University of Copenhagen

string & a long

Elias Rasmussen Lolck, Thor Vejen Eriksen, William Bille Meyling

North-Western European Regional Contest

March 8, 2024

Setup

bashrc.sh

5fbff3
```
alias v=nvim
mcd() {
  mkdir "$1" && cd "$1"
}
# fast:
setxkbmap -option caps:escape
xset r rate 200 120
# normal:
xset r rate 500 35
# debug compile (C++):
dc() {
  base_name=$(basename "$1" .cpp)
  command="g++ -Wshadow -Wall $1 -o $base_name -g -
     fsanitize=address,undefined -D_GLIBCXX_DEBUG -std=c++2a
      -Wfatal-errors"
  $command
}
set -o vi
```

hash.sh

5246ca
```
# hashes a file, ignoring whitespaces and comments
# use for verifying that code is copied correctly
cpp -dD -P -fpreprocessed | tr -d '[:space:]' | md5sum |
     cut -c-6
```

init.lua

8f5b51
```
-- the entire init.lua from before, pick and choose:

local options = {
  cmdheight = 1,
  ignorecase = true,
  mouse = "a",
  expandtab = true,
  shiftwidth = 4,
  tabstop = 4,
  cursorline = true,
  number = true,
  relativenumber = true,
  numberwidth = 1,
  signcolumn = "yes",
  wrap = false,
  scrolloff = 6,
  sidescrolloff = 6,
  foldmethod="indent",
  foldlevel=99,
  colorcolumn='80',
}

local keymap = vim.api.nvim_set_keymap
local ops = { noremap = true, silent = true }

keymap("v", "<A-Down>", ":m '>+1<CR>gv=gv", ops)
keymap("v", "<A-Up>", ":m '<-2<CR>gv=gv", ops)
keymap("v", "p", "\"_dP", ops)

for k, v in pairs(options) do
  vim.opt[k] = v
end

local function hashCurrentBuffer()
    local buffer_content = table.concat(vim.api.
     nvim_buf_get_lines(0, 0, -1, false), "\n")
    local command = "echo '"..buffer_content.."' | cpp -dD
     -P -fpreprocessed | tr -d '[:space:]' | md5sum | cut -c
     -6"
    local hash = vim.fn.system(command)
    hash = hash:gsub("%s+", "")
    print("Buffer Hash: " .. hash)
end
vim.api.nvim_create_user_command('Hash', hashCurrentBuffer,
     {})
```

KACTL template

bd2055
```
// in addition to template.h, kactl uses:
#define rep(i,a,b) for(int i = a; i < (b); ++i)
#define sz(x) (int)(x).size()
typedef pair<int,int> pii;
typedef vector<int> vi;
```

template

9aea93
```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define all(x) (x).begin(), (x).end()

int main() {
  ios::sync_with_stdio(0); cin.tie(0);
}
```

Data_structures

Disjoint Set Union

Description: Classic DSU using path compression and union by rank. `unite` returns true iff `u` and `v` were disjoint.
Usage: `Dsu d(n); d.unite(a, b); d.find(a);`
Complexity: `find()`, `unite()` are amortized $\mathcal{O}(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. Basically $\mathcal{O}(1)$.

5168ab
```
struct Dsu {
  vector <int> p, rank;
  Dsu(int n) {
    p.resize(n); rank.resize(n, 0);
    iota(p.begin(), p.end(), 0);
  }
  int find(int x) {
    return p[x] == x ? x : p[x] = find(p[x]);
  }
  bool unite(int u, int v) {
    if ((u = find(u)) == (v = find(v))) return false;
    if (rank[u] < rank[v]) swap(u, v);
    p[v] = u;
    rank[u] += rank[u] == rank[v];
    return true;
  }
};
```

Li-Chao tree

Description: Contianer of lines, online insertion/querying. Retrieve the line f with minimum $f(x)$ for a given x.
Usage: `LCT lct(n); lct.insert(line, 0, n); lct.query(x, 0, n);`
Complexity: $\mathcal{O}(\log n)$ per insertion/query

ba9fe6
```
struct Line { ll a, b; ll f(ll x) { return a * x + b; } };
constexpr const Line LINF { 0, 1LL << 60 };
struct LCT {
  vector <Line> v; // coord-compression: modify v[x] -> v[
     conert(x)]
  LCT(int size) { v.resize(size + 1, LINF); }
  void insert(Line line, int l, int r) {
    if (l > r) return;
    int mid = (l + r) >> 1;
    if (line.f(mid) < v[mid].f(mid)) swap(line, v[mid]);
    if (line.f(l) < v[mid].f(l)) insert(line, l, mid - 1);
    else insert(line, mid + 1, r);
  }
  Line query(int x, int l, int r) {
    if (l > r) return LINF;
    int mid = (l + r) >> 1;
    if (x == mid) return v[mid]; // faster on avg. - not
     necessary
    if (x < mid) return best_of(v[mid], query(x, l, mid -
     1), x);
    return best_of(v[mid], query(x, mid + 1, r), x);
  }
  Line best_of(Line a, Line b, ll x) { return a.f(x) < b.f(
     x) ? a : b; }
};
```

Rollback Union Find

Description: Yoinked from kactl. Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().
Usage: `int t = uf.time(); ...; uf.rollback(t);`
Complexity: $\mathcal{O}(\log n)$.

de4ad0
```
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;
  }
};
```

Fenwick tree

Description: Computes prefix sums and single element updates. Uses 0-indexing.
Usage: `Fen f(n); f.update(ind, val); f.query(ind); f.lower_bound(sum);`
Complexity: $\mathcal{O}(\log n)$ per update/query

1743e1
```
struct Fen {
  vector <ll> v;
  Fen(int s) : v(s, 0) { }
  void update(int ind, ll val) {
    for (; ind < (int) v.size(); ind |= ind + 1) v[ind] +=
     val;
  }
  ll query(int ind) { // [0, ind), ind < 0 returns 0
```

```
    ll res = 0;
    for (; ind > 0; ind &= ind - 1) res += v[ind - 1]; //
     operation can be modified
    return res;
  }
  int lower_bound(ll sum) { // returns first i with query(i
     + 1) >= sum, n if not found
    int ind = 0;
    for (int p = 1 << 25; p; p >>= 1) // 1 << 25 can be
     lowered to ceil(log2(v.size()))
      if (ind + p <= (int) v.size() && v[ind + p - 1] < sum
     )
        sum -= v[(ind += p) - 1];
    return ind;
  }
};
```

Fast hash map

Description: 3x faster hash map, 1.5x more memory usage, similar API to std::unordered_map. Initial capacity, if provided, must be power of 2.
Usage: `hash_map <key_t, val_t> mp; mp[key] = val; mp.find(key); mp.begin(); mp.end(); mp.erase(key); mp.size();`
Complexity: $\mathcal{O}(1)$ per operation on average.

bf4708

```
#include <bits/extc++.h>

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

template <typename KEY_T, typename VAL_T> using hash_map =
    __gnu_pbds::gp_hash_table <KEY_T, VAL_T, chash>;
```

Implicit 2D segment tree

Description: Classic implicit 2D segment tree taken from my solution to IOI game 2013. It is in rough shape, but it works. Designed to be `[inclusive, exclusive)`. It is old and looks shady, only rely slightly on it, maybe even just make a new one if you need one.
Usage: See usage example at the bottom.
Complexity: $\mathcal{O}(\log^2 n)$ per operation *I think*.

