emplace_back(fi, fj);
    has[fi] = has[fj] = 0;
    rep(sw,0,2) {
        ll a = modpow(A[fi][fj], mod-2);
        rep(i,0,M) if (has[i] && A[i][fj]) {
            ll b = A[i][fj] * a % mod;
            rep(j,0,M) A[i][j] = (A[i][j] - A[fi][j] * b) % mod;
        }
        swap(fi,fj);
    }
}
return ret;
}
```

6.4 DFS algorithms

SCC.h

**Description:** Finds strongly connected components in a directed graph. If vertices  $u, v$  belong to the same component, we can reach  $u$  from  $v$  and vice versa.  
**Usage:** `scc(graph, [&](vi& v) { ... })` visits all components in reverse topological order. `comp[i]` holds the component index of a node (a component only has edges to components with lower index). `ncomps` will contain the number of components.  
**Time:**  $\mathcal{O}(E + V)$

76b5c9, 24 lines

```
vi val, comp, z, cont;
int Time, ncomps;
template<class G, class F> int dfs(int j, G& g, F& f) {
    int low = val[j] = ++Time, x; z.push_back(j);
    for (auto e : g[j]) if (comp[e] < 0)
        low = min(low, val[e] ?: dfs(e,g,f));

    if (low == val[j]) {
        do {
            x = z.back(); z.pop_back();
            comp[x] = ncomps;
            cont.push_back(x);
        } while (x != j);
        f(cont); cont.clear();
        ncomps++;
    }
    return val[j] = low;
}
template<class G, class F> void scc(G& g, F f) {
    int n = sz(g);
    val.assign(n, 0); comp.assign(n, -1);
    Time = ncomps = 0;
    rep(i,0,n) if (comp[i] < 0) dfs(i, g, f);
}
```

BiconnectedComponents.h

**Description:** Finds all biconnected components in an undirected graph, and runs a callback for the edges in each. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle.  
**Usage:** `int eid = 0; ed.resize(N);` for each edge (a,b) { `ed[a].emplace_back(b, eid);` `ed[b].emplace_back(a, eid++);` } `bicomps[&](const vi& edgelist) {...};`  
**Time:**  $\mathcal{O}(E + V)$

782746, 32 lines

```
vi tin, st;
vector<vector<pii>> ed;
int Time;
template<class F>
int dfs(int at, int par, F& f) {
```

```
int me = tin[at] = ++Time, top = me;
for (auto [y, e] : ed[at]) if (e != par) {
    if (tin[y]) {
        top = min(top, tin[y]);
        if (tin[y] < me)
            st.push_back(e);
    } else {
        int si = sz(st);
        int up = dfs(y, e, f);
        top = min(top, up);
        if (up == me) {
            st.push_back(e);
            f(vi(st.begin() + si, st.end()));
            st.resize(si);
        }
        else if (up < me) st.push_back(e);
        else { /* e is a bridge */ }
    }
}
return top;
}

template<class F>
void bicomps(F f) {
    tin.assign(sz(ed), 0);
    rep(i,0,sz(ed)) if (!tin[i]) dfs(i, -1, f);
}
```

ArticulationPoints.h

**Description:** In the biconnected components code, at is an articulation point only if

- it is the root and has > 1 children in the tree (not in original graph),
- it is not the root and there's a child (tree) edge such that up ≥ me.

"BiconnectedComponents.h"

2sat.h

**Description:** Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type  $(a||b)&&(!a||c)&&(d||!b)&&...$  becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions (~x).  
**Usage:** TwoSat ts(number of boolean variables);  
ts.either(0, ~3); // Var 0 is true or var 3 is false  
ts.setValue(2); // Var 2 is true  
ts.atMostOne({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true  
ts.solve(); // Returns true iff it is solvable  
ts.values[0..N-1] holds the assigned values to the vars  
**Time:**  $\mathcal{O}(N + E)$ , where  $N$  is the number of boolean variables, and  $E$  is the number of clauses.

5f9706, 56 lines

```
struct TwoSat {
    int N;
    vector<vi> gr;
    vi values; // 0 = false, 1 = true

    TwoSat(int n = 0) : N(n), gr(2*n) {}

    int addVar() { // (optional)
        gr.emplace_back();
        gr.emplace_back();
        return N++;
    }

    void either(int f, int j) {
        f = max(2*f, -1-2*f);
        j = max(2*j, -1-2*j);
        gr[f].push_back(j^1);
        gr[j].push_back(f^1);
    }
}
```

```
}
void setValue(int x) { either(x, x); }

void atMostOne(const vi& li) { // (optional)
    if (sz(li) <= 1) return;
    int cur = ~li[0];
    rep(i,2,sz(li)) {
        int next = addVar();
        either(cur, ~li[i]);
        either(cur, next);
        either(~li[i], next);
        cur = ~next;
    }
    either(cur, ~li[1]);
}

vi val, comp, z; int time = 0;
int dfs(int i) {
    int low = val[i] = ++time, x; z.push_back(i);
    for(int e : gr[i]) if (!comp[e])
        low = min(low, val[e] ?: dfs(e));
    if (low == val[i]) do {
        x = z.back(); z.pop_back();
        comp[x] = low;
        if (values[x>>1] == -1)
            values[x>>1] = x&1;
    } while (x != i);
    return val[i] = low;
}

bool solve() {
    values.assign(N, -1);
    val.assign(2*N, 0); comp = val;
    rep(i,0,2*N) if (!comp[i]) dfs(i);
    rep(i,0,N) if (comp[2*i] == comp[2*i+1]) return 0;
    return 1;
}
};
```

EulerWalk.h

**Description:** Eulerian undirected/directed path/cycle algorithm. Input should be a vector of (dest, global edge index), where for undirected graphs, forward/backward edges have the same index. Returns a list of nodes in the Eulerian path/cycle with src at both start and end, or empty list if no cycle/path exists. To get edge indices back, add .second to s and ret.  
**Time:**  $\mathcal{O}(V + E)$

780b64, 15 lines

```
vi eulerWalk(vector<vector<pii>>& gr, int nedges, int src=0) {
    int n = sz(gr);
    vi D(n), its(n), eu(nedges), ret, s = {src};
    D[src]++; // to allow Euler paths, not just cycles
    while (!s.empty()) {
        int x = s.back(), y, e, &it = its[x], end = sz(gr[x]);
        if (it == end){ ret.push_back(x); s.pop_back(); continue; }
        tie(y, e) = gr[x][it++];
        if (!eu[e]) {
            D[x]--, D[y]++;
            eu[e] = 1; s.push_back(y);
        }
    }
    for (int x : D) if (x < 0 || sz(ret) != nedges+1) return {};
    return {ret.rbegin(), ret.rend()};
}
```

6.5 Coloring

EdgeColoring.h

**Description:** Given a simple, undirected graph with max degree  $D$ , computes a  $(D + 1)$ -coloring of the edges such that no neighboring edges share a color. ( $D$ -coloring is NP-hard, but can be done for bipartite graphs by repeated matchings of max-degree nodes.)  
**Time:**  $\mathcal{O}(NM)$

e210e2, 31 lines

```
vi edgeColoring(int N, vector<pii> eds) {
    vi cc(N + 1), ret(sz(eds)), fan(N), free(N), loc;
    for (pii e : eds) ++cc[e.first], ++cc[e.second];
    int u, v, ncols = *max_element(all(cc)) + 1;
    vector<vi> adj(N, vi(ncols, -1));
    for (pii e : eds) {
        tie(u, v) = e;
        fan[0] = v;
        loc.assign(ncols, 0);
        int at = u, end = u, d, c = free[u], ind = 0, i = 0;
        while (d = free[v], !loc[d] && (v = adj[u][d]) != -1)
            loc[d] = ++ind, cc[ind] = d, fan[ind] = v;
        cc[loc[d]] = c;
        for (int cd = d; at != -1; cd ^= c ^ d, at = adj[at][cd])
            swap(adj[at][cd], adj[end = at][cd ^ c ^ d]);
        while (adj[fan[i]][d] != -1) {
            int left = fan[i], right = fan[++i], e = cc[i];
            adj[u][e] = left;
            adj[left][e] = u;
            adj[right][e] = -1;
            free[right] = e;
        }
        adj[u][d] = fan[i];
        adj[fan[i]][d] = u;
        for (int y : {fan[0], u, end})
            for (int& z = free[y] = 0; adj[y][z] != -1; z++);
    }
    rep(i,0,sz(eds))
        for (tie(u, v) = eds[i]; adj[u][ret[i]] != v;) ++ret[i];
    return ret;
}
```

6.6 Heuristics

MaximalCliques.h

**Description:** Runs a callback for all maximal cliques in a graph (given as a symmetric bitset matrix; self-edges not allowed). Callback is given a bitset representing the maximal clique.  
**Time:**  $\mathcal{O}\left(3^{n/3}\right)$ , much faster for sparse graphs

d03aee, 12 lines

```
using B = bitset<128>;
template<class F>
void cliques(vector<B>& eds, F f, B P = ~B(), B X={}, B R={}) {
    if (!P.any()) { if (!X.any()) f(R); return; }
    auto q = (P | X)._Find_first();
    auto cands = P & ~eds[q];
    rep(i,0,sz(eds)) if (cands[i]) {
        R[i] = 1;
        cliques(eds, f, P & eds[i], X & eds[i], R);
        R[i] = P[i] = 0; X[i] = 1;
    }
}
```

MaximumClique.h

**Description:** Quickly finds a maximum clique of a graph (given as symmetric bitset matrix; self-edges not allowed). Can be used to find a maximum independent set by finding a clique of the complement graph.  
**Time:** Runs in about 1s for n=155 and worst case random graphs (p=.90). Runs faster for sparse graphs.

9cf916, 49 lines

```
using vb = vector<bitset<200>>;
struct Maxclique {
    double limit=0.025, pk=0;
```