ae92aa

```
constexpr const int MX_RC = 1 << 30;

struct Inner {
  long long val;
  int lv, rv;
  Inner* lc,* rc;
  Inner(long long _val, int _l, int _r) :
  val(_val), lv(_l), rv(_r), lc(nullptr), rc(nullptr)
  { }
  ~Inner() {
    delete(lc);
    delete(rc);
  }
  void update(int ind, long long nev, int l = 0, int r =
    MX_RC) {
    if (!(r - l - 1)) {
      assert(lv == l && rv == r);
      assert(ind == l);
      val = nev;
      return;
    }
    int mid = (l + r) >> 1;
    if (ind < mid) {
      if (lc) {
        if (lc->lv != l || lc->rv != mid) {
          Inner* tmp = lc;
          lc = new Inner(0, l, mid);
          (tmp->lv < ((l + mid) >> 1) ? lc->lc : lc->rc) =
     tmp;
        }
        lc->update(ind, nev, l, mid);
      } else lc = new Inner(nev, ind, ind + 1);
    } else {
      if (rc) {
        if (rc->lv != mid || rc->rv != r) {
          Inner* tmp = rc;
          rc = new Inner(0, mid, r);
          (tmp->lv < ((mid + r) >> 1) ? rc->lc : rc->rc) =
     tmp;
        }
        rc->update(ind, nev, mid, r);
      } else rc = new Inner(nev, ind, ind + 1);
    }
    val = std::gcd(lc ? lc->val : 0, rc ? rc->val : 0);
  }
  long long query(int tl, int tr, int l = 0, int r = MX_RC)
     {
    if (l >= tr || r <= tl) return 0;
    if (!(rv - lv - 1)) {
      if (lv >= tr || rv <= tl) return 0;
      return val;
    }
    assert(l == lv && r == rv);
    if (l >= tl && r <= tr) return val;
    int mid = (l + r) >> 1;
    return std::gcd(lc ? lc->query(tl, tr, l, mid) : 0, rc
     ? rc->query(tl, tr, mid, r) : 0);
  }
  void fill(Inner* source) {
    val = source->val;
    if (!(lv - rv - 1)) return;
    if (source->lc) {
      lc = new Inner(source->lc->val, source->lc->lv,
     source->lc->rv);
      lc->fill(source->lc);
    }
    if (source->rc) {
      rc = new Inner(source->rc->val, source->rc->lv,
     source->rc->rv);
      rc->fill(source->rc);
    }
  }
};

struct Outer {
  Inner* inner;
  int lv, rv;
  Outer* lc,* rc;
  Outer(Inner* _inner, int _l, int _r) :
  inner(_inner), lv(_l), rv(_r), lc(nullptr), rc(nullptr)
  { }
  void update(int ind_outer, int ind_inner, long long nev,
     int l = 0, int r = MX_RC) {
    if (!(r - l - 1)) {
      assert(lv == l && rv == r);
      assert(ind_outer == l);
      assert(inner);
      inner->update(ind_inner, nev);
      return;
    }
    int mid = (l + r) >> 1;
    if (ind_outer < mid) {
      if (lc) {
        if (lc->lv != l || lc->rv != mid) {
          Outer* tmp = lc;
          lc = new Outer(new Inner(0, 0, MX_RC), l, mid);
          lc->inner->fill(tmp->inner);
          (tmp->lv < ((l + mid) >> 1) ? lc->lc : lc->rc) =
     tmp;
        }
        lc->update(ind_outer, ind_inner, nev, l, mid);
      } else {
        lc = new Outer(new Inner(0, 0, MX_RC), ind_outer,
     ind_outer + 1);
        lc->inner->update(ind_inner, nev);
      }
    } else {
      if (rc) {
        if (rc->lv != mid || rc->rv != r) {
          Outer* tmp = rc;
          rc = new Outer(new Inner(0, 0, MX_RC), mid, r);
          rc->inner->fill(tmp->inner);
          (tmp->lv < ((mid + r) >> 1) ? rc->lc : rc->rc) =
     tmp;
        }
        rc->update(ind_outer, ind_inner, nev, mid, r);
      } else {
        rc = new Outer(new Inner(nev, 0, MX_RC), ind_outer,
     ind_outer + 1);
        rc->inner->update(ind_inner, nev);
      }
    }
    inner->update(ind_inner, std::gcd(
    lc ? lc->inner->query(ind_inner, ind_inner + 1) : 0,
    rc ? rc->inner->query(ind_inner, ind_inner + 1) : 0));
  }
  long long query(int tl_outer, int tr_outer, int tl_inner,
     int tr_inner, int l = 0, int r = MX_RC) {
    if (l >= tr_outer || r <= tl_outer) return 0;
    if (!(rv - lv - 1)) {
      if (lv >= tr_outer || rv <= tl_outer) return 0;
      return inner->query(tl_inner, tr_inner);
    }
    assert(l == lv && r == rv);
    if (l >= tl_outer && r <= tr_outer)
      return inner->query(tl_inner, tr_inner);
    int mid = (l + r) >> 1;
    return std::gcd(
    lc ? lc->query(tl_outer, tr_outer, tl_inner, tr_inner,
    l, mid) : 0,
    rc ? rc->query(tl_outer, tr_outer, tl_inner, tr_inner,
    mid, r) : 0);
  }
};

// this is how it has been used in the solution to IOI game
     2013
Outer root(new Inner(0, 0, MX_RC), 0, MX_RC);
void update(int r, int c, long long k) {
  root.update(r, c, k);
}
long long calculate(int r_l, int c_l, int r_r, int c_r) {
  return root.query(r_l, r_r + 1, c_l, c_r + 1);
}
```

Lazy segment tree

Description: Zero-indexed, bounds are [l, r), operations can be modified. $\mathcal{O}(\log n)$ find_first and the like can be implemented by checking bounds, then checking left tree, then right tree, recursively.
Usage: `Lazy_segtree seg(n); seg.update(l, r, val); seg.query(l, r);`
Complexity: $\mathcal{O}(\log n)$ per update/query

69cb07

```
struct Lazy_segtree {
  typedef ll T; // change type here
  typedef ll LAZY_T; // change type here
  static constexpr T unit = 0; // change unit here
```

```
  static constexpr LAZY_T lazy_unit = 0; // change lazy
     unit here
  T f(T l, T r) { return l + r; } // change operation here
  void push(int now, int l, int r) {
    if (w[now] == lazy_unit) return;
    v[now] += w[now] * (r - l); // operation can be
     modified
    if (r - l - 1)
      w[now * 2 + 1].first += w[now],
      w[now * 2 + 2].first += w[now];
    w[now] = lazy_unit;
  }
  int size;
  vector <T> v;
  vector <LAZY_T> w;
  Lazy_segtree(int s = 0) : size(s ? 1 << (32 -
     __builtin_clz(s)) : 0), v(size << 1, unit), w(size <<
     1, lazy_unit) { }
  template <typename U> void update(int l, int r, U val) {
     update(l, r, val, 0, 0, size); }
  T query(int l, int r) { return query(l, r, 0, 0, size); }
  template <typename U> void update(int tl, int tr, U val,
     int now, int l, int r) {
    push(now, l, r);
    if (l >= tr || r <= tl) return;
    if (l >= tl && r <= tr) {
      // this does not *have* to accumulate, push is called
      before this:
      w[now] += val; // operation can be modified
      push(now, l, r);
      return;
    }
    int mid = (l + r) >> 1;
    update(tl, tr, val, now * 2 + 1, l, mid);
    update(tl, tr, val, now * 2 + 2, mid, r);
    v[now] = f(v[now * 2 + 1], v[now * 2 + 2]);
  }
  T query(int tl, int tr, int now, int l, int r) {
    push(now, l, r);
    if (l >= tr || r <= tl) return unit;
    if (l >= tl && r <= tr) return v[now];
    int mid = (l + r) >> 1;
    return f(query(tl, tr, now * 2 + 1, l, mid), query(tl,
     tr, now * 2 + 2, mid, r));
  }
  template <typename U> void build(const vector <U>& a) {
    for (int i = 0; i < (int) a.size(); i++) v[size - 1 + i
     ] = a[i]; // operation can be modified
    for (int i = size - 2; i >= 0; i--) v[i] = f(v[i * 2 +
     1], v[i * 2 + 2]);
  }
};
```

Matrix

Description: Yoinked from kactl. 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 = (AÑ) * vec;`
Complexity: $\mathcal{O}(n^3)$ per multiplication, $\mathcal{O}(n^3 \log p)$ per exponentiation.

c43c7d

```
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 operator^(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;
  }
};
```

Ordered Map

Description: `extc++.h` order statistics tree. `find_by_order` returns an iterator to the kth element (0-indexed), `order_of_key` returns the index of the element (0-indexed), i.e. the number of elements less than the argument.
Usage: `ordered_set <int> s; s.insert(1); s.insert(2); *s.find_by_order(0) = 3; s.erase(3); s.order_of_key(2);`
Complexity: Everything is $\mathcal{O}(\log n)$.

225ff4

```
#include <bits/extc++.h>
// if judge does not have extc++.h, use:
// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T> using ordered_set = tree <T,
    null_type, less <T>, rb_tree_tag,
    tree_order_statistics_node_update>;
template <typename T, typename U> using ordered_map = tree
    <T, U, less <T>, rb_tree_tag,
    tree_order_statistics_node_update>;

// yeet from kactl:
void example() {
  ordered_set <int> t, t2; t.insert(8);
  auto it = t.insert(10).first;
  assert(it == t.lower_bound(9));
  assert(t.order_of_key(10) == 1);
  assert(t.order_of_key(11) == 2);
  assert(*t.find_by_order(0) == 8);
  t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

Segment tree

Description: Zero-indexed, bounds are [l, r), operations can be modified. $\mathcal{O}(\log n)$ find_first and the like can be implemented by checking bounds, then checking left tree, then right tree, recursively.
Usage: `Segtree seg(n); seg.update(ind, val); seg.query(l, r);`
Complexity: $\mathcal{O}(\log n)$ per update/query.

f40463

```
struct Segtree {
  typedef ll T; // change type here
  static constexpr T unit = 0; // change unit here
  T f(T l, T r) { return l + r; } // change operation here
  int size;
  vector <T> v;
  Segtree(int s = 0) : size(s ? 1 << (32 - __builtin_clz(s)
    ) : 0), v(size << 1, unit) { }
  void update(int ind, T val) { update(ind, val, 0, 0, size
    ); }
  T query(int l, int r) { return query(l, r, 0, 0, size); }
  void update(int ind, T val, int now, int l, int r) {
    if (!(r - l - 1)) { v[now] = val; return; } //
     operation can be modified
    int mid = (l + r) >> 1;
    if (ind < mid) update(ind, val, now * 2 + 1, l, mid);
    else update(ind, val, now * 2 + 2, mid, r);
    v[now] = f(v[now * 2 + 1], v[now * 2 + 2]);
  }
  T query(int tl, int tr, int now, int l, int r) {
    if (l >= tr || r <= tl) return unit;
    if (l >= tl && r <= tr) return v[now];
    int mid = (l + r) >> 1;
    return f(query(tl, tr, now * 2 + 1, l, mid), query(tl,
     tr, now * 2 + 2, mid, r));
  }
  template <typename U> void build(const vector <U>& a) {
    for (int i = 0; i < (int) a.size(); i++) v[size - 1 + i
     ] = a[i]; // operation can be modified
    for (int i = size - 2; i >= 0; i--) v[i] = f(v[i * 2 +
     1], v[i * 2 + 2]);
  }
};
```

Sparse table

Description: Yoinked from kactl. Classic sparse table, implemented with range minimum queries, can be modified.
Usage: `Sparse s(vec); s.query(a, b);`
Complexity: $\mathcal{O}(|V| \log |V| + Q)$.

5b1135

```
template<class T> struct Sparse {
  vector<vector<T>> jmp;
  Sparse(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) { // interval [a, 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)]);
  }
};
```

Treap

Description: Yoinked from kactl. A short self-balancing tree. It acts as a sequential container with log-time splits/joins, and is easy to augment with additional data.
Complexity: $\mathcal{O}(\log n)$ operations.

9556fc

```
struct Node {
  Node *l = 0, *r = 0;
  int val, y, c = 1;
  Node(int val) : val(val), y(rand()) {}
  void recalc();
};

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

template<class F> void each(Node* n, F f) {
  if (n) { each(n->l, f); f(n->val); each(n->r, f); }
}

pair<Node*, Node*> split(Node* n, int k) {
  if (!n) return {};
  if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
    auto pa = split(n->l, k);
    n->l = pa.second;
    n->recalc();
```

```
    return {pa.first, n};
  } else {
    auto pa = split(n->r, k - cnt(n->l) - 1); // and just "
     k"
    n->r = pa.first;
    n->recalc();
    return {n, pa.second};
  }
}

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

Node* ins(Node* t, Node* n, int pos) {
  auto pa = split(t, pos);
  return merge(merge(pa.first, n), pa.second);
}

// Example application: move the range [l, r) to index k
void move(Node*& t, int l, int r, int k) {
  Node *a, *b, *c;
  tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
  if (k <= l) t = merge(ins(a, b, k), c);
  else t = merge(a, ins(c, b, k - r));
}
```

Geometry

3D convex hull

Description: Yoinked from kactl. 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.
Complexity: $\mathcal{O}(n^2)$.

6e20a4

```
#include "Point_3D.h"

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;
};
```

Angle

Description: Yoinked from kactl. 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

```
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 * (ll)b.x) <
         make_tuple(b.t, b.half(), a.x * (ll)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
    )};
}
```

Circle circle intersection

Description: Yoinked from kactl. Computes the pair of points at which two circles intersect. Returns false in case of no intersection.
Complexity: $\mathcal{O}(1)$.

583f44

```
#include "Point.h"

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;
}
```

Circle line intersection

Description: Yoinked from kactl. 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>`.

a84598

```
#include "Point.h"

template<class P>
vector<P> 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 < 0) return {};
  if (h2 == 0) return {p};
  P h = ab.unit() * sqrt(h2);
  return {p - h, p + h};
}
```

Circle polygon intersection

Description: Yoinked from kactl. Returns the area of the intersection of a circle with a ccw polygon.
Complexity: $\mathcal{O}(n)$.

8d446d

```
#include "Point.h"

typedef Point<double> P;
#define arg(p, q) atan2(p.cross(q), p.dot(q))
double circlePoly(P c, double r, vector<P> ps) {
  auto tri = [&](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 <= 0) return arg(p, q) * r2;
    auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det
    ));
    if (t < 0 || 1 <= 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;
}
```

Circle tangents

Description: Yoinked from kactl. 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.

15aa0d

```
#include "Point.h"

template<class P>
vector<pair<P, P>> 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 < 0)  return {};
  vector<pair<P, P>> 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;
}
```

Circumcircle

Description: Yoinked from kactl. 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.

fca490

```
#include "Point.h"

typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A-C).dist()/
      abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C-A, c = B-A;
  return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
}
```

Closest pair of points

Description: Yoinked from kactl. Finds the closest pair of points.
Complexity: $\mathcal{O}(n \log n)$.

ac9ec9

```
#include "Point.h"

typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
  assert(sz(v) > 1);
  set<P> S;
  sort(all(v), [](P a, P b) { return a.y < b.y; });
  pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
  int j = 0;
  for (P p : v) {
    P d{1 + (ll)sqrt(ret.first), 0};
    while (v[j].y <= p.y - d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d), hi = S.upper_bound(p +
     d);
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).dist2(), {*lo, p}});
    S.insert(p);
  }
  return ret.second;
}
```

Convex hull

Description: Yoinked from kactl. Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.
Complexity: $\mathcal{O}(n \log n)$.

ec8c60

```
#include "Point.h"

typedef Point<ll> P;
vector<P> convexHull(vector<P> pts) {
  if (sz(pts) <= 1) return pts;
  sort(all(pts));
  vector<P> h(sz(pts)+1);
  int s = 0, t = 0;
  for (int it = 2; it--; s = --t, reverse(all(pts)))
    for (P p : pts) {
      while (t >= s + 2 && h[t-2].cross(h[t-1], p) <= 0) t
     --;
      h[t++] = p;
    }
  return {h.begin(), h.begin() + t - (t == 2 && h[0] == h
     [1])};
}
```

Delaunay triangulation

Description: Yoinked from kactl. Computes the Delaunay triangulation of a set of points. Each circumcircle contains none of the input points. If any three points are collinear or any four are on the same circle, behavior is undefined.
Complexity: $\mathcal{O}(n^2)$.

0da7c8

```
#include "Point.h"
#include "3d_hull.h"

template<class P, class F>
void delaunay(vector<P>& ps, F trifun) {
  if (sz(ps) == 3) { int d = (ps[0].cross(ps[1], ps[2]) <
     0);
    trifun(0,1+d,2-d); }
  vector<P3> p3;
  for (P p : ps) p3.emplace_back(p.x, p.y, p.dist2());
  if (sz(ps) > 3) for(auto t:hull3d(p3)) if ((p3[t.b]-p3[t.
     a]).
      cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
    trifun(t.a, t.c, t.b);
}
```

Dynamic Convex Hull

Description: Supports building a convex hull one point at a time. Viewing the convex hull along the way.

431bba

```
struct point {
    ll x, y;
    point(ll x=0, ll y=0): x(x), y(y) {}
    point operator-(const point &p) const { return point(x-
     p.x, y-p.y); }
    point operator*(const ll k) const { return point(k*x, k
     *y); }
    ll cross(const point &p) const { return x*p.y - p.x*y;
     }
    bool operator<(const point &p) const { return x < p.x
     || x == p.x && y < p.y; }
};

bool above(set<point> &hull, point p, ll scale = 1) {
    auto it = hull.lower_bound(point((p.x+scale-1)/scale,
     0));
    if (it == hull.end()) return true;
    if (p.y <= it->y*scale) return false;
    if (it == hull.begin()) return true;
    auto jt = it--;
    return (p-*it*scale).cross(*jt-*it) < 0;
}

void add(set<point> &hull, point p) {
    if (!above(hull, p)) return;
    auto pit = hull.insert(p).first;
    while (pit != hull.begin()) {
        auto it = prev(pit);
        if (it->y <= p.y || (it != hull.begin() && (*it-*
     prev(it)).cross(*pit-*it) >= 0))
            hull.erase(it);
        else
            break;
    }
    auto it = next(pit);
    while (it != hull.end()) {
        if (next(it) != hull.end() && (*it-p).cross(*next(
     it)-*it) >= 0)
            hull.erase(it++);
        else
            break;
    }
}
```

Hull diameter

Description: Yoinked from kactl. Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).
Complexity: $\mathcal{O}(n)$.

326872

```
#include "Point.h"

typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
  int n = sz(S), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
  rep(i,0,j)
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}})
     ;
      if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >=
     0)
        break;
    }
  return res.second;
}
```

Inside polygon

Description: Yoinked from kactl. 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);`
Complexity: $\mathcal{O}(n)$.

bbf6f5

```
#include "Point.h"
#include "On_segment.h"
```

```
#include "Segment_distance.h"

template<class P>
bool inPolygon(vector<P> &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) <= eps) return !strict;
    cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) >
     0;
  }
  return cnt;
}
```

KD-tree

Description: Yoinked from kactl. 2D, can be extended to 3D. See comments for details.

08fbca

```
#pragma once

#include "Point.h"

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;
  }

  // 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);
  }
};
```

Line hull intersection

Description: Yoinked from kactl. Line-convex polygon intersection. The polygon must be ccw and have no collinear points. `lineHull(line, poly)` returns a pair describing the intersection of a line with the polygon:

- $(-1,-1)$ if no collision,
- $(i,-1)$ if touching the corner i,
- (i,i) if along side $(i,i+1)$,
- (i,j) if crossing sides $(i,i+1)$ and $(j,j+1)$.

In the last case, if a corner i is crossed, this is treated as happening on side $(i,i+1)$. The points are returned in the same order as the line hits the polygon.

Complexity: $\mathcal{O}(\log n)$.

32bb13

```
#include "Point.h"

#define cmp(i,j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%
    n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
template <class P> int extrVertex(vector<P>& poly, P dir) {
  int n = sz(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if (extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
    (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) =
     m;
  }
  return lo;
}

#define cmpL(i) sgn(a.cross(poly[i], b))
template <class P>
array<int, 2> lineHull(P a, P b, vector<P>& poly) {
  int endA = extrVertex(poly, (a - b).perp());
  int endB = extrVertex(poly, (b - a).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<int, 2> res;
  rep(i,0,2) {
    int lo = endB, hi = endA, n = sz(poly);
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
      (cmpL(m) == cmpL(endB) ? lo : hi) = m;
    }
    res[i] = (lo + !cmpL(hi)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res[0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
      case 0: return {res[0], res[0]};
      case 2: return {res[1], res[1]};
    }
  return res;
}
```

Line line intersection

Description: Yoinked from kactl. 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;`

ae219b

```
#include "Point.h"

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};
}
```

Line projection and reflection

Description: Yoinked from kactl. 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.

c08b13

```
#include "Point.h"

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();
}
```

Linear transformation

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

a1c4fb

```
#include "Point.h"

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();
}
```

Manhatten MST

Description: Yoinked from kactl. Given N points, returns up to $4N$ edges, which are guaranteed to contain a minimum spanning tree for the graph with edge weights $w(p,q) = |p.x - q.x| + |p.y - q.y|$. Edges are in the form $(distance, src, dst)$. Use a standard MST algorithm on the result to find the final MST.

Complexity: $\mathcal{O}(n \log n)$.

c264e8
```
#include "Point.h"

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;
}
```

Minimum enclosing circle

Description: Yoinked from kactl. Computes the minimum circle that encloses a set of points.
Complexity: $\mathcal{O}(n)$.

0b9ff5
```
#include "circumcircle.h"

pair<P, double> mec(vector<P> 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() > r * EPS) {
    o = ps[i], r = 0;
    rep(j,0,i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      rep(k,0,j) if ((o - ps[k]).dist() > r * EPS) {
        o = ccCenter(ps[i], ps[j], ps[k]);
        r = (o - ps[i]).dist();
      }
    }
  }
  return {o, r};
}
```

Is on segment

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

5fde08
```
#include "Point.h"

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

2D Point

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

47ec0a
```
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 << ")"; }
};
```

3D Point

Description: Yoinked from kactl. Class to handle points in 3D space. T can be e.g. double or long long. (Avoid int.).