```
struct Vertex { int i, d=0; };
using vv = vector<Vertex>;
vb e;
vv V;
vector<vi> C;
vi qmax, q, S, old;
void init(vv& r) {
    for (auto& v : r) v.d = 0;
    for (auto& v : r) for (auto j : r) v.d += e[v.i][j.i];
    sort(all(r), [](auto a, auto b) { return a.d > b.d; });
    int mxD = r[0].d;
    rep(i,0,sz(r)) r[i].d = min(i, mxD) + 1;
}
void expand(vv& R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (sz(R)) {
        if (sz(q) + R.back().d <= sz(qmax)) return;
        q.push_back(R.back().i);
        vv T;
        for(auto v:R) if (e[R.back().i][v.i]) T.push_back({v.i});
        if (sz(T)) {
            if (S[lev]++ / ++pk < limit) init(T);
            int j = 0, mxk = 1, mnk = max(sz(qmax) - sz(q) + 1, 1);
            C[1].clear(), C[2].clear();
            for (auto v : T) {
                int k = 1;
                auto f = [&](int i) { return e[v.i][i]; };
                while (any_of(all(C[k]), f)) k++;
                if (k > mxk) mxk = k, C[mxk + 1].clear();
                if (k < mnk) T[j++].i = v.i;
                C[k].push_back(v.i);
            }
            if (j > 0) T[j - 1].d = 0;
            rep(k,mnk,mxk + 1) for (int i : C[k])
                T[j].i = i, T[j++].d = k;
            expand(T, lev + 1);
        } else if (sz(q) > sz(qmax)) qmax = q;
        q.pop_back(), R.pop_back();
    }
}
vi maxClique() { init(V), expand(V); return qmax; }
Maxclique(vb conn) : e(conn), C(sz(e)+1), S(sz(C)), old(S) {
    rep(i,0,sz(e)) V.push_back({i});
}
};
```

MaximumIndependentSet.h

**Description:** To obtain a maximum independent set of a graph, find a max clique of the complement. If the graph is bipartite, see MinimumVertex-Cover.

6.7 Trees

BinaryLifting.h

**Description:** Calculate power of two jumps in a tree, to support fast upward jumps and LCAs. Assumes the root node points to itself.  
**Time:** construction  $\mathcal{O}(N \log N)$ , queries  $\mathcal{O}(\log N)$

```
vector<vi> treeJump(vi& P){
    int on = 1, d = 1;
    while(on < sz(P)) on *= 2, d++;
    vector<vi> jmp(d, P);
    rep(i,1,d) rep(j,0,sz(P))
        jmp[i][j] = jmp[i-1][jmp[i-1][j]];
    return jmp;
}

int jmp(vector<vi>& tbl, int nod, int steps){
```

```
    rep(i,0,sz(tbl))
        if(steps&(1<<i)) nod = tbl[i][nod];
    return nod;
}

int lca(vector<vi>& tbl, vi& depth, int a, int b) {
    if (depth[a] < depth[b]) swap(a, b);
    a = jmp(tbl, a, depth[a] - depth[b]);
    if (a == b) return a;
    for (int i = sz(tbl); i--;) {
        int c = tbl[i][a], d = tbl[i][b];
        if (c != d) a = c, b = d;
    }
    return tbl[0][a];
}
```

LCA.h

**Description:** Data structure for computing lowest common ancestors in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected.

**Time:**  $\mathcal{O}(N \log N + Q)$

```
"../data-structures/RMQ.h" e5b4c6, 21 lines

struct LCA {
    int T = 0;
    vi tin, path, ret;
    RMQ<int> rmq;

    LCA(vector<vi>& C) : tin(sz(C)), rmq((dfs(C,0,-1), ret)) {}
    void dfs(vector<vi>& C, int v, int par) {
        tin[v] = T++;
        for (int y : C[v]) if (y != par) {
            path.push_back(v), ret.push_back(tin[v]);
            dfs(C, y, v);
        }
    }

    int lca(int a, int b) {
        if (a == b) return a;
        tie(a, b) = minmax(tin[a], tin[b]);
        return path[rmq.query(a, b)];
    }

    //dist(a,b){return depth[a] + depth[b] - 2*depth[lca(a,b)];}
};
```

CompressTree.h

**Description:** Given a rooted tree and a subset S of nodes, compute the minimal subtree that contains all the nodes by adding all (at most  $|S| - 1$ ) pairwise LCA's and compressing edges. Returns a list of (par, orig\_index) representing a tree rooted at 0. The root points to itself.

**Time:**  $\mathcal{O}(|S| \log |S|)$

```
"LCA.h" 63e21d, 21 lines

using vpi = vector<pair<int, int>>;
vpi compressTree(LCA& lca, const vi& subset) {
    static vi rev; rev.resize(sz(lca.time));
    vi li = subset, &T = lca.time;
    auto cmp = [&](int a, int b) { return T[a] < T[b]; };
    sort(all(li), cmp);
    int m = sz(li)-1;
    rep(i,0,m) {
        int a = li[i], b = li[i+1];
        li.push_back(lca.lca(a, b));
    }
    sort(all(li), cmp);
    li.erase(unique(all(li)), li.end());
    rep(i,0,sz(li)) rev[li[i]] = i;
    vpi ret = {pii(0, li[0])};
    rep(i,0,sz(li)-1) {
        int a = li[i], b = li[i+1];
```

```
        ret.emplace_back(rev[lca.lca(a, b)], b);
    }
    return ret;
}

HLD.h
Description: Decomposes a tree into vertex disjoint heavy paths and light edges such that the path from any leaf to the root contains at most  $\log(n)$  light edges. Takes as input the full adjacency list. VALS_EDGES being true means that values are stored in the edges, as opposed to the nodes. Root must be 0.
83e69a, 33 lines

template <bool VALS_EDGES> struct HLD {
    int N, T = 0;
    vector<vi> g;
    vi par, sub, dep, nxt, tin;
    HLD(vector<vi> g_)
        : N(sz(g_)), g(g_), par(N, -1), sub(N, 1), dep(N),
          nxt(N), tin(N){ dfsSz(0); dfsHld(0); }
    void dfsSz(int u) {
        for (int& v : g[u]) {
            g[v].erase(find(all(g[v]), u));
            par[v] = u, dep[v] = dep[u] + 1;
            dfsSz(v);
            sub[u] += sub[v];
            if (sub[v] > sub[g[u][0]]) swap(v, g[u][0]);
        }
    }
    void dfsHld(int u) {
        tin[u] = T++;
        for (int v : g[u]) {
            nxt[v] = (v == g[u][0] ? nxt[u] : v);
            dfsHld(v);
        }
    }
    template <class F> void process_path(int u, int v, F op) {
        for (; nxt[u] != nxt[v]; v = par[nxt[v]]) {
            if (dep[nxt[u]] > dep[nxt[v]]) swap(u, v);
            op(tin[nxt[v]], tin[v] + 1);
        }
        if (dep[u] > dep[v]) swap(u, v);
        op(tin[u] + VALS_EDGES, tin[v] + 1);
    }
    // process_subtree(u): [tin[u] + VALS_EDGES, tin[u] + sub[v])
};

LinkCutTree.h
Description: Represents a forest of unrooted trees. You can add and remove edges (as long as the result is still a forest), and check whether two nodes are in the same tree.
Time: All operations take amortized  $\mathcal{O}(\log N)$ .
0fb462, 90 lines

struct Node { // Splay tree. Root's pp contains tree's parent.
    Node *p = 0, *pp = 0, *c[2];
    bool flip = 0;
    Node() { c[0] = c[1] = 0; fix(); }
    void fix() {
        if (c[0]) c[0]->p = this;
        if (c[1]) c[1]->p = this;
        // (+ update sum of subtree elements etc. if wanted)
    }
    void pushFlip() {
        if (!flip) return;
        flip = 0; swap(c[0], c[1]);
        if (c[0]) c[0]->flip ^= 1;
        if (c[1]) c[1]->flip ^= 1;
    }
    int up() { return p ? p->c[1] == this : -1; }
    void rot(int i, int b) {
```



```

    int h = i ^ b;
    Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ? y : x;
    if ((y->p = p)) p->c[up()] = y;
    c[i] = z->c[i ^ 1];
    if (b < 2) {
        x->c[h] = y->c[h ^ 1];
        y->c[h ^ 1] = x;
    }
    z->c[i ^ 1] = this;
    fix(); x->fix(); y->fix();
    if (p) p->fix();
    swap(pp, y->pp);
}

void splay() {
    for (pushFlip(); p; ) {
        if (p->p) p->p->pushFlip();
        p->pushFlip(); pushFlip();
        int c1 = up(), c2 = p->up();
        if (c2 == -1) p->rot(c1, 2);
        else p->p->rot(c2, c1 != c2);
    }
}

Node* first() {
    pushFlip();
    return c[0] ? c[0]->first() : (splay(), this);
}

};

struct LinkCut {
    vector<Node> node;
    LinkCut(int N) : node(N) {}

    void link(int u, int v) { // add an edge (u, v)
        assert(!connected(u, v));
        makeRoot(&node[u]);
        node[u].pp = &node[v];
    }

    void cut(int u, int v) { // remove an edge (u, v)
        Node *x = &node[u], *top = &node[v];
        makeRoot(top); x->splay();
        assert(top == (x->pp ? x->c[0]));
        if (x->pp) x->pp = 0;
        else {
            x->c[0] = top->p = 0;
            x->fix();
        }
    }

    bool connected(int u, int v) { // are u, v in the same tree?
        Node* nu = access(&node[u])->first();
        return nu == access(&node[v])->first();
    }

    void makeRoot(Node* u) {
        access(u);
        u->splay();
        if (u->c[0]) {
            u->c[0]->p = 0;
            u->c[0]->flip ^= 1;
            u->c[0]->pp = u;
            u->c[0] = 0;
            u->fix();
        }
    }

    Node* access(Node* u) {
        u->splay();
        while (Node* pp = u->pp) {
            pp->splay(); u->pp = 0;
            if (pp->c[1]) {
                pp->c[1]->p = 0; pp->c[1]->pp = pp; }
            pp->c[1] = u; pp->fix(); u = pp;
        }
    }
};

```

```

    }
    return u;
}

};

BicsiLinkCutTree.h
Description: 1-indexed LinkCutTree with path, subtree aggregates, and LCA queries.
13ac91, 100 lines

struct SplayTree {
    struct Node {
        int ch[2] = {0, 0}, p = 0;
        ll self = 0, path = 0; // Path aggregates
        ll sub = 0, vir = 0; // Subtree aggregates
        bool flip = 0; // Lazy tags
    };
    vector<Node> T;
    SplayTree(int n) : T(n + 1) {}

    void push(int x) {
        if (!x || !T[x].flip) return;
        int l = T[x].ch[0], r = T[x].ch[1];