8058ae
```
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); }
  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P 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; }
  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;
  }
};
```

Is point in convex polygon

Description: Yoinked from kactl. Determine whether a point t lies inside a convex hull (CCW order, with no collinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.
Complexity: $\mathcal{O}(\log n)$.

78b7ca
```
#include "Point.h"
#include "Side_of.h"
#include "On_segment.h"

typedef Point<ll> P;

bool inHull(const vector<P>& l, P p, bool strict = true) {
  int a = 1, b = sz(l) - 1, r = !strict;
  if (sz(l) < 3) return r && onSegment(l[0], l.back(), p);
  if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
  if (sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p)<=
      -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r;
}
```

Polygon area

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

714de9
```
#include "Point.h"

template<class T>
T polygonArea2(vector<Point<T>>& v) {
  T a = v.back().cross(v[0]);
  rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
  return a;
}
```

Polygon center of mass

Description: Yoinked from kactl. Returns the center of mass for a polygon.
Complexity: $\mathcal{O}(n)$.

6906c1
```
#include "Point.h"

typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (int i = 0, j = sz(v) - 1; i < 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;
}
```

Polygon cut

Description: Yoinked from kactl. 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));`

076bef
```
#include "Point.h"
#include "Line_intersection.h"

typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  rep(i,0,sz(poly)) {
    P cur = poly[i], prev = i ? poly[i-1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0))
      res.push_back(lineInter(s, e, cur, prev).second);
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

Polygon union

Description: Yoinked from kactl. 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.)
Complexity: $\mathcal{O}(n^2)$ where n is the total number of points.

82e22a
```
#include "Point.h"
#include "Side_of.h"

typedef Point<double> P;
double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y;
    }
double polyUnion(vector<vector<P>>& 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<pair<double, int>> 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) < 0)
            segs.emplace_back(sa / (sa - sb), sgn(sc - sd))
    ;
        } else if (!sc && !sd && j<i && sgn((B-A).dot(D-C))
    >0){
          segs.emplace_back(rat(C - A, B - A), 1);
          segs.emplace_back(rat(D - A, B - A), -1);
        }
      }
    }
    sort(all(segs));
    for (auto& s : segs) s.first = min(max(s.first, 0.0),
    1.0);
    double sum = 0;
    int cnt = segs[0].second;
    rep(j,1,sz(segs)) {
      if (!cnt) sum += segs[j].first - segs[j - 1].first;
      cnt += segs[j].second;
    }
    ret += A.cross(B) * sum;
  }
  return ret / 2;
}
```

Polyhedron volume

Description: Yoinked from kactl. Magic formula for the volume of a polyhedron. Faces should point outwards.

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

Points line-segments distance

Description: Yoinked from kactl. 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;`

a3104f
```
#include "Point.h"

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;
}
```

Line segment line segment intersection

Description: Yoinked from kactl. 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;`

12fa1d
```
#include "Point.h"
#include "OnSegment.h"

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)};
}
```

Side of

Description: Yoinked from kactl. Returns where p is as seen from s towards e. 1/0/-1 $\Leftrightarrow$ 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;`

f459d8
```
#include "Point.h"

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);
}
```

Spherical distance

Description: Yoinked from kactl. Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_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. $dx \cdot radius$ is then the difference between the two points in the x direction and d*radius is the total distance between the points.

611f07
```
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);
}
```

Line distance

Description: Yoinked from kactl. 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.

d1d032
```
#include "Point.h"

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

Graphs

Articulation points finding

Description: Yoinked from CP-algorithms. Standard articulation points finding algorithm.
Complexity: $\mathcal{O}(V + E)$.

23f413
```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
  visited[v] = true;
  tin[v] = low[v] = timer++;
```

```
  int children=0;
  for (int to : adj[v]) {
    if (to == p) continue;
    if (visited[to]) {
      low[v] = min(low[v], tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v], low[to]);
      if (low[to] >= tin[v] && p!=-1)
        IS_CUTPOINT(v);
      ++children;
    }
  }
  if(p == -1 && children > 1)
    IS_CUTPOINT(v);
}

void find_cutpoints() {
  timer = 0;
  visited.assign(n, false);
  tin.assign(n, -1);
  low.assign(n, -1);
  for (int i = 0; i < n; ++i) {
    if (!visited[i])
      dfs (i);
  }
}
```

Bellman-Ford

Description: Yoinked from kactl. 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 $V^2 \max |w_i| < \sim 2^{63}$.
Usage: `bellmanFord(nodes, edges, s)`.
Complexity: $\mathcal{O}(VE)$.

```
830a8f
const 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;
  }
}
```

Biconnected components

Description: Yoinked from kactl. 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(b, eid); ed[b].emplace(a, eid++); } bicomps([&](const vi& edgelist) { ... });`
Complexity: $\mathcal{O}(E+V)$.

```
2965e5
vi num, st;
vector<vector<pii>> ed;
int Time;
template<class F>
int dfs(int at, int par, F& f) {
  int me = num[at] = ++Time, e, y, top = me;
  for (auto pa : ed[at]) if (pa.second != par) {
    tie(y, e) = pa;
    if (num[y]) {
      top = min(top, num[y]);
      if (num[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) {
  num.assign(sz(ed), 0);
  rep(i,0,sz(ed)) if (!num[i]) dfs(i, -1, f);
}
```

Binary lifting with LCA

Description: Yoinked from kactl. Finds power of two jumps in a tree - and standard LCA. Assumes the root node points to itself!
Usage: `vector<vi> jmps = treeJump(parents); int l = lca(jmps, depth, a, b);`
Complexity: $\mathcal{O}(N \log N)$ construction. $\mathcal{O}(\log N)$ per query.

```
bfce85
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];
}
```

Bridge finding

Description: Yoinked from CP-algorithms. Standard bridge finding algorithm.
Complexity: $\mathcal{O}(V+E)$.

```
a44485
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
  visited[v] = true;
  tin[v] = low[v] = timer++;
  for (int to : adj[v]) {
    if (to == p) continue;
    if (visited[to]) {
      low[v] = min(low[v], tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v], low[to]);
      if (low[to] > tin[v])
        IS_BRIDGE(v, to);
    }
  }
}

void find_bridges() {
  timer = 0;
  visited.assign(n, false);
  tin.assign(n, -1);
  low.assign(n, -1);
  for (int i = 0; i < n; ++i) {
    if (!visited[i])
      dfs(i);
  }
}
```

DFS Bipartite Matching

Description: Yoinked from kactl. 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);`
Complexity: $\mathcal{O}(VE)$.

```
522b98
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);
}
```

Dinic's Algorithm

Description: Yoinked from kactl. Finds the maximum flow from s to t in a directed graph. To obtain the actual flow values, look at all edges with capacity > 0 (zero capacity edges are residual edges).
Usage: `Dinic dinic(n); dinic.addEdge(a, b, c); dinic.maxFlow(s, t);`
Complexity: $\mathcal{O}(VE \log U)$ where $U = \max \mid \text{capacity} \mid$. $\mathcal{O}(\min(\sqrt{E}, V^{2/3})E)$ if $U = 1$; so $\mathcal{O}(\sqrt{V}E)$ for bipartite matching.

d7f0f1
```
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; }
};
```

MST in directed graphs

Description: Yoinked from kactl. Finds a minimum spanning tree/arborescence of a directed graph, given a root node. If no MST exists, returns -1.
Usage: `pair <ll, vi> res = DMST(n, edges, root);`
Complexity: $\mathcal{O}(E \log V)$.

de32e7
```
#include "../Data_structures/dsu_rollback.h"

struct Edge { int a, b; ll w; };
struct Node { /// lazy skew heap 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>>> cycs;
  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) { /// found cycle, contract
        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;
        cycs.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] : cycs) { // 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};
}
```

(D + 1)-edge coloring

Description: Yoinked from kactl. 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.)
Usage: `vi res = edgeColoring(N, eds);`
Complexity: $\mathcal{O}(NM)$.

e210e2
```
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;
}
```

Edmonds-Karp

Description: Yoinked from kactl. Flow algorithm with guaranteed complexity $\mathcal{O}(VE^2)$. To get edge flow values, compare capacities before and after, and take the positive values only.
Usage: `edmondsKarp(graph, source, sink);`
Complexity: $\mathcal{O}(EV^2)$.

482fe0
```
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;
```

```
    }
  }
}
```

Floyd-Warshall

Description: Yoinked from kactl. Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix m, where $m[i][j] =$ `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.
Usage: `floydWarshall(m);`
Complexity: $\mathcal{O}(n^3)$.

531245

```
const 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;
}
```

General matching

Description: Yoinked from kactl. Matching for general graphs. Finds a maximum subset of edges such that each vertex is incident to at most one edge. Fails with probability $\frac{N}{\text{mod}}$.
Usage: `generalMatching(N, ed)`
Complexity: $\mathcal{O}(N^3)$.

0d7362

```
#include "../Maths/Matrix_inverse_mod.h"

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 = rand() % 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 = rand() % 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;
}
```

Global minimum cut

Description: Yoinked from kactl. Finds a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Usage: `pair<int, vi> res = globalMinCut(mat);`
Complexity: $\mathcal{O}(V^3)$.