        T[l].flip ^= 1, T[r].flip ^= 1;
        swap(T[x].ch[0], T[x].ch[1]);
        T[x].flip = 0;
    }

    void pull(int x) {
        int l = T[x].ch[0], r = T[x].ch[1];
        push(l); push(r);
        T[x].path = T[l].path + T[x].self + T[r].path;
        T[x].sub = T[x].vir + T[l].sub + T[r].sub + T[x].self;
    }

    void set(int x, int d, int y) {
        T[x].ch[d] = y; T[y].p = x; pull(x);
    }

    void splay(int x) {
        auto dir = [&](int x) {
            int p = T[x].p;
            if (!p) return -1;
            return T[p].ch[0] == x ? 0 : T[p].ch[1] == x ? 1 : -1;
        };
        auto rotate = [&](int x) {
            int y = T[x].p, z = T[y].p, dx = dir(x), dy = dir(y);
            set(y, dx, T[x].ch[!dx]);
            set(x, !dx, y);
            if (~dy) set(z, dy, x);
            T[x].p = z;
        };
        for (push(x); ~dir(x);) {
            int y = T[x].p, z = T[y].p;
            push(z); push(y); push(x);
            int dx = dir(x), dy = dir(y);
            if (~dy) rotate(dx != dy ? x : y);
            rotate(x);
        }
    }

    struct LinkCut : SplayTree {
        LinkCut(int n) : SplayTree(n) {}

        int access(int x) {
            int u = x, v = 0;
            for (; u; v = u, u = T[u].p) {
                splay(u);
            }
        }
    };
};

```

```

    int& ov = T[u].ch[1];
    T[u].vir += T[ov].sub;
    T[u].vir -= T[v].sub;
    ov = v; pull(u);
}

return splay(x), v;
}

void reroot(int x) { access(x); T[x].flip ^= 1; push(x); }
void link(int u, int v) {
    reroot(u); access(v);
    T[v].vir += T[u].sub;
    T[u].p = v;
    pull(v);
}

void cut(int u, int v) {
    reroot(u); access(v);
    T[v].ch[0] = T[u].p = 0;
    pull(v);
}

// Rooted tree LCA. Returns 0 if u and v arent connected.
int LCA(int u, int v) {
    if (u == v) return u;
    access(u); int ret = access(v);
    return T[u].p ? ret : 0;
}

// Query subtree of u with v as root.
ll subtree(int u, int v) {
    reroot(v); access(u);
    return T[u].vir + T[u].self;
}

// Query path [u..v]
ll path(int u, int v) {
    reroot(u); access(v);
    return T[v].path;
}

// Update vertex u with value v
void update(int u, ll v) {
    access(u); T[u].self = v; pull(u);
}

};

```

## RerootDP.h

**Description:** root\_dp[u] gives value at  $u$  when  $u$  is root, edge\_dp[u][e] gives value for edge  $e$  of  $u$  when  $u$  is root.

74dab7, 55 lines

```

namespace reroot {
    const auto exclusive = [] (const auto& a, const auto& base,
        const auto& merge_into, int vertex) {
        using Aggregate = decay_t<decltype(base)>;
        int n = sz(a); vector<Aggregate> b(n, base);
        for (int bit = (int)lg(n); bit >= 0; --bit) {
            per(i, 0, n) b[i] = b[i >> 1];
            int sz = n - (n & !bit);
            rep(i, 0, sz) {
                int index = (i >> bit) ^ 1;
                b[index] = merge_into(b[index], a[i], vertex, i);
            }
        }
        return b;
    };

    // MergeInto: Aggregate * Value * Vertex(int) * EdgeIndex(int)
    //      -> Aggregate
    // Base: Vertex(int) -> Aggregate
    // FinalizeMerge: Aggregate * Vertex(int) * EdgeIndex(int) ->
    //      Value
    const auto rerooter = [] (const auto& g, const auto& base, const
        auto& merge_into, const auto& finalize_merge) {
        using Aggregate = decay_t<decltype(base(0))>;
    };
};

```

```

using Value = decay_t<decltype(finalize_merge(base(0), 0, 0))>;
int n = sz(g); vector<Value> root_dp(n), dp(n);
vector<vector<Value>> edge_dp(n), redge_dp(n);
vector<int> bfs, par(n);
bfs.reserve(n); bfs.push_back(0);
rep(i, 0, n) {
    int u = bfs[i];
    for (auto v : g[u]) {
        if (par[u] == v) continue;
        par[v] = u;
        bfs.push_back(v);
    }
}
per(i, 0, n) {
    int u = bfs[i], p_e = -1;
    Aggregate aggregate = base(u);
    rep(e, 0, sz(g[u])) {
        int v = g[u][e];
        if (par[u] == v) p_e = e;
        else aggregate = merge_into(aggregate, dp[v], u, e);
    }
    dp[u] = finalize_merge(aggregate, u, p_e);
}
for (auto u : bfs) {
    dp[par[u]] = dp[u];
    edge_dp[u].reserve(g[u].size());
    for (auto v : g[u]) edge_dp[u].push_back(dp[v]);
    auto dp_ex = exclusive(edge_dp[u], base(u), merge_into, u);
    redge_dp[u].reserve(g[u].size());
    rep(i, 0, sz(dp_ex)) redge_dp[u].push_back(finalize_merge(
        dp_ex[i], u, i));
    root_dp[u] = finalize_merge(n > 1 ? merge_into(dp_ex[0],
        edge_dp[u][0], u, 0) : base(u), u, -1);
    rep(i, 0, sz(g[u])) dp[g[u][i]] = redge_dp[u][i];
}
return make_tuple(move(root_dp), move(edge_dp), move(redge_dp));
};
}

```

## DirectedMST.h

**Description:** Finds a minimum spanning tree/arborescence of a directed graph, given a root node. If no MST exists, returns -1.

**Time:**  $\mathcal{O}(E \log V)$

../data-structures/UnionFindRollback.h 39e620, 60 lines

```

struct Edge { int a, b; ll w; };
struct Node {
    Edge key;
    Node *l, *r;
    ll delta;
    void prop() {
        key.w += delta;
        if (l) l->delta += delta;
        if (r) r->delta += delta;
        delta = 0;
    }
    Edge top() { prop(); return key; }
};
Node *merge(Node *a, Node *b) {
    if (!a || !b) return a ? b;
    a->prop(), b->prop();
    if (a->key.w > b->key.w) swap(a, b);
    swap(a->l, (a->r = merge(b, a->r)));
    return a;
}
void pop(Node*& a) { a->prop(); a = merge(a->l, a->r); }

pair<ll, vi> dmst(int n, int r, vector<Edge>& g) {

```

```

RollbackUF uf(n);
vector<Node*> heap(n);
for (Edge e : g) heap[e.b] = merge(heap[e.b], new Node{e});
ll res = 0;
vi seen(n, -1), path(n), par(n);
seen[r] = r;
vector<Edge> Q(n), in(n, {-1,-1}), comp;
deque<tuple<int, int, vector<Edge>>> cyps;
rep(s,0,n) {
    int u = s, qi = 0, w;
    while (seen[u] < 0) {
        if (!heap[u]) return {-1,{};};
        Edge e = heap[u]->top();
        heap[u]->delta -= e.w, pop(heap[u]);
        Q[qi] = e, path[qi++] = u, seen[u] = s;
        res += e.w, u = uf.find(e.a);
        if (seen[u] == s) {
            Node* cyc = 0;
            int end = qi, time = uf.time();
            do cyc = merge(cyc, heap[w = path[--qi]]);
            while (uf.join(u, w));
            u = uf.find(u), heap[u] = cyc, seen[u] = -1;
            cyps.push_front({u, time, {Q[qi], &Q[end]}});
        }
    }
    rep(i,0,qi) in[uf.find(Q[i].b)] = Q[i];
}

for (auto& [u,t,comp] : cyps) { // restore sol (optional)
    uf.rollback(t);
    Edge inEdge = in[u];
    for (auto& e : comp) in[uf.find(e.b)] = e;
    in[uf.find(inEdge.b)] = inEdge;
}
rep(i,0,n) par[i] = in[i].a;
return {res, par};
}

```

## DominatorTree.h

**Description:** Returns the immediate dominator of each node reachable from root.

**Time:**  $\mathcal{O}((n+m)\alpha(n))$

29be8c, 33 lines

```

vi find_dominators(const vector<vector<int>> &g, int root) {
    int n = sz(g); vi pos(n, -1), ord, par(n, -1);
    auto dfs = [&](auto &self, int u) -> void {
        pos[u] = sz(ord); ord.push_back(u);
        for (auto v : g[u]) if (pos[v]==-1) par[v]=u, self(self,v);
    };
    dfs(dfs, root);
    vi p(n), best(n), sdom = pos;
    iota(all(p), 0); iota(all(best), 0);
    auto fb = [&](auto &self, int x) -> int {
        if (p[x] != x) {
            int u = self(self, p[x]);
            if (sdom[u] < sdom[best[x]]) best[x] = u;
            p[x] = p[p[x]];
        }
        if (sdom[best[p[x]]] < sdom[best[x]]) best[x] = best[p[x]];
        return best[x];
    };
    vector<vector<int>> gr(n), bucket(n);
    vi idom(n, -1), link(n, -1);
    rep(i, 0, n) for (auto v:g[u]) gr[v].push_back(u);
    per(it, 0, sz(ord)) {
        int w = ord[it];
        for (auto u : gr[w])
            if (pos[u] != -1) uin(sdom[w], sdom[fb(fb, u)]);
        idom[w] = ord[sdom[w]];
    }
}

```