8b0e19

```
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;
}
```

Heavy-light decomposition

Description: Yoinked from kactl. 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. Code does additive modifications and max queries, but can support commutative segtree modifications/queries on paths and subtrees. Takes as input the full adjacency list. `VALS_EDGES` being true means that values are stored in the edges, as opposed to the nodes. All values initialized to the segtree default. Root must be 0. NOTE: below implementation uses kactl lazy segtree, this detail must be modified!
Usage: `HLD <false> hld(adj); hld.query_path(u, v); ...`
Complexity: $\mathcal{O}(\log n)$ segtree operations per operation.

1295ab

```
#include "...kactl_segtree..."

template <bool VALS_EDGES> struct HLD {
  int N, tim = 0;
  vector<vi> adj;
  vi par, siz, depth, rt, pos;
  Node *tree;
  HLD(vector<vi> adj_)
    : N(sz(adj_)), adj(adj_), par(N, -1), siz(N, 1), depth(
    N),
      rt(N),pos(N),tree(new Node(0, N)){ dfsSz(0); dfsHld
    (0); }
  void dfsSz(int v) {
    if (par[v] != -1) adj[v].erase(find(all(adj[v]), par[v
    ]));
    for (int& u : adj[v]) {
      par[u] = v, depth[u] = depth[v] + 1;
      dfsSz(u);
      siz[v] += siz[u];
      if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
    }
  }
  void dfsHld(int v) {
    pos[v] = tim++;
    for (int u : adj[v]) {
      rt[u] = (u == adj[v][0] ? rt[v] : u);
      dfsHld(u);
    }
  }
  template <class B> void process(int u, int v, B op) {
    for (; rt[u] != rt[v]; v = par[rt[v]]) {
      if (depth[rt[u]] > depth[rt[v]]) swap(u, v);
      op(pos[rt[v]], pos[v] + 1);
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(pos[u] + VALS_EDGES, pos[v] + 1);
  }
  void modifyPath(int u, int v, int val) {
    process(u, v, [&](int l, int r) { tree->add(l, r, val);
      });
  }
  int queryPath(int u, int v) { // Modify depending on
     problem
    int res = -1e9;
    process(u, v, [&](int l, int r) {
        res = max(res, tree->query(l, r));
    });
    return res;
  }
  int querySubtree(int v) { // modifySubtree is similar
    return tree->query(pos[v] + VALS_EDGES, pos[v] + siz[v
    ]);
  }
};
```

Hopcroft-Karp Bipartite Matching

Description: Yoinked from kactl. 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);`
Complexity: $\mathcal{O}(\sqrt{V}E)$.

f612e4

```
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);
    /// Find the starting nodes for BFS (i.e. layer 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);
    /// Find all layers using bfs.
```

```
    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);
    }
    /// Use DFS to scan for augmenting paths.
    rep(a,0,sz(g))
      res += dfs(a, 0, g, btoa, A, B);
  }
}
```

Link-cut tree

Description: Yoinked from kactl. 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.
Usage: See comments in code.
Complexity: Amortized $\mathcal{O}(\log n)$ per (any) operation.

5909e2

```
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];
      z->c[h ^ 1] = b ? x : this;
    }
    y->c[i ^ 1] = b ? this : x;
    fix(); x->fix(); y->fix();
    if (p) p->fix();
    swap(pp, y->pp);
  }
  void splay() { /// Splay this up to the root. Always
     finishes without flip set.
    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() { /// Return the min element of the subtree
     rooted at this, splayed to the top.
    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) { /// Move u to root of
     represented tree.
    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) { /// Move u to root aux tree.
     Return the root of the root aux tree.
    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;
  }
};
```

Maximum clique callbacks

Description: Yoinked from kactl. 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.
Usage: `cliques(eds, callback, ...);`
Complexity: $\mathcal{O}(3^{n/3})$ - *much* faster for sparse graphs.

b0d5b1

```
typedef bitset<128> B;
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;
  }
}
```

Maximum clique

Description: Yoinked from kactl. Finds a maximum clique of a graph given as a symmetric bitset matrix. Can be used to find a maximum independent set by finding a clique of the complement graph.
Complexity: About 1 second for $n = 155$, worst case random graphs ($p = .90$). Runs faster for sparse graphs.

f7c0bc

```
typedef vector<bitset<200>> vb;
struct Maxclique {
  double limit=0.025, pk=0;
  struct Vertex { int i, d=0; };
  typedef vector<Vertex> vv;
  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});
  }
};
```

Minimum cost maximum flow

Description: Yoinked from kactl. `cap[i][j] != cap[j][i]` is allowed; double edges are not. 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.
Complexity: $\mathcal{O}(E^2)$ `o_0`.

fe85cc
```
#include <bits/extc++.h>

const ll INF = numeric_limits<ll>::max() / 4;
typedef vector<ll> VL;

struct MCMF {
  int N;
  vector<vi> ed, red;
  vector<VL> cap, flow, cost;
  vi seen;
  VL dist, pi;
  vector<pii> par;

  MCMF(int N) :
    N(N), ed(N), red(N), cap(N, VL(N)), flow(cap), cost(cap
    ),
    seen(N), dist(N), pi(N), par(N) {}

  void addEdge(int from, int to, ll cap, ll cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
    ed[from].push_back(to);
    red[to].push_back(from);
  }

  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});

    auto relax = [&](int i, ll cap, ll cost, int dir) {
      ll val = di - pi[i] + cost;
      if (cap && val < dist[i]) {
        dist[i] = val;
        par[i] = {s, dir};
        if (its[i] == q.end()) its[i] = q.push({-dist[i], i
    });
        else q.modify(its[i], {-dist[i], i});
      }
    };

    while (!q.empty()) {
      s = q.top().second; q.pop();
      seen[s] = 1; di = dist[s] + pi[s];
      for (int i : ed[s]) if (!seen[i])
        relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
      for (int i : red[s]) if (!seen[i])
        relax(i, flow[i][s], -cost[i][s], 0);
    }
    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 (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
        fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p
    ]);
      totflow += fl;
      for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
        if (r) flow[p][x] += fl;
        else flow[x][p] -= fl;
    }
    rep(i,0,N) rep(j,0,N) totcost += cost[i][j] * flow[i][j
    ];
    return {totflow, totcost};
  }

  // 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 (int to : ed[i]) if (cap[i][to])
          if ((v = pi[i] + cost[i][to]) < pi[to])
            pi[to] = v, ch = 1;
    assert(it >= 0); // negative cost cycle
  }
};
```

Minimum vertex cover

Description: Yoinked from kactl. 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.
Complexity: Idk, look code.

b09e48
```
#include "DFS_matching.h"

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 q, 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;
}
```

Strongly connected components

Description: Yoinked from kactl. 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.
Complexity: $\mathcal{O}(E + V)$.

76b5c9
```
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);
}
```

Topological sort

Description: Yoinked from kactl. Given is an oriented graph. Output is an ordering of vertices, such that there are edges only from left to right. If there are cycles, the returned list will have size smaller than n – nodes reachable from cycles will not be returned.
Usage: `vi res = topoSort(gr);`
Complexity: $\mathcal{O}(V + E)$.

66a137
```
vi topoSort(const vector<vi>& gr) {
  vi indeg(sz(gr)), ret;
  for (auto& li : gr) for (int x : li) indeg[x]++;
  queue<int> q; // use priority_queue for lexic. largest
    ans.
  rep(i,0,sz(gr)) if (indeg[i] == 0) q.push(i);
  while (!q.empty()) {
    int i = q.front(); // top() for priority queue
    ret.push_back(i);
    q.pop();
    for (int x : gr[i])
      if (--indeg[x] == 0) q.push(x);
  }
  return ret;
}
```

Weighted bipartite matching

Description: Yoinked from kactl. 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 cost, match), where `L[i]` is matched with `R[match[i]]`. Negate costs for max cost. Requires $N \leq M$.
Complexity: $\mathcal{O}(N^2M)$.

1e0fe9
```
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
}
```

Two SAT

Description: Yoinked from kactl. Solves 2-SAT.
Usage: `TwoSat ts(n)` where n is the number of variables. `ts.either(i,j)` means that either i or j must be true. `ts.setValue(i)` means that i must be true. `ts.atMostOne(l)` means that at most one of the variables in l can be true. `ts.solve()` returns true iff it is solvable. `ts.values` will contain one possible solution. Negated variables are represented by bit-inversions (`~x`).
Complexity: $\mathcal{O}(N+E)$ where N is the number of variables and E is the number of clauses.

5f9706

```
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;
  }
};
```

Maths

Chinese remainder theorem

Description: Yoinked from kactl. `crt(a, m, b, n)` computes x such that $x \equiv a \pmod m$, $x \equiv b \pmod n$. If $|a| < m$ and $|b| < n$, x will obey $0 \le x < \text{lcm}(m,n)$. Assumes $mn < 2^{62}$.
Complexity: $\mathcal{O}(\log n)$.