```

for (auto u : bucket[w]) link[u] = fb(fb, u);
for (auto u : g[w]) if (par[u] == w) p[u] = w;
bucket[ord[sdom[w]]].push_back(w);
}
rep(it, 1, sz(ord)) idom[ord[it]] = idom[link[ord[it]]];
return idom;
}

```

## 6.8 Math

### 6.8.1 Number of Spanning Trees

Create an  $N \times N$  matrix  $\text{mat}$ , and for each edge  $a \rightarrow b \in G$ , do  $\text{mat}[a][b]--$ ,  $\text{mat}[b][b]++$  (and  $\text{mat}[b][a]--$ ,  $\text{mat}[a][a]++$  if  $G$  is undirected). Remove the  $i$ th row and column and take the determinant; this yields the number of directed spanning trees rooted at  $i$  (if  $G$  is undirected, remove any row/column).

### 6.8.2 Graph Realization Problem

A simple graph with node degrees  $d_1 \geq \dots \geq d_n$  exists iff  $d_1 + \dots + d_n$  is even and for every  $k = 1 \dots n$ ,

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

## Geometry (7)

### 7.1 Geometric primitives

Point.h

**Description:** Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

47ec0a, 28 lines

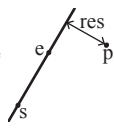
```

template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<class T>
struct Point {
    typedef Point P;
    T x, y;
    explicit Point(T x=0, T y=0) : x(x), y(y) {}
    bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }
    bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
    P operator+(P p) const { return P(x+p.x, y+p.y); }
    P operator-(P p) const { return P(x-p.x, y-p.y); }
    P operator*(T d) const { return P(x*d, y*d); }
    P operator/(T d) const { return P(x/d, y/d); }
    T dot(P p) const { return x*p.x + y*p.y; }
    T cross(P p) const { return x*p.y - y*p.x; }
    T cross(P a, P b) const { return (a-*this).cross(b-*this); }
    T dist2() const { return x*x + y*y; }
    double dist() const { return sqrt((double)dist2()); }
    // angle to x-axis in interval [-pi, pi]
    double angle() const { return atan2(y, x); }
    P unit() const { return *this/dist(); } // makes dist()==1
    P perp() const { return P(-y, x); } // rotates +90 degrees
    P normal() const { return perp().unit(); }
    // returns point rotated 'a' radians ccw around the origin
    P rotate(double a) const {
        return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
    friend ostream& operator<<(ostream& os, P p) {
        return os << "(" << p.x << "," << p.y << ")"; }
};

```

### lineDistance.h

**Description:**  
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long. Using Point3D will always give a non-negative distance. For Point3D, call .dist on the result of the cross product.

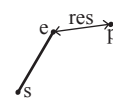


```
template<class P>
double lineDist(const P& a, const P& b, const P& p) {
    return (double) (b-a).cross(p-a)/(b-a).dist();
}
```

### SegmentDistance.h

**Description:**  
Returns the shortest distance between point p and the line segment from point s to e.

**Usage:** Point<double> a, b(2,2), p(1,1);  
bool onSegment = segDist(a,b,p) < 1e-10;



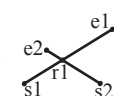
```
"Point.h"
5c88f4, 6 lines

typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
    if (s==e) return (p-s).dist();
    auto d = (e-s).dist2(), t = min(d,max(.0, (p-s).dot(e-s)));
    return ((p-s)*d-(e-s)*t).dist()/d;
}
```

### SegmentIntersection.h

**Description:**  
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.

**Usage:** vector<P> inter = segInter(s1,e1,s2,e2);  
if (sz(inter)==1)  
cout << "segments intersect at " << inter[0] << endl;

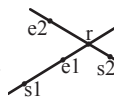


```
"Point.h", "OnSegment.h"
9d57f2, 13 lines

template<class P> vector<P> segInter(P a, P b, P c, P d) {
    auto oa = c.cross(d, a), ob = c.cross(d, b),
        oc = a.cross(b, c), od = a.cross(b, d);
    // Checks if intersection is single non-endpoint point.
    if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
        return {(a * ob - b * oa) / (ob - oa)};
    set<P> s;
    if (onSegment(c, d, a)) s.insert(a);
    if (onSegment(c, d, b)) s.insert(b);
    if (onSegment(a, b, c)) s.insert(c);
    if (onSegment(a, b, d)) s.insert(d);
    return {all(s)};
}
```

### lineIntersection.h

**Description:**  
If a unique intersection point of the lines going through s1,e1 and s2,e2 exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.



**Usage:** auto res = lineInter(s1,e1,s2,e2);  
if (res.first == 1)  
cout << "intersection point at " << res.second << endl;

```
"Point.h"
a01f81, 8 lines

template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
    auto d = (e1 - s1).cross(e2 - s2);
    if (d == 0) // if parallel
        return {-(s1.cross(e1, s2) == 0), P(0, 0)};
    auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
    return {1, (s1 * p + e1 * q) / d};
}
```

### sideOf.h

**Description:** Returns where p is as seen from s towards e. 1/0/-1 ⇔ left/on line/right. If the optional argument eps is given 0 is returned if p is within distance eps from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.

**Usage:** bool left = sideOf(p1,p2,q)==1;

```
"Point.h"
3af81c, 9 lines

template<class P>
int sideOf(P s, P e, P p) { return sgn(s.cross(e, p)); }

template<class P>
int sideOf(const P& s, const P& e, const P& p, double eps) {
    auto a = (e-s).cross(p-s);
    double l = (e-s).dist()*eps;
    return (a > l) - (a < -l);
}
```

### OnSegment.h

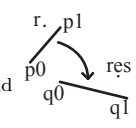
**Description:** Returns true iff p lies on the line segment from s to e. Use (segDist(s,e,p)<=epsilon) instead when using Point<double>.

```
"Point.h"
c597e8, 3 lines

template<class P> bool onSegment(P s, P e, P p) {
    return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 0;
}
```

### linearTransformation.h

**Description:**  
Apply the linear transformation (translation, rotation and scaling) which takes line p0-p1 to line q0-q1 to point r.



```
"Point.h"
03a306, 6 lines

typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) {
    P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
    return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.dist2();
}
```

### LineProjectionReflection.h

**Description:** Projects point p onto line ab. Set refl=true to get reflection of point p across line ab instead. The wrong point will be returned if P is an integer point and the desired point doesn't have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow.

```
"Point.h"
b5562d, 5 lines

template<class P>
P lineProj(P a, P b, P p, bool refl=false) {
    P v = b - a;
    return p - v.perp()*(1+refl)*v.cross(p-a)/v.dist2();
}
```

### Angle.h

**Description:** A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping. Sometimes also represents points or vectors.

**Usage:** vector<Angle> v = {w[0], w[0].t360() ...}; // sorted  
int j = 0; rep(i,0,n) { while (v[j] < v[i].t180()) ++j; }  
// sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i

```
0f0602, 35 lines

struct Angle {
    int x, y;
    int t;
    Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
    Angle operator-(Angle b) const { return {x-b.x, y-b.y, t}; }
    int half() const {
        assert(x || y);
        return y < 0 || (y == 0 && x < 0);
    }
    Angle t90() const { return {-y, x, t + (half() && x >= 0)}; }
    Angle t180() const { return {-x, -y, t + half()}; }
    Angle t360() const { return {x, y, t + 1}; }
};

bool operator<(Angle a, Angle b) {
    // add a.dist2() and b.dist2() to also compare distances
    return make_tuple(a.t, a.half(), a.y * (11)b.x) <
        make_tuple(b.t, b.half(), a.x * (11)b.y);
}
```

// Given two points, this calculates the smallest angle between them, i.e., the angle that covers the defined line segment.

pair<Angle, Angle> segmentAngles(Angle a, Angle b) {  
 if (b < a) swap(a, b);  
 return (b < a.t180() ?  
 make\_pair(a, b) : make\_pair(b, a.t360()));  
}  
Angle operator+(Angle a, Angle b) { // point a + vector b  
 Angle r(a.x + b.x, a.y + b.y, a.t);  
 if (a.t180() < r) r.t--;  
 return r.t180() < a ? r.t360() : r;  
}  
Angle angleDiff(Angle a, Angle b) { // angle b - angle a  
 int tu = b.t - a.t; a.t = b.t;  
 return {a.x\*b.x + a.y\*b.y, a.x\*b.y - a.y\*b.x, tu - (b < a)};  
}

## 7.2 Circles

### CircleIntersection.h

**Description:** Computes the pair of points at which two circles intersect. Returns false in case of no intersection.

```
"Point.h"
84d6d3, 11 lines

typedef Point<double> P;
bool circleInter(P a,P b,double r1,double r2,pair<P, P>* out) {
    if (a == b) { assert(r1 != r2); return false; }
    P vec = b - a;
    double d2 = vec.dist2(), sum = r1+r2, dif = r1-r2,
        p = (d2 + r1*r1 - r2*r2)/(d2*2), h2 = r1*r1 - p*p*d2;
    if (sum*sum < d2 || dif*dif > d2) return false;
    P mid = a + vec*p, per = vec.perp() * sqrt(fmax(0, h2) / d2);
    *out = {mid + per, mid - per};
    return true;
}
```

### CircleTangents.h

**Description:** Finds the external tangents of two circles, or internal if r2 is negated. Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .first = .second and the tangent line is perpendicular to the line between the centers). .first and .second give the tangency points at circle 1 and 2 respectively. To find the tangents of a circle with a point set r2 to 0.

"Point.h"	b0153d, 13 lines
<pre>template&lt;class P&gt; vector&lt;pair&lt;P, P&gt;&gt; tangents(P c1, double r1, P c2, double r2) {     P d = c2 - c1;     double dr = r1 - r2, d2 = d.dist2(), h2 = d2 - dr * dr;     if (d2 == 0    h2 &lt; 0) return {};     vector&lt;pair&lt;P, P&gt;&gt; out;     for (double sign : {-1, 1}) {         P v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;         out.push_back({c1 + v * r1, c2 + v * r2});     }     if (h2 == 0) out.pop_back();     return out; }</pre>	

CircleLine.h

**Description:** Finds the intersection between a circle and a line. Returns a vector of either 0, 1, or 2 intersection points. P is intended to be Point<double>.

"Point.h"	e0cfba, 9 lines
<pre>template&lt;class P&gt; vector&lt;P&gt; circleLine(P c, double r, P a, P b) {     P ab = b - a, p = a + ab * (c-a).dot(ab) / ab.dist2();     double s = a.cross(b, c), h2 = r*r - s*s / ab.dist2();     if (h2 &lt; 0) return {};     if (h2 == 0) return {p};     P h = ab.unit() * sqrt(h2);     return {p - h, p + h}; }</pre>	

CirclePolygonIntersection.h

**Description:** Returns the area of the intersection of a circle with a ccw polygon.

**Time:**  $\mathcal{O}(n)$

"../../content/geometry/Point.h"	alee63, 19 lines
<pre>typedef Point&lt;double&gt; P; #define arg(p, q) atan2(p.cross(q), p.dot(q)) double circlePoly(P c, double r, vector&lt;P&gt; ps) {     auto tri = [&amp;](P p, P q) {         auto r2 = r * r / 2;         P d = q - p;         auto a = d.dot(p)/d.dist2(), b = (p.dist2()-r*r)/d.dist2();         auto det = a * a - b;         if (det &lt;= 0) return arg(p, q) * r2;         auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det));         if (t &lt; 0    1 &lt;= s) return arg(p, q) * r2;         P u = p + d * s, v = p + d * t;         return arg(p,u) * r2 + u.cross(v)/2 + arg(v,q) * r2;     };     auto sum = 0.0;     rep(i,0,sz(ps))         sum += tri(ps[i] - c, ps[(i + 1) % sz(ps)] - c);     return sum; }</pre>	

circumcircle.h

Description:

The circumcirle of a triangle is the circle intersecting all three vertices. ccRadius returns the radius of the circle going through points A, B and C and ccCenter returns the center of the same circle.

"Point.h"	1caa3a, 9 lines
<pre>typedef Point&lt;double&gt; P; double ccRadius(const P&amp; A, const P&amp; B, const P&amp; C) {     return (B-A).dist()*(C-B).dist()*(A-C).dist()/         abs((B-A).cross(C-A))/2; } P ccCenter(const P&amp; A, const P&amp; B, const P&amp; C) {     P b = C-A, c = B-A;     return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2; }</pre>	

MinimumEnclosingCircle.h

**Description:** Computes the minimum circle that encloses a set of points.

**Time:** expected  $\mathcal{O}(n)$

"circumcircle.h"	09dd0a, 17 lines
<pre>pair&lt;P, double&gt; mec(vector&lt;P&gt; ps) {     shuffle(all(ps), mt19937(time(0)));     P o = ps[0];     double r = 0, EPS = 1 + 1e-8;     rep(i,0,sz(ps)) if ((o - ps[i]).dist() &gt; r * EPS) {         o = ps[i], r = 0;         rep(j,0,i) if ((o - ps[j]).dist() &gt; r * EPS) {             o = (ps[i] + ps[j]) / 2;             r = (o - ps[i]).dist();             rep(k,0,j) if ((o - ps[k]).dist() &gt; r * EPS) {                 o = ccCenter(ps[i], ps[j], ps[k]);                 r = (o - ps[i]).dist();             }         }     }     return {o, r}; }</pre>	

7.3 Polygons

InsidePolygon.h

**Description:** Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.

**Usage:** vector<P> v = {P{4,4}, P{1,2}, P{2,1}};

bool in = inPolygon(v, P{3, 3}, false);

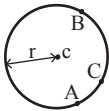
**Time:**  $\mathcal{O}(n)$

"Point.h", "OnSegment.h", "SegmentDistance.h"	2bf504, 11 lines
<pre>template&lt;class P&gt; bool inPolygon(vector&lt;P&gt; &amp;p, P a, bool strict = true) {     int cnt = 0, n = sz(p);     rep(i,0,n) {         P q = p[(i + 1) % n];         if (onSegment(p[i], q, a)) return !strict;         //or: if (segDist(p[i], q, a) &lt;= eps) return !strict;         cnt ^= ((a.y&lt;p[i].y) - (a.y&lt;q.y)) * a.cross(p[i], q) &gt; 0;     }     return cnt; }</pre>	

PolygonArea.h

**Description:** Returns twice the signed area of a polygon. Clockwise enumeration gives negative area. Watch out for overflow if using int as T!

"Point.h"	f12300, 6 lines
<pre>template&lt;class T&gt; T polygonArea2(vector&lt;Point&lt;T&gt;&gt;&amp; v) {     T a = v.back().cross(v[0]);</pre>	



rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
return a;
}

PolygonCenter.h

**Description:** Returns the center of mass for a polygon.

**Time:**  $\mathcal{O}(n)$

"Point.h"	9706dc, 9 lines
<pre>typedef Point&lt;double&gt; P; P polygonCenter(const vector&lt;P&gt;&amp; v) {     P res(0, 0); double A = 0;     for (int i = 0, j = sz(v) - 1; i &lt; sz(v); j = i++) {         res = res + (v[i] + v[j]) * v[j].cross(v[i]);         A += v[j].cross(v[i]);     }     return res / A / 3; }</pre>	

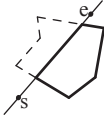
PolygonCut.h

**Description:** Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.

**Usage:** vector<P> p = ...;

p = polygonCut(p, P(0,0), P(1,0));

"Point.h", "lineIntersection.h"	f2b7d4, 13 lines
<pre>typedef Point&lt;double&gt; P; vector&lt;P&gt; polygonCut(const vector&lt;P&gt;&amp; poly, P s, P e) {     vector&lt;P&gt; res;     rep(i,0,sz(poly)) {         P cur = poly[i], prev = i ? poly[i-1] : poly.back();         bool side = s.cross(e, cur) &lt; 0;         if (side != (s.cross(e, prev) &lt; 0))             res.push_back(lineInter(s, e, cur, prev).second);         if (side)             res.push_back(cur);     }     return res; }</pre>	



PolygonUnion.h

**Description:** Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)

**Time:**  $\mathcal{O}(N^2)$ , where N is the total number of points

"Point.h", "sideOf.h"	3931c6, 33 lines
<pre>typedef Point&lt;double&gt; P; double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y; } double polyUnion(vector&lt;vector&lt;P&gt;&gt;&amp; poly) {     double ret = 0;     rep(i,0,sz(poly)) rep(v,0,sz(poly[i])) {         P A = poly[i][v], B = poly[i][(v + 1) % sz(poly[i])];         vector&lt;pair&lt;double, int&gt;&gt; segs = {{0, 0}, {1, 0}};         rep(j,0,sz(poly)) if (i != j) {             rep(u,0,sz(poly[j])) {                 P C = poly[j][u], D = poly[j][(u + 1) % sz(poly[j])];                 int sc = sideOf(A, B, C), sd = sideOf(A, B, D);                 if (sc != sd) {                     double sa = C.cross(D, A), sb = C.cross(D, B);                     if (min(sc, sd) &lt; 0)                         segs.emplace_back(sa / (sa - sb), sgn(sc - sd));                 } else if (!sc &amp;&amp; !sd &amp;&amp; j&lt;i &amp;&amp; sgn((B-A).dot(D-C))&gt;0) {                     segs.emplace_back(rat(C - A, B - A), 1);                     segs.emplace_back(rat(D - A, B - A), -1);                 }             }         }     }</pre>	

```
typedef Point<int> P;
vector<array<int, 3>> manhattanMST(vector<P> ps) {
    vi id(sz(ps));
    iota(all(id), 0);
    vector<array<int, 3>> edges;
    rep(k,0,4) {
        sort(all(id), [&](int i, int j) {
            return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;});
        map<int, int> sweep;
        for (int i : id) {
            for (auto it = sweep.lower_bound(-ps[i].y);
                 it != sweep.end(); sweep.erase(it++)) {
                int j = it->second;
                P d = ps[i] - ps[j];
                if (d.y > d.x) break;
            }
        }
    }
}
```

```
edges.push_back({d.y + d.x, i, j});
}
sweep[-ps[i].y] = i;
}
for (P& p : ps) if (k & 1) p.x = -p.x; else swap(p.x, p.y);
}
return edges;
}
```

kdTree.h

**Description:** KD-tree (2d, can be extended to 3d)

"Point.h"	bac5b0, 63 lines
-----------	------------------

```
typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();
```

```
bool on_x(const P& a, const P& b) { return a.x < b.x; }
bool on_y(const P& a, const P& b) { return a.y < b.y; }
```

```
struct Node {
    P pt; // if this is a leaf, the single point in it
    T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF; // bounds
    Node *first = 0, *second = 0;

    T distance(const P& p) { // min squared distance to a point
        T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
        T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
        return (P(x,y) - p).dist2();
    }
}
```

```
Node(vector<P>&& vp) : pt(vp[0]) {
    for (P p : vp) {
        x0 = min(x0, p.x); x1 = max(x1, p.x);
        y0 = min(y0, p.y); y1 = max(y1, p.y);
    }
    if (vp.size() > 1) {
        // split on x if width >= height (not ideal...)
        sort(all(vp), x1 - x0 >= y1 - y0 ? on_x : on_y);
        // divide by taking half the array for each child (not
        // best performance with many duplicates in the middle)
        int half = sz(vp)/2;
        first = new Node({vp.begin(), vp.begin() + half});
        second = new Node({vp.begin() + half, vp.end()});
    }
}
```

```
struct KDTree {
    Node* root;
    KDTree(const vector<P>& vp) : root(new Node({all(vp)})) {}

    pair<T, P> search(Node *node, const P& p) {
        if (!node->first) {
            // uncomment if we should not find the point itself:
            // if (p == node->pt) return {INF, P()};
            return make_pair((p - node->pt).dist2(), node->pt);
        }

        Node *f = node->first, *s = node->second;
        T bfirst = f->distance(p), bsec = s->distance(p);
        if (bfirst > bsec) swap(bsec, bfirst), swap(f, s);

        // search closest side first, other side if needed
        auto best = search(f, p);
        if (bsec < best.first)
            best = min(best, search(s, p));
        return best;
    }
}
```

kdTree FastDelaunay PolyhedronVolume Point3D

```
// find nearest point to a point, and its squared distance
// (requires an arbitrary operator< for Point)
pair<T, P> nearest(const P& p) {
    return search(root, p);
}
};
```

FastDelaunay.h

**Description:** Fast Delaunay triangulation. Each circumcircle contains none of the input points. There must be no duplicate points. If all points are on a line, no triangles will be returned. Should work for doubles as well, though there may be precision issues in 'circ'. Returns triangles in order {t[0][0], t[0][1], t[0][2], t[1][0], ...}, all counter-clockwise.

**Time:**  $\mathcal{O}(n \log n)$

"Point.h"	eefdf5, 88 lines
-----------	------------------

```
typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX, LLONG_MAX); // not equal to any other point
```

```
struct Quad {
    Q rot, o; P p = arb; bool mark;
    P& F() { return r()->p; }
    Q& r() { return rot->rot; }
    Q prev() { return rot->o->rot; }
    Q next() { return r()->prev(); }
} *H;
```

```
bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
    lll p2 = p.dist2(), A = a.dist2()-p2,
        B = b.dist2()-p2, C = c.dist2()-p2;
    return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B > 0;
}
```

```
Q makeEdge(P orig, P dest) {
    Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}}};
    H = r->o; r->r()->r() = r;
    rep(i,0,4) r = r->rot, r->p = arb, r->o = i & 1 ? r : r->r();
    r->p = orig; r->F() = dest;
    return r;
}
```

```
void splice(Q a, Q b) {
    swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
    Q q = makeEdge(a->F(), b->p);
    splice(q, a->next());
    splice(q->r(), b);
    return q;
}
```