9a1b35

```
#include "Euclid.h"

ll crt(ll a, ll m, ll b, ll n) {
  if (n > m) swap(a, b), swap(m, n);
  ll x, y, g = euclid(m, n, x, y);
  assert((a - b) % g == 0); // else no solution
  x = (b - a) % n * x % n / g * m + a;
  return x < 0 ? x + m*n/g : x;
}
```

Continued fractions

Description: Yoinked from kactl. Given N and a real number $x \ge 0$, finds the closest rational approximation p/q with $p, q \le N$. It will obey $|p/q - x| \le 1/qN$. For consecutive convergents, $p_{k+1}q_k - q_{k+1}p_k = (-1)^k$. ($p_k/q_k$ alternates between $> x$ and $< x$.) If x is rational, y eventually becomes ∞; if x is the root of a degree 2 polynomial then a's eventually become cyclic.
Complexity: $\mathcal{O}(\log n)$.

dd6c5e

```
typedef double d; // for N ~ 1e7; long double for N ~ 1e9
pair<ll, ll> approximate(d x, ll N) {
  ll LP = 0, LQ = 1, P = 1, Q = 0, inf = LLONG_MAX; d y = x
    ;
  for (;;) {
    ll lim = min(P ? (N-LP) / P : inf, Q ? (N-LQ) / Q : inf
    ),
       a = (ll)floor(y), b = min(a, lim),
       NP = b*P + LP, NQ = b*Q + LQ;
    if (a > b) {
      // If b > a/2, we have a semi-convergent that gives
     us a
      // better approximation; if b = a/2, we *may* have
     one.
      // Return {P, Q} here for a more canonical
     approximation.
      return (abs(x - (d)NP / (d)NQ) < abs(x - (d)P / (d)Q)
    ) ?
        make_pair(NP, NQ) : make_pair(P, Q);
    }
    if (abs(y = 1/(y - (d)a)) > 3*N) {
      return {NP, NQ};
    }
    LP = P; P = NP;
    LQ = Q; Q = NQ;
  }
}
```

Determinant

Description: Yoinked from kactl. Calculates determinant of a matrix. Destroys the matrix.
Complexity: $\mathcal{O}(N^3)$.

bd5cec

```
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;
}
```

Divisor Count

Description: Counts number of divisors

2ff470

```
ll divisor_cnt(ll n) {
    ll cnt = 1;
    map<ll, ll> factors = factorize(n);
    for (auto p : factors) cnt *= p.second+1;
    return cnt;
}
```

Sieve of Eratosthenes

Description: Yoinked from kactl. Prime sieve for generating all primes up to a certain limit. `isprime[i]` is true iff i is a prime.
Complexity: $lim = 100'000'000 \approx 0.8$ s. Runs 30% faster if only odd indices are stored.

7c144c

```
const int MAX_PR = 5'000'000;
bitset<MAX_PR> isprime;
vi eratosthenesSieve(int lim) {
  isprime.set(); isprime[0] = isprime[1] = 0;
  for (int i = 4; i < lim; i += 2) isprime[i] = 0;
  for (int i = 3; i*i < lim; i += 2) if (isprime[i])
    for (int j = i*i; j < lim; j += i*2) isprime[j] = 0;
  vi pr;
  rep(i,2,lim) if (isprime[i]) pr.push_back(i);
  return pr;
}
```

Euclid

Description: Yoinked from kactl. Finds two integers x and y, such that $ax + by = \gcd(a,b)$. If a and b are coprime, then x is the inverse of $a \pmod b$.
Complexity: $\mathcal{O}(\log n)$.

33ba8f

```
ll euclid(ll a, ll b, ll &x, ll &y) {
  if (!b) return x = 1, y = 0, a;
  ll d = euclid(b, a % b, y, x);
  return y -= a/b * x, d;
}
```

Fast fourier transform

Description: Yoinked from kactl. `fft(a)` computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. N must be a power of 2. Useful for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, divide by n, reverse(start+1, end), FFT back. Rounding is safe if $(\sum a_i^2 + \sum b_i^2)\log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs). Otherwise, use NTT/FFTMod.
Complexity: $\mathcal{O}(n \log n)$ with $N = |A| + |B|$. ($\sim 1s$ for $N = 2^{22}$)

3dd197

```
typedef complex<double> C;
typedef vector<double> vd;
```

```cpp
void fft(vector<C>& a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster if double)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
  }
  vi rev(n);
  rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
      // C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-
     rolled)  /// include-line
      auto x = (double *)&rt[j+k], y = (double *)&a[i+j+k];
             /// exclude-line
      C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x[1]*y[0]);
             /// exclude-line
      a[i + j + k] = a[i + j] - z;
      a[i + j] += z;
    }
}
vd conv(const vd& a, const vd& b) {
  if (a.empty() || b.empty()) return {};
  vd res(sz(a) + sz(b) - 1);
  int L = 32 - __builtin_clz(sz(res)), n = 1 << L;
  vector<C> in(n), out(n);
  copy(all(a), begin(in));
  rep(i,0,sz(b)) in[i].imag(b[i]);
  fft(in);
  for (C& x : in) x *= x;
  rep(i,0,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  rep(i,0,sz(res)) res[i] = imag(out[i]) / (4 * n);
  return res;
}
```

Fast fourier transform under arbitrary MOD

Description: Yoinked from kactl. 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 $[0, \text{mod})$.
Complexity: $\mathcal{O}(n \log n)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT).

499494

```cpp
#include "FFT.h"

typedef vector<ll> vl;
template<int M> vl convMod(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  vl res(sz(a) + sz(b) - 1);
  int B=32-__builtin_clz(sz(res)), n=1<<B, cut=int(sqrt(M))
    ;
  vector<C> L(n), R(n), outs(n), outl(n);
  rep(i,0,sz(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut)
    ;
  rep(i,0,sz(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut)
    ;
  fft(L), fft(R);
  rep(i,0,n) {
    int j = -i & (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) / 1i;
  }
  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;
  }
  return res;
}
```

Fast sieve of Eratosthenes

Description: Yoinked from kactl. Prime sieve for generating all primes smaller than LIM.
Complexity: LIM$= 1e9 \approx 1.5$s. Utalizes cache locality.

6b2912

```cpp
const int LIM = 1e6;
bitset<LIM> isPrime;
vi eratosthenes() {
  const int S = (int)round(sqrt(LIM)), R = LIM / 2;
  vi pr = {2}, sieve(S+1); pr.reserve(int(LIM/log(LIM)*1.1)
    );
  vector<pii> cp;
  for (int i = 3; i <= S; i += 2) if (!sieve[i]) {
    cp.push_back({i, i * i / 2});
    for (int j = i * i; j <= S; j += 2 * i) sieve[j] = 1;
  }
  for (int L = 1; L <= R; L += S) {
    array<bool, S> block{};
    for (auto &[p, idx] : cp)
      for (int i=idx; i < S+L; idx = (i+=p)) block[i-L] =
     1;
    rep(i,0,min(S, R - L))
      if (!block[i]) pr.push_back((L + i) * 2 + 1);
  }
  for (int i : pr) isPrime[i] = 1;
  return pr;
}
```

Gauss-Jordan elimination

Description: Yoinked from CP-algorithms. The description is taken from CP-algorithms as well: Following is an implementation of Gauss-Jordan. Choosing the pivot row is done with heuristic: choosing maximum value in the current column. The input to the function gauss is the system matrix a. The last column of this matrix is vector b. The function returns the number of solutions of the system $(0, 1, \text{or } \infty)$. If at least one solution exists, then it is returned in the vector ans. Implementation notes:

- The function uses two pointers - the current column col and the current row row.
- For each variable x_i, the value $where(i)$ is the line where this column is not zero. This vector is needed because some variables can be independent.
- In this implementation, the current i th line is not divided by a_{ii} as described above, so in the end the matrix is not identity matrix (though apparently dividing the i th line can help reducing errors).
- After finding a solution, it is inserted back into the matrix - to check whether the system has at least one solution or not. If the test solution is successful, then the function returns 1 or inf, depending on whether there is at least one independent variable.

kactl also has code for solving linear systems somewhere in the document, if needed.
Complexity: $\mathcal{O}(\min(n, m) \cdot nm)$ – I.e. cubic.

d847fe

```cpp
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be
    infinity or a big number

int gauss (vector < vector<double> > a, vector<double> &
    ans) {
  int n = (int) a.size();
  int m = (int) a[0].size() - 1;

  vector<int> where (m, -1);
  for (int col=0, row=0; col<m && row<n; ++col) {
    int sel = row;
    for (int i=row; i<n; ++i)
      if (abs (a[i][col]) > abs (a[sel][col]))
        sel = i;
    if (abs (a[sel][col]) < EPS)
      continue;
    for (int i=col; i<=m; ++i)
      swap (a[sel][i], a[row][i]);
    where[col] = row;

    for (int i=0; i<n; ++i)
      if (i != row) {
        double c = a[i][col] / a[row][col];
        for (int j=col; j<=m; ++j)
          a[i][j] -= a[row][j] * c;
      }
    ++row;
  }

  ans.assign (m, 0);
  for (int i=0; i<m; ++i)
    if (where[i] != -1)
      ans[i] = a[where[i]][m] / a[where[i]][i];
  for (int i=0; i<n; ++i) {
    double sum = 0;
    for (int j=0; j<m; ++j)
      sum += ans[j] * a[i][j];
    if (abs (sum - a[i][m]) > EPS)
      return 0;
  }

  for (int i=0; i<m; ++i)
    if (where[i] == -1)
      return INF;
  return 1;
}
```

Integer determinant

Description: Yoinked from kactl. Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.
Complexity: $\mathcal{O}(n^3)$.

3313dc

```cpp
const 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;
}
```

Integration

Description: Yoinked from kactl. 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.
Complexity: $\mathcal{O}(n)$ evaluations of f.