```
pair<Q,Q> rec(const vector<P>& s) {
    if (sz(s) <= 3) {
        Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back());
        if (sz(s) == 2) return { a, a->r() };
        splice(a->r(), b);
        auto side = s[0].cross(s[1], s[2]);
        Q c = side ? connect(b, a) : 0;
        return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
    }
}
```

```
#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
Q A, B, ra, rb;
int half = sz(s) / 2;
tie(ra, A) = rec({all(s) - half});
tie(B, rb) = rec({sz(s) - half + all(s)});
while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
        (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
Q base = connect(B->r(), A);
```

```
if (A->p == ra->p) ra = base->r();
if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    while (circ(e->dir->F(), H(base), e->F())) { \
        Q t = e->dir; \
        splice(e, e->prev()); \
        splice(e->r(), e->r()->prev()); \
        e->o = H; H = e; e = t; \
    }
for (;;) {
    DEL(LC, base->r(), o); DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
        base = connect(RC, base->r());
    else
        base = connect(base->r(), LC->r());
}
return { ra, rb };
}
```

```
vector<P> triangulate(vector<P> pts) {
    sort(all(pts)); assert(unique(all(pts)) == pts.end());
    if (sz(pts) < 2) return {};
    Q e = rec(pts).first;
    vector<Q> q = {e};
    int qi = 0;
    while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
    #define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p); \
        q.push_back(c->r()); c = c->next(); } while (c != e); }
    ADD; pts.clear();
    while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
    return pts;
}
```

7.5 3D

PolyhedronVolume.h

**Description:** Magic formula for the volume of a polyhedron. Faces should point outwards.

template<class V, class L> double signedPolyVolume(const V& p, const L& trilst) { double v = 0; for (auto i : trilst) v += p[i.a].cross(p[i.b]).dot(p[i.c]); return v / 6; }	3058c3, 6 lines
--	-----------------

Point3D.h

**Description:** Class to handle points in 3D space. T can be e.g. double or long long.

template<class T> struct Point3D { typedef Point3D P; typedef const P& R; T x, y, z; explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {} bool operator<(R p) const { return tie(x, y, z) < tie(p.x, p.y, p.z); } bool operator==(R p) const { return tie(x, y, z) == tie(p.x, p.y, p.z); } operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); } operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); } operator*(T d) const { return P(x*d, y*d, z*d); } operator/(T d) const { return P(x/d, y/d, z/d); } T dot(R p) const { return x*p.x + y*p.y + z*p.z; } P cross(R p) const { return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x); } } T dist2() const { return x*x + y*y + z*z; }	8058ae, 32 lines
---	------------------

```
double dist() const { return sqrt((double)dist2()); }
//Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
double phi() const { return atan2(y, x); }
//Zenith angle (latitude) to the z-axis in interval [0, pi]
double theta() const { return atan2(sqrt(x*x+y*y),z); }
P unit() const { return *this/(T)dist(); } //makes dist()==1
//returns unit vector normal to *this and p
P normal(P p) const { return cross(p).unit(); }
//returns point rotated 'angle' radians ccw around axis
P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit();
    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
}
};
```

3dHull.h

**Description:** Computes all faces of the 3-dimension hull of a point set. \*No four points must be coplanar\*, or else random results will be returned. All faces will point outwards.

**Time:**  $O(n^2)$

"Point3D.h" 5b45fc, 49 lines

```
typedef Point3D<double> P3;
```

```
struct PR {
void ins(int x) { (a == -1 ? a : b) = x; }
void rem(int x) { (a == x ? a : b) = -1; }
int cnt() { return (a != -1) + (b != -1); }
int a, b;
};
```

```
struct F { P3 q; int a, b, c; };
```

```
vector<F> hull3d(const vector<P3>& A) {
    assert(sz(A) >= 4);
    vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
#define E(x,y) E[f.x][f.y]
    vector<F> FS;
    auto mf = [&](int i, int j, int k, int l) {
        P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
        if (q.dot(A[l]) > q.dot(A[i]))
            q = q * -1;
        F f{q, i, j, k};
        E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
        FS.push_back(f);
    };
    rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
        mf(i, j, k, 6 - i - j - k);

    rep(i,4,sz(A)) {
        rep(j,0,sz(FS)) {
            F f = FS[j];
            if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
                E(a,b).rem(f.c);
                E(a,c).rem(f.b);
                E(b,c).rem(f.a);
                swap(FS[j--], FS.back());
                FS.pop_back();
            }
        }
        int nw = sz(FS);
        rep(j,0,nw) {
            F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.c);
            C(a, b, c); C(a, c, b); C(b, c, a);
        }
    }
    for (F& it : FS) if ((A[it.b] - A[it.a]).cross(
        A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
    return FS;
};
```

};

sphericalDistance.h

**Description:** Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 ( $\phi_1$ ) and f2 ( $\phi_2$ ) from x axis and zenith angles (latitude) t1 ( $\theta_1$ ) and t2 ( $\theta_2$ ) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx\*radius is then the difference between the two points in the x direction and d\*radius is the total distance between the points.

```
double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
    double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
    double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
    double dz = cos(t2) - cos(t1);
    double d = sqrt(dx*dx + dy*dy + dz*dz);
    return radius*2*asin(d/2);
}
```

Strings (8)

KMP.h

**Description:** kmp[x] computes the length of the longest prefix of s that ends at x (abacaba -> 0010123). Time: O(n) aut[i][j] = the length of the longest prefix of s that equals at s[0..i] + a + j. Time: O(n \* 26)

55ce30, 26 lines

```
vector<int> kmp(const string& s) {
    vector<int> p(sz(s));
    rep(i, 1, sz(s)) {
        int g = p[i - 1];
        while (g && s[i] != s[g]) g = p[g - 1];
        p[i] = g + (s[i] == s[g]);
    }
    return p;
}

vector<int> match(const string& s, const string& pat) {
    vi p = kmp(pat + '\0' + s), res;
    rep(i,sz(p)-sz(s),sz(p))
        if (p[i] == sz(pat)) res.push_back(i - 2 * sz(pat));
    return res;
}

const char a = 'a'; const int alpha = 26;
void compute_automaton(string s, vector<vector<int>>& aut) {
    s += '#'; int n = s.size();
    vector<int> pi = kmp(s);
    aut.assign(n, vector<int>(alpha));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < alpha; c++) {
            if (i > 0 && a + c != s[i])
                aut[i][c] = aut[pi[i - 1]][c];
            else
                aut[i][c] = i + (a + c == s[i]);
        }
    }
}
```

Zfunc.h

**Description:** z[x] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)

8382b3, 9 lines

```
vector<int> Z(const string& S) {
    vector<int> z(sz(S));
    int l = -1, r = -1;
    rep(i, 1, sz(S)) {
        z[i] = i >= r ? 0 : min(r - i, z[i - 1]);
        while (i + z[i] < sz(S) && S[i + z[i]] == S[z[i]]) z[i]++;
        if (i + z[i] > r) l = i, r = i + z[i];
    }
    return z;
}
```

Manacher.h

**Description:** For each position in a string, computes p[0][i] = half len of even palin around pos i ([i - len, i + len -1]) p[1][i] = long odd (rounded down) [i - len, i + len].

e59a1d, 12 lines

```
array<vector<int>, 2> manacher(const string& s) {
    int n = sz(s);
    array<vector<int>, 2> p = {vector<int>(n + 1), vector<int>(n)};
    rep(z, 0, 2) for (int i = 0, l = 0, r = 0; i < n; i++) {
        int t = r - i + !z;
        if (i < r) p[z][i] = min(t, p[z][l + t]);
        int L = i - p[z][i], R = i + p[z][i] - !z;
        while (L >= 1 && R + 1 < n && s[L - 1] == s[R + 1]) p[z][i]
            ++, L--, R++;
        if (R > r) l = L, r = R;
    }
    return p;
}
```

MinRotation.h

**Description:** Finds the lexicographically smallest rotation of a string. O(N)

**Usage:** rotate(v.begin(), v.begin() + minRotation(v), v.end())

d07a42, 8 lines

```
int minRotation(string s) {
    int a=0, N=sz(s); s += s;
    rep(b,0,N) rep(k,0,N) {
        if (a+k == b || s[a+k] < s[b+k]) {b += max(0, k-1); break;}
        if (s[a+k] > s[b+k]) {a = b; break;}
    }
    return a;
}
```

SuffixArray.h

**Description:** sa[i] is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size n + 1, and sa[0] = n. The lcp array contains longest common prefixes for neighbouring strings in the suffix array: lcp[i] = lcp(sa[i], sa[i-1]), lcp[0] = 0. remember eqclass size can terminate before logn so take min while comp.

ca5a80, 24 lines

```
struct SuffixArray {
    vector<int> sa, lcp;
    vector<vector<int>> > eqclass;
    SuffixArray(string& s, int lim = 256) {
        int n = sz(s) + 1, k = 0, a, b;
        vector<int> x(all(s) + 1), y(n), ws(max(n, lim)), rank(n);
        sa = lcp = y, iota(all(sa), 0);
        for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
            p = j, iota(all(y), n - j);
            rep(i, 0, n) if (sa[i] >= j) y[p++] = sa[i] - j;
            fill(all(ws), 0);
            rep(i, 0, n) ws[x[i]]++;
            rep(i, 1, lim) ws[i] += ws[i - 1];
            for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
            swap(x, y), p = 1, x[sa[0]] = 0;
            rep(i, 1, n) a = sa[i - 1], b = sa[i],
                x[b] = (y[a] == y[b] && y[a + j] == y[b + j]) ? p
                    - 1 : p++;
            eqclass.pb(x);
        }
        rep(i, 1, n) rank[sa[i]] = i;
        for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
            for (k && k--, j = sa[rank[i] - 1]; s[i + k] == s[j + k];
                k++);
    }
};
```



SuffixTree.h

**Description:** Ukkonen’s algorithm for online suffix tree construction. Each node contains indices [l, r] into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r] substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though). finding the lexicographically kth smallest substring (repeation allowed). (s + ('z' + 1)) dfs on nodes to store subtree info dfs to find the kth on ( remember edges have more than 1 char) array of transitions (state, letter), left and right boundaries of the substring of a which correspond to incoming edge p[N] = parent of the node, s[N] = suffix link, m = the num. of nodes v = cur node, q = cur position

c6a175, 73 lines

```
struct SuffixTree {
    enum { ALPHA = 28 };
    int toi(char c) { return c - 'a'; }
    string a;
    int mxn, v = 0, q = 0, m = 2;
    vi r, s, p, l; vector<array<int, ALPHA>> t;
    vector<ll> nodes, leaf;