4756fc
```
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(i,1,n*2)
    v += f(a + i*h) * (i&1 ? 4 : 2);
  return v * h / 3;
}
```

Linear Diophantine Equation

Description: See below

002852
```
/*
A Linear Diophantine Equation (in two variables) is an
    equation of the general form:

$$ax + by = c$$

where $a$, $b$, $c$ are given integers, and $x$, $y$ are
    unknown integers.

## The degenerate case

A degenerate case that need to be taken care of is when $a
    = b = 0$. It is easy to see that we either have no
    solutions or infinitely many solutions, depending on
    whether $c = 0$ or not. In the rest of this article, we
     will ignore this case.

## Analytic solution

When $a \neq 0$ and $b \neq 0$, the equation $ax+by=c$ can
    be equivalently treated as either of the following:

\begin{gather}
ax \equiv c \pmod b,\newline
by \equiv c \pmod a.
\end{gather}

Without loss of generality, assume that $b \neq 0$ and
    consider the first equation. When $a$ and $b$ are co-
    prime, the solution to it is given as

$$x \equiv ca^{-1} \pmod b,$$

where $a^{-1}$ is the [modular inverse](module-inverse.md)
    of $a$ modulo $b$.

When $a$ and $b$ are not co-prime, values of $ax$ modulo
    $b$ for all integer $x$ are divisible by $g=\gcd(a, b)$
    , so the solution only exists when $c$ is divisible by
    $g$. In this case, one of solutions can be found by
    reducing the equation by $g$:

$$(a/g) x \equiv (c/g) \pmod{b/g}.$$

By the definition of $g$, the numbers $a/g$ and $b/g$ are
    co-prime, so the solution is given explicitly as

$$\begin{cases}
x \equiv (c/g)(a/g)^{-1}\pmod{b/g},\\
y = \frac{c-ax}{b}.
\end{cases}$$

## Algorithmic solution

To find one solution of the Diophantine equation with 2
    unknowns, you can use the [Extended Euclidean algorithm
    ](extended-euclid-algorithm.md). First, assume that $a$
     and $b$ are non-negative. When we apply Extended
    Euclidean algorithm for $a$ and $b$, we can find their
    greatest common divisor $g$ and 2 numbers $x_g$ and
    $y_g$ such that:

$$a x_g + b y_g = g$$

If $c$ is divisible by $g = \gcd(a, b)$, then the given
    Diophantine equation has a solution, otherwise it does
    not have any solution. The proof is straight-forward: a
     linear combination of two numbers is divisible by
    their common divisor.

Now supposed that $c$ is divisible by $g$, then we have:

$$a \cdot x_g \cdot \frac{c}{g} + b \cdot y_g \cdot \frac{c
    }{g} = c$$

Therefore one of the solutions of the Diophantine equation
    is:

$$x_0 = x_g \cdot \frac{c}{g},$$

$$y_0 = y_g \cdot \frac{c}{g}.$$

The above idea still works when $a$ or $b$ or both of them
    are negative. We only need to change the sign of $x_0$
    and $y_0$ when necessary.

Finally, we can implement this idea as follows (note that
    this code does not consider the case $a = b = 0$):
*/
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

bool find_any_solution(int a, int b, int c, int &x0, int &
    y0, int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
```
/*
Getting all solutions

From one solution (x_0, y_0), we can obtain all the
    solutions of the given equation.

Let $g = \gcd(a, b)$ and let x_0, y_0 be integers which
    satisfy the following:

$$a \cdot x_0 + b \cdot y_0 = c$$

Now, we should see that adding b / g to x_0, and, at
    the same time subtracting a / g from y_0 will not
    break the equality:

$$a \cdot \left(x_0 + \frac{b}{g}\right) + b \cdot \left(
    y_0 - \frac{a}{g}\right) = a \cdot x_0 + b \cdot y_0 +
    a \cdot \frac{b}{g} - b \cdot \frac{a}{g} = c$$

Obviously, this process can be repeated again, so all the
    numbers of the form:

$$x = x_0 + k \cdot \frac{b}{g}$$

$$y = y_0 - k \cdot \frac{a}{g}$$

are solutions of the given Diophantine equation.

Moreover, this is the set of all possible solutions of the
    given Diophantine equation.

Finding the number of solutions and the solutions in a
    given interval

From previous section, it should be clear that if we don't
    impose any restrictions on the solutions, there would
    be infinite number of them. So in this section, we add
    some restrictions on the interval of x and y, and
    we will try to count and enumerate all the solutions.

Let there be two intervals: $[min_x; max_x]$ and $[min_y;
    max_y]$ and let's say we only want to find the
    solutions in these two intervals.

Note that if a or b is 0, then the problem only has
    one solution. We don't consider this case here.

First, we can find a solution which have minimum value of
    x, such that $x \ge min_x$. To do this, we first find
     any solution of the Diophantine equation. Then, we
    shift this solution to get $x \ge min_x$ (using what we
     know about the set of all solutions in previous
    section). This can be done in $O(1)$.
Denote this minimum value of x by l_{x1}.

Similarly, we can find the maximum value of x which
    satisfy $x \le max_x$. Denote this maximum value of x
     by r_{x1}.

Similarly, we can find the minimum value of y $(y \ge
    min_y)$ and maximum values of y $(y \le max_y)$.
    Denote the corresponding values of x by l_{x2} and
    r_{x2}.

The final solution is all solutions with x in intersection
    of $[l_{x1}, r_{x1}]$ and $[l_{x2}, r_{x2}]$. Let
    denote this intersection by $[l_x, r_x]$.

Following is the code implementing this idea.
Notice that we divide a and b at the beginning by g.
Since the equation $a x + b y = c$ is equivalent to the
    equation $\frac{a}{g} x + \frac{b}{g} y = \frac{c}{g}$,
     we can use this one instead and have $\gcd(\frac{a}{g
    }, \frac{b}{g}) = 1$, which simplifies the formulas.
*/
void shift_solution(int & x, int & y, int a, int b, int cnt
) {
    x += cnt * b;
    y -= cnt * a;
}

int find_all_solutions(int a, int b, int c, int minx, int
    maxx, int miny, int maxy) {
    int x, y, g;
    if (!find_any_solution(a, b, c, x, y, g))
        return 0;
    a /= g;
    b /= g;

    int sign_a = a > 0 ? +1 : -1;
    int sign_b = b > 0 ? +1 : -1;

    shift_solution(x, y, a, b, (minx - x) / b);
    if (x < minx)
        shift_solution(x, y, a, b, sign_b);
```
```

```
    if (x > maxx)
        return 0;
    int lx1 = x;

    shift_solution(x, y, a, b, (maxx - x) / b);
    if (x > maxx)
        shift_solution(x, y, a, b, -sign_b);
    int rx1 = x;

    shift_solution(x, y, a, b, -(miny - y) / a);
    if (y < miny)
        shift_solution(x, y, a, b, -sign_a);
    if (y > maxy)
        return 0;
    int lx2 = x;

    shift_solution(x, y, a, b, -(maxy - y) / a);
    if (y > maxy)
        shift_solution(x, y, a, b, sign_a);
    int rx2 = x;

    if (lx2 > rx2)
        swap(lx2, rx2);
    int lx = max(lx1, lx2);
    int rx = min(rx1, rx2);

    if (lx > rx)
        return 0;
    return (rx - lx) / abs(b) + 1;
}
/*
Once we have $l_x$ and $r_x$, it is also simple to
    enumerate through all the solutions. Just need to
    iterate through $x = l_x + k \cdot \frac{b}{g}$ for all
     $k \ge 0$ until $x = r_x$, and find the corresponding
    $y$ values using the equation $a x + b y = c$.

## Find the solution with minimum value of $x + y$ { data-
    toc-label='Find the solution with minimum value of <
    script type="math/tex">x + y</script>' }

Here, $x$ and $y$ also need to be given some restriction,
    otherwise, the answer may become negative infinity.

The idea is similar to previous section: We find any
    solution of the Diophantine equation, and then shift
    the solution to satisfy some conditions.

Finally, use the knowledge of the set of all solutions to
    find the minimum:

$$x' = x + k \cdot \frac{b}{g},$$

$$y' = y - k \cdot \frac{a}{g}.$$

Note that $x + y$ change as follows:

$$x' + y' = x + y + k \cdot \left(\frac{b}{g} - \frac{a}{g
    }\right) = x + y + k \cdot \frac{b-a}{g}$$

If $a < b$, we need to select smallest possible value of
    $k$. If $a > b$, we need to select the largest possible
     value of $k$. If $a = b$, all solution will have the
    same sum $x + y$.
```

Linear Recurrences

Description: Having a linear recurrence of the form $f(n) = a_1 \cdot f(n-1) + a_2 \cdot f(n-2) \cdots$ can be solved in log time with matrix exponentation.

5a1314

```
#define Matrix vector<vector<ll>>
const ll m = 1000000007;
Matrix operator*(const Matrix& a, const Matrix& b) {
    Matrix c = Matrix(len(a), vector<ll>(len(b[0])));
    for (int i = 0; i < len(a); i++) {
        for (int j = 0; j < len(b[0]); j++) {
            for (int k = 0; k < len(b); k++) {
                c[i][j] += a[i][k]*b[k][j]%m;
                c[i][j] %= m;
            }
        }
    }
    return c;
}
// DOES THIS WORK? Why dp needed?
Matrix fast_exp(const Matrix& a, ll b, map<ll, Matrix>& dp)
     {
    if (dp.count(b)) return dp[b];
    if (b == 1) return a;
    if (b%2) return dp[b] = fast_exp(a, b/2, dp)*fast_exp(a
    , b/2, dp)*a;
    return dp[b] = fast_exp(a, b/2, dp)*fast_exp(a, b/2, dp
    );
}
Matrix operator^(const Matrix& a, ll b) {
    map<ll, Matrix> dp;
    return fast_exp(a, b, dp);
}
void linear_recurrence() {
    /*
        dp[j] += dp[i] * X[i][j] <-- genral case
    */
}
```

Matrix inverse

Description: Yoinked from kactl. Invert matrix A. Returns rank; result is stored in A unless singular (rank $< n$). Can easily be extended to prime moduli; for prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.