    SuffixTree(string _a):a(_a),mxn{2*sz(a)+10},r(mxn,sz(a)),s(
        mxn),p(mxn),l(mxn),t(mxn),nodes(mxn),leaf(mxn) {
        for (auto& x : t) fill(all(x), -1);
        fill(all(t[1]), 0); s[0] = 1;
        l[0] = l[1] = -1; r[0] = r[1] = 0;
        rep(i, 0, sz(a)) ukkadd(i, toi(a[i]));}

    void ukkadd(int i, int c) {
        suff:
        if (r[v] <= q) {
            if (t[v][c] == -1) {
                t[v][c] = m, l[m] = i;
                p[m++] = v, v = s[v], q = r[v];
                goto suff;}
            v = t[v][c];
            q = l[v];}
        if (q == -1 || c == toi(a[q])) ++q;
        else {
            l[m + 1] = i, p[m + 1] = m;
            l[m] = l[v], r[m] = q;
            p[m] = p[v], t[m][c] = m + 1;
            t[m][toi(a[q])] = v;
            l[v] = q, p[v] = m;
            t[p[m]][toi(a[l[m]])] = m;
            v = s[p[m]], q = l[m];
            while (q < r[m]) {
                v = t[v][toi(a[q])];
                q += r[v] - l[v];}
            s[m] = q == r[m] ? v : m + 2;
            q = r[v] - (q - r[m]);
            m += 2;
            goto suff;}}

    void dfs_nodes(int node) {
        bool pre = false;
        nodes[node] = 0, leaf[node] = 0;
        rep(c, 0, ALPHA) if (t[node][c] != -1) {
            pre |= true; dfs_nodes(t[node][c]);
            nodes[node] += nodes[t[node][c]];
            leaf[node] += leaf[t[node][c]];}
        if (node) { if (!pre) {
            leaf[node] = 1, nodes[node] = r[node] - l[node] - 1;
        } else
            nodes[node] += (r[node] - l[node]) * leaf[node];}
    };
    // void kth(int node, ll k, string& ans) {
    //     int cur = max(l[node], 0);
    //     while (k > 0 and cur < r[node]) {
```

```
//         k -= leaf[node], ans += a[cur++];
//     }
//     if (k <= 0) return;
//     rep(c, 0, ALPHA) if (t[node][c] != -1) {
//         if (nodes[t[node][c]] >= k) {
//             kth(t[node][c], k, ans);
//             return;
//         }
//         k -= nodes[t[node][c]];
//     }
// }
// static string kth_substring(string s, ll k) {
//     SuffixTree st(s + (char)('z' + 1));
//     st.dfs_nodes(0);
//     string ans = "";
//     st.kth(0, k, ans);
//     return ans;
// }
```

Hashing.h

**Description:** Self-explanatory methods for string hashing. 2d2a67, 44 lines

```
// Arithmetic mod 2^64-1. 2x slower than mod 2^64 and more
// code, but works on evil test data (e.g. Thue-Morse, where
// ABBA... and BAAB... of length 2^10 hash the same mod 2^64).
// "typedef ull H;" instead if you think test data is random,
// or work mod 10^9+7 if the Birthday paradox is not a problem.
typedef uint64_t ull;
struct H {
    ull x; H(ull x=0) : x(x) {}
    H operator+(H o) { return x + o.x + (x + o.x < x); }
    H operator-(H o) { return *this + ~o.x; }
    H operator*(H o) { auto m = (__uint128_t)x * o.x;
        return H((ull)m) + (ull)(m >> 64); }
    ull get() const { return x + !~x; }
    bool operator==(H o) const { return get() == o.get(); }
    bool operator<(H o) const { return get() < o.get(); }
};
static const H C = (1ll)1e11+3; // (order ~ 3e9; random also ok)

struct HashInterval {
    vector<H> ha, pw;
    HashInterval(string& str) : ha(sz(str)+1), pw(ha) {
        pw[0] = 1;
        rep(i,0,sz(str))
            ha[i+1] = ha[i] * C + str[i],
            pw[i+1] = pw[i] * C;
    }
    H hashInterval(int a, int b) { // hash [a, b)
        return ha[b] - ha[a] * pw[b - a];
    }
};

vector<H> getHashes(string& str, int length) {
    if (sz(str) < length) return {};
    H h = 0, pw = 1;
    rep(i,0,length)
        h = h * C + str[i], pw = pw * C;
    vector<H> ret = {h};
    rep(i,length,sz(str)) {
        ret.push_back(h = h * C + str[i] - pw * str[i-length]);
    }
    return ret;
}

H hashString(string& s){H h{}; for(char c:s) h=h*C+c;return h;}
```

PalindromeTree.h

**Description:** O(N \* 26) node 1 - root with len -1, node 2 - root with len 0 , max suffix palindrome num in node stores depth of the node along suffixlink

4b43d1, 45 lines

```
struct PalindromeTree {
    static constexpr int ALPHA = 26;
    struct node {
        int next[ALPHA], len, sufflink, num;
        node() { memset(next, 0, sizeof(next)); }
    };
    string s; vector<node> tree;
    int num, suff; long long ans = 0;
    bool addLetter(int pos) {
        int cur = suff, curlen = 0, let = s[pos] - 'a';
        while (true) {
            curlen = tree[cur].len;
            if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos
                ]) break;
            cur = tree[cur].sufflink;
        }
        if (tree[cur].next[let]) {
            suff = tree[cur].next[let];
            return false;
        }
        suff = ++num;
        tree[num].len = tree[cur].len + 2;
        tree[cur].next[let] = num;
        if (tree[num].len == 1) {
            tree[num].sufflink = 2, tree[num].num = 1;
            return true;
        }
        while (true) {
            cur = tree[cur].sufflink, curlen = tree[cur].len;
            if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos
                ]) {
                tree[num].sufflink = tree[cur].next[let];
                break;
            }
        }
        tree[num].num = 1 + tree[tree[num].sufflink].num;
        return true;
    }
};

PalindromeTree(string s) : s(s), tree(sz(s) + 10) {
    num = 2, suff = 2, tree[1].len = -1;
    tree[1].sufflink = 1; tree[2].len = 0;
    tree[2].sufflink = 1;
    rep(i, 0, sz(s)) {addLetter(i);
        ans += tree[suff].num;}
    }
};
```

AhoCorasick.h

**Description:** Initialize with AhoCorasick ac(patterns); the automaton start node will be at index 0. backp stores the ind of the longest string which forms a suffix of the string ending at i. back info in a node is the suffix link. final N gives the transition from find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll(–, word) finds all words (up to  $N\sqrt{N}$  many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. **Time:** construction takes  $\mathcal{O}(26N)$ , where  $N$  = sum of length of patterns.

fa079e, 80 lines

```
struct AhoCorasick {
    enum { alpha = 26, first = 'a' }; // Remeber to change this!
    struct Node {
        int back, next[alpha], start = -1, end = -1, nmatches = 0;
        // (nmatches is optional)
        Node(int v) { memset(next, v, sizeof(next)); }
        // add variables to store extra elements to back prop
```



```
// int vis = 0;
};
vector<Node> N;
vector<int> backp; // endpos
vector<vector<int>> > adj;
void insert(string& s, int j) {
    assert(!s.empty());
    int n = 0;
    for (char c : s) {
        int& m = N[n].next[c - first];
        if (m == -1) { n = m = sz(N);
            N.emplace_back(-1);
        } else n = m;
    }
    if (N[n].end == -1) N[n].start = j;
    backp.push_back(N[n].end);
    N[n].end = j; N[n].nmatches++;
    // endpos[j] = n;
}
AhoCorasick(vector<string>& pat) : N(1, -1) {
    // endpos.resize(sz(pat));
    rep(i, 0, sz(pat)) insert(pat[i], i);
    N[0].back = sz(N); N.emplace_back(0);
    queue<int> q;
    for (q.push(0); !q.empty(); q.pop()) {
        int n = q.front(), prev = N[n].back;
        rep(i, 0, alpha) {
            int &ed = N[n].next[i], y = N[prev].next[i];
            if (ed == -1) ed = y;
            else {
                N[ed].back = y;
                (N[ed].end == -1 ? N[ed].end : backp[N[ed].start]) =
                    N[y].end;
                N[ed].nmatches += N[y].nmatches;
                q.push(ed);
            }
        }
    }
    // take out info of patterns form a tree using .back and
    // prop back
    adj.resize(sz(N));
    rep(i, 0, sz(N)-1)
        adj[N[i].back].push_back(i);
}
// if you need to something up the tree
// void treeup(int u){
//     rep(i, 0, sz(adj[u])){
//         treeup(adj[u][i]);
//         N[u].vis += N[adj[u][i]].vis;
//     }
// }
vector<int> find(string word) {
    int n = 0;
    vector<int> res; // ll count = 0;
    for (char c : word) {
        n = N[n].next[c - first];
        res.push_back(N[n].end);
        // N[n].vis++;
        // count += N[n].nmatches;
    }
    // treeup(sz(N)-1);
    return res;
}
vector<vector<int>> findAll(vector<string>& pat, string word)
{
    vector<int> r = find(word);
    vector<vector<int>> res(sz(word));
    rep(i, 0, sz(word)) {
        int ind = r[i];
```

```
while (ind != -1) {
    res[i - sz(pat[ind]) + 1].push_back(ind);
    ind = backp[ind];
}
}
return res;
}
};
```

Various (9)

9.1 Intervals

IntervalContainer.h

**Description:** Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).  
**Time:**  $\mathcal{O}(\log N)$

```
set<pii>::iterator addInterval(set<pii>& is, int L, int R) {
    if (L == R) return is.end();
    auto it = is.lower_bound({L, R}), before = it;
    while (it != is.end() && it->first <= R) {
        R = max(R, it->second);
        before = it = is.erase(it);
    }
    if (it != is.begin() && (--it)->second >= L) {
        L = min(L, it->first);
        R = max(R, it->second);
        is.erase(it);
    }
    return is.insert(before, {L, R});
}
```

```
void removeInterval(set<pii>& is, int L, int R) {
    if (L == R) return;
    auto it = addInterval(is, L, R);
    auto r2 = it->second;
    if (it->first == L) is.erase(it);
    else (int&)it->second = L;
    if (R != r2) is.emplace(R, r2);
}
```

IntervalCover.h

**Description:** Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add || R.empty(). Returns empty set on failure (or if G is empty).  
**Time:**  $\mathcal{O}(N \log N)$

```
template<class T>
vi cover(pair<T, T> G, vector<pair<T, T>> I) {
    vi S(sz(I)), R;
    iota(all(S), 0);
    sort(all(S), [&](int a, int b) { return I[a] < I[b]; });
    T cur = G.first;
    int at = 0;
    while (cur < G.second) { // (A)
        pair<T, int> mx = make_pair(cur, -1);
        while (at < sz(I) && I[S[at]].first <= cur) {
            mx = max(mx, make_pair(I[S[at]].second, S[at]));
            at++;
        }
        if (mx.second == -1) return {};
        cur = mx.first;
        R.push_back(mx.second);
    }
    return R;
}
```

**ConstantIntervals.h**  
**Description:** Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval.  
**Usage:** constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});  
**Time:**  $\mathcal{O}(k \log \frac{n}{k})$

```
template<class F, class G, class T>
void rec(int from, int to, F& f, G& g, int& i, T& p, T q) {
    if (p == q) return;
    if (from == to) {
        g(i, to, p);
        i = to; p = q;
    } else {
        int mid = (from + to) >> 1;
        rec(from, mid, f, g, i, p, f(mid));
        rec(mid+1, to, f, g, i, p, q);
    }
}
template<class F, class G>
void constantIntervals(int from, int to, F f, G g) {
    if (to <= from) return;
    int i = from; auto p = f(i), q = f(to-1);
    rec(from, to-1, f, g, i, p, q);
    g(i, to, q);
}
```

9.2 Misc. algorithms

TernarySearch.h

**Description:** Find the smallest i in [a,b] that maximizes f(i), assuming that f(a) < ... < f(i) ≥ ... ≥ f(b). To reverse which of the sides allows non-strict inequalities, change the < marked with (A) to <=, and reverse the loop at (B). To minimize f, change it to >, also at (B).  
**Usage:** int ind = ternSearch(0,n-1,[&](int i){return a[i];});  
**Time:**  $\mathcal{O}(\log(b-a))$

```
template<class F>
int ternSearch(int a, int b, F f) {
    assert(a <= b);
    while (b - a >= 5) {
        int mid = (a + b) / 2;
        if (f(mid) < f(mid+1)) a = mid; // (A)
        else b = mid+1;
    }
    rep(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
    return a;
}
```

LIS.h

**Description:** Compute indices for the longest increasing subsequence.  
**Time:**  $\mathcal{O}(N \log N)$

```
template<class I> vi lis(const vector<I>& S) {
    if (S.empty()) return {};
    vi prev(sz(S));
    typedef pair<I, int> p;
    vector<p> res;
    rep(i, 0, sz(S)) {
        // change 0 -> i for longest non-decreasing subsequence
        auto it = lower_bound(all(res), p{S[i], 0});
        if (it == res.end()) res.emplace_back(), it = res.end()-1;
        *it = {S[i], i};
        prev[i] = it == res.begin() ? 0 : (it-1)->second;
    }
    int L = sz(res), cur = res.back().second;
    vi ans(L);
```

```
while (L--) ans[L] = cur, cur = prev[cur];
return ans;
}
```

## FastKnapsack.h

**Description:** Given N non-negative integer weights w and a non-negative target t, computes the maximum  $S \leq t$  such that S is the sum of some subset of the weights.

**Time:**  $\mathcal{O}(N \max(w_i))$

b20ccc, 16 lines

```
int knapsack(vi w, int t) {
    int a = 0, b = 0, x;
    while (b < sz(w) && a + w[b] <= t) a += w[b++];
    if (b == sz(w)) return a;
    int m = *max_element(all(w));
    vi u, v(2*m, -1);
    v[a+m-t] = b;
    rep(i, b, sz(w)) {
        u = v;
        rep(x, 0, m) v[x+w[i]] = max(v[x+w[i]], u[x]);
        for (x = 2*m; --x > m;) rep(j, max(0, u[x]), v[x])
            v[x-w[j]] = max(v[x-w[j]], j);
    }
    for (a = t; v[a+m-t] < 0; a--);
    return a;
}
```

### 9.3 Dynamic programming

## KnuthDP.h

**Description:** When doing DP on intervals:  $a[i][j] = \min_{i \leq k < j} (a[i][k] + A[k][j]) + f(i, j)$ , where the (minimal) optimal  $k$  increases with both  $i$  and  $j$ , one can solve intervals in increasing order of length, and search  $k = p[i][j]$  for  $a[i][j]$  only between  $p[i][j - 1]$  and  $p[i + 1][j]$ . This is known as Knuth DP. Sufficient criteria for this are if  $f(b, c) \leq f(a, d)$  and  $f(a, c) + f(b, d) \leq f(a, d) + f(b, c)$  for all  $a \leq b \leq c \leq d$ . Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.

Time:  $\mathcal{O}(N^2)$ 

## DivideAndConquerDP.h

**Description:** Given  $a[i] = \min_{l_o(i) \leq k < h(i)} (f(i, k))$  where the (minimal) optimal  $k$  increases with  $i$ , computes  $\bar{a}[i]$  for  $i = L..R - 1$ .

Time:  $\mathcal{O}((N + (hi - lo)) \log N)$  d38d2b, 18 lines

```

struct DP { // Modify at will:
    int lo(int ind) { return 0; }
    int hi(int ind) { return ind; }
    ll f(int ind, int k) { return dp[ind][k]; }
    void store(int ind, int k, ll v) { res[ind] = pii(k, v); }

    void rec(int L, int R, int LO, int HI) {
        if (L >= R) return;
        int mid = (L + R) >> 1;
        pair<ll, int> best(LLONG_MAX, LO);
        rep(k, max(LO, lo(mid)), min(HI, hi(mid)))
            best = min(best, make_pair(f(mid, k), k));
        store(mid, best.second, best.first);
        rec(L, mid, LO, best.second+1);
        rec(mid+1, R, best.second, HI);
    }

    void solve(int L, int R) { rec(L, R, INT_MIN, INT_MAX); }
};

```

## 9.4 Debugging tricks

- `signal(SIGSEGV, [](int) { _Exit(0); });`  
converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions). `_GLIBCXX_DEBUG` failures generate SIGABRT (or SIGSEGV on gcc 5.4.0 apparently).
- `feenableexcept(29);` kills the program on NaNs (1), 0-divs (4), infinities (8) and denormals (16).

## 9.5 Optimization tricks

`__builtin_ia32_ldmxcsr(40896);` disables denormals (which make floats 20x slower near their minimum value).

### 9.5.1 Bit hacks

- $x \& -x$  is the least bit in  $x$ .
- `for (int x = m; x; ) { --x &= m; ... }` loops over all subset masks of  $m$  (except  $m$  itself).
- `c = x&-x, r = x+c; ((r^x) >> 2)/c` |  $r$  is the next number after  $x$  with the same number of bits set.
- `rep(i, 0, N) rep(S, 0, 1 << N)`  
`if (S >> i & 1) D[S] += D[S ^ (1 << i)];`  
 computes all sums of subsets.

### 9.5.2 Pragmas

- **#pragma** GCC optimize ("Ofast") will make GCC auto-vectorize loops and optimizes floating points better.
- **#pragma** GCC target ("avx2") can double performance of vectorized code, but causes crashes on old machines.
- **#pragma** GCC optimize ("trapv") kills the program on integer overflows (but is really slow).

## FastMod.h

**Description:** Compute  $a\%b$  about 5 times faster than usual, where  $b$  constant but not known at compile time. Returns a value congruent to  $(\text{mod } b)$  in the range  $[0, 2b)$ .

```
using ull = unsigned long long;
struct FastMod {
    ull b, m;
    FastMod(ull b) : b(b), m(-1ULL / b) {}
    ull reduce(ull a) { // a % b + (0 or b)
        return a - (ull)((__uint128_t(m) * a) >> 64) * b;
    }
};
```

## FastInput.h

**Description:** Read an integer from stdin. Usage requires your program to be run with a pipe in input from file.

**Usage:** ./a.out < input.txt  
**Time:** About 5x as fast as cin/scanf.

```
inline char gc() { // like getchar()
    static char buf[1 << 16];
    static size_t bc, be;
    if (bc >= be) {
```

```

    buf[0] = 0, bc = 0;
    be = fread(buf, 1, sizeof(buf), stdin);
}
return buf[bc++]; // returns 0 on EOF
}

```

```
int readInt() {
    int a, c;
    while ((a = gc()) < 40);
    if (a == '-') return -readInt();
    while ((c = gc()) >= 48) a = a * 10 + c - 48;
    return a - 48;
}
```

## BumpAllocator.h

**Description:** When you need to dynamically allocate many objects and don't care about freeing them. "new X" otherwise has an overhead of something like 0.05us + 16 bytes per allocation.

---

745db2, 8 lines

```
// Either globally or in a single class:
static char buf[450 << 20];
void* operator new(size_t s) {
    static size_t i = sizeof buf;
    assert(s < i);
    return (void*)&buf[i -= s];
}
void operator delete(void*) {}
```

## SmallPtr.h

**Description:** A 32-bit pointer that points into BumpAllocator memory.

"BumpAllocator.h"	2dd6c9, 10 lines
-------------------	------------------

```
template<class T> struct ptr {
    unsigned ind;
    ptr(T* p = 0) : ind(p ? unsigned((char*)p - buf) : 0) {
        assert(ind < sizeof buf);
    }
    T& operator*() const { return *(T*)(buf + ind); }
    T* operator->() const { return &***this; }
    T& operator[](int a) const { return (&***this)[a]; }
    explicit operator bool() const { return ind; }
};
```

## BumpAllocatorSTL.h

**Description:** BumpAllocator for STL containers.

```
Usage: vector<vector<int, small<int>>> ed(N);
```

bb66d4, 14 lines

```
char buf[450 << 20] alignas(16);
size_t buf_ind = sizeof buf;
```

```
template<class T> struct small {
    typedef T value_type;
    small() {}
    template<class U> small(const U&) {}
    T* allocate(size_t n) {
        buf_ind -= n * sizeof(T);
        buf_ind &= 0 - alignof(T);
        return (T*)(buf + buf_ind);
    }
    void deallocate(T*, size_t) {}
};
```

## Unrolling.h

---

Onioning.n
520e76, 5 lines

```
#define F {...; ++i;}
int i = from;
while (i&3 && i < to) F // for alignment, if needed
while (i + 4 <= to) { F F F F }
while (i < to) F
```