Complexity: $\mathcal{O}(n^3)$.

ebfff6

```
int matInv(vector<vector<double>>& A) {
  int n = sz(A); vi col(n);
  vector<vector<double>> tmp(n, vector<double>(n));
  rep(i,0,n) tmp[i][i] = 1, col[i] = i;

  rep(i,0,n) {
    int r = i, c = i;
    rep(j,i,n) rep(k,i,n)
      if (fabs(A[j][k]) > fabs(A[r][c]))
        r = j, c = k;
    if (fabs(A[r][c]) < 1e-12) return i;
    A[i].swap(A[r]); tmp[i].swap(tmp[r]);
    rep(j,0,n)
      swap(A[j][i], A[j][c]), swap(tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    double v = A[i][i];
    rep(j,i+1,n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      rep(k,i+1,n) A[j][k] -= f*A[i][k];
      rep(k,0,n) tmp[j][k] -= f*tmp[i][k];
    }
    rep(j,i+1,n) A[i][j] /= v;
    rep(j,0,n) tmp[i][j] /= v;
    A[i][i] = 1;
  }

  /// forget A at this point, just eliminate tmp backward
  for (int i = n-1; i > 0; --i) rep(j,0,i) {
    double v = A[j][i];
    rep(k,0,n) tmp[j][k] -= v*tmp[i][k];
  }
  rep(i,0,n) rep(j,0,n) A[col[i]][col[j]] = tmp[i][j];
  return n;
}
```

Matrix inverse mod prime

Description: Yoinked from kactl. Returns rank; result is stored in A unless singular (rank $< n$). For prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.

Complexity: $\mathcal{O}(n^3)$.

0d3e3f

```
#include "Mod_pow.h"

int matInv(vector<vector<ll>>& A) {
  int n = sz(A); vi col(n);
  vector<vector<ll>> tmp(n, vector<ll>(n));
  rep(i,0,n) tmp[i][i] = 1, col[i] = i;

  rep(i,0,n) {
    int r = i, c = i;
    rep(j,i,n) rep(k,i,n) if (A[j][k]) {
      r = j; c = k; goto found;
    }
    return i;
found:
    A[i].swap(A[r]); tmp[i].swap(tmp[r]);
    rep(j,0,n) swap(A[j][i], A[j][c]), swap(tmp[j][i], tmp[
    j][c]);
    swap(col[i], col[c]);
    ll v = modpow(A[i][i], mod - 2);
    rep(j,i+1,n) {
      ll f = A[j][i] * v % mod;
      A[j][i] = 0;
      rep(k,i+1,n) A[j][k] = (A[j][k] - f*A[i][k]) % mod;
      rep(k,0,n) tmp[j][k] = (tmp[j][k] - f*tmp[i][k]) %
    mod;
    }
    rep(j,i+1,n) A[i][j] = A[i][j] * v % mod;
    rep(j,0,n) tmp[i][j] = tmp[i][j] * v % mod;
    A[i][i] = 1;
  }

  for (int i = n-1; i > 0; --i) rep(j,0,i) {
    ll v = A[j][i];
    rep(k,0,n) tmp[j][k] = (tmp[j][k] - v*tmp[i][k]) % mod;
  }

  rep(i,0,n) rep(j,0,n)
    A[col[i]][col[j]] = tmp[i][j] % mod + (tmp[i][j] < 0 ?
    mod : 0);
  return n;
}
```

Millar-Rabin primality test

Description: Yoinked from kactl. Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$.

Complexity: 7 times the complexity of $a^b \mod c$.

922927

```
#include "Mod_mul_LL.h"

bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
  ull A[] = {2, 325, 9375, 28178, 450775, 9780504,
     1795265022},
      s = __builtin_ctzll(n-1), d = n >> s;
  for (ull a : A) {   // ^ count trailing zeroes
    ull p = modpow(a%n, d, n), i = s;
    while (p != 1 && p != n - 1 && a % n && i--)
      p = modmul(p, p, n);
```

```
    if (p != n-1 && i != s) return 0;
  }
  return 1;
}
```

Modular inverses

Description: Yoinked from kactl. Pre-computation of modular inverses. Assumes LIM $\leq$ mod and that mod is a prime.

b4a981
```
// const ll mod = 1000000007, LIM = 200000; ///include-line
ll* inv = new ll[LIM] - 1; inv[1] = 1;
rep(i,2,LIM) inv[i] = mod - (mod / i) * inv[mod % i] % mod;
```

Modulo multiplication for 64-bit integers

Description: Yoinked from kactl. Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$. This runs 2x faster than the naive `(__int128_t)a * b % M`.
Complexity: $\mathcal{O}(1)$ for `modmul`, $\mathcal{O}(\log b)$ for `modpow`.

bbbd8f
```
typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) {
  ll ret = a * b - M * ull(1.L / M * a * b);
  return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
  ull ans = 1;
  for (; e; b = modmul(b, b, mod), e /= 2)
    if (e & 1) ans = modmul(ans, b, mod);
  return ans;
}
```

Mod pow

Description: Yoinked from kactl. What u think mans. (this interface is used by a few other things, hence included in the document)
Complexity: $\mathcal{O}(\log e)$.

b83e45
```
const ll mod = 1000000007; // faster if const

ll modpow(ll b, ll e) {
  ll ans = 1;
  for (; e; b = b * b % mod, e /= 2)
    if (e & 1) ans = ans * b % mod;
  return ans;
}
```

Modular arithmetic

Description: Yoinked from kactl. Simple operators for modular arithmetic. You need to set `mod` to some number first and then you can use the structure.

bf099e
```
#include "Euclid.h"

const ll mod = 17; // change to something else
struct Mod {
  ll x;
  Mod(ll xx) : x(xx) {}
  Mod operator+(Mod b) { return Mod((x + b.x) % mod); }
  Mod operator-(Mod b) { return Mod((x - b.x + mod) % mod);
    }
  Mod operator*(Mod b) { return Mod((x * b.x) % mod); }
  Mod operator/(Mod b) { return *this * invert(b); }
  Mod invert(Mod a) {
    ll x, y, g = euclid(a.x, mod, x, y);
    assert(g == 1); return Mod((x + mod) % mod);
  }
  Mod operator^(ll e) {
    if (!e) return Mod(1);
    Mod r = *this ^ (e / 2); r = r * r;
    return e&1 ? *this * r : r;
  }
};
```

Number theoretic transform

Description: Yoinked from kactl. `ntt(a)` computes $\hat{f}(k) = \sum_x a[x] g^{xk}$ for all k, where $g = \text{root}^{(mod-1)/N}$. 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 $[0, mod)$.
Complexity: $\mathcal{O}(n \log n)$.

9c7e10
```
#include "Mod_pow.h"

const ll mod = (119 << 23) + 1, root = 62; // = 998244353
// For p < 2^30 there is also e.g. 5 << 25, 7 << 26, 479 <<
    21
// and 483 << 21 (same root). The last two are > 10^9.
typedef vector<ll> vl;
void ntt(vl &a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  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};
}
```

Polynomial root finding

Description: Yoinked from kactl. Finds the real roots to a polynomial.
Usage: `polyRoots({{2,-3,1}}, -1e9, 1e9); // solve x^2-3x+2 = 0`
Complexity: $\mathcal{O}(n^2 \log(\frac{1}{\epsilon}))$.

7c7d6e
```
#include "Polynomial.h"

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;
}
```

Polynomial thing

Description: Yoinked from kactl. Some poly things I guess.

c9b7b0
```
struct Poly {
  vector<double> a;
  double operator()(double x) const {
    double val = 0;
    for (int i = sz(a); i--;) (val *= x) += a[i];
    return val;
  }
  void diff() {
    rep(i,1,sz(a)) a[i-1] = i*a[i];
    a.pop_back();
  }
  void divroot(double x0) {
    double b = a.back(), c; a.back() = 0;
    for(int i=sz(a)-1; i--;) c = a[i], a[i] = a[i+1]*x0+b,
     b=c;
    a.pop_back();
  }
};
```

Simplex

Description: Yoinked from kactl. 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, -1,-2; vd b = 1,1,-4, c = -1,-1, x; T val = LPSolver(A, b, c).solve(x);`
Complexity: $\mathcal{O}(NM \cdot \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case.

aa8530
```
typedef double T; // long double, Rational, double + mod<P
    >...
typedef vector<T> vd;
typedef vector<vd> vvd;

const 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;
  }
};
```

Solve linear equations

Description: Yoinked from kactl. 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.
Complexity: $\mathcal{O}(n^2 m)$.

44c9ab

```
typedef vector<double> vd;
const 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)
}
```

Solve linear equations extended

Description: Yoinked from kactl. To get all uniquely determined values of x back from SolveLinear, make the following changes:

f65c10

```
#include "Solve_linear.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:; }
```

Phi function

Description: Yoinked from kactl. *Euler's* ϕ function is defined as $\phi(n) :=$ # of positive integers $\leq n$ that are coprime with n. $\phi(1) = 1$, p prime $\Rightarrow \phi(p^k) = (p-1)p^{k-1}$, m, n coprime $\Rightarrow \phi(mn) = \phi(m)\phi(n)$. If $n = p_1^{k_1} p_2^{k_2} \dots p_r^{k_r}$ then $\phi(n) = (p_1 - 1)p_1^{k_1 - 1} \dots (p_r - 1)p_r^{k_r - 1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$. $\sum_{d|n} \phi(d) = n$, $\sum_{1 \leq k \leq n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$ **Euler's thm**: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod{n}$. **Fermat's little thm**: p prime $\Rightarrow a^{p-1} \equiv 1 \pmod{p}$ $\forall a$.

cf7d6d

```
const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
  rep(i,0,LIM) phi[i] = i&1 ? i : i/2;
  for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
    for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

Strings

Aho-Corasick automaton

Description: Yoinked from kactl. Used for multiple pattern matching. Initialize with `AhoCorasick ac(patterns)`; the automaton start node will be at index 0. `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. To find the longest words that start at each position, reverse all input. For large alphabets, split each symbol into chunks, with sentinel bits for symbol boundaries.
Complexity: $26 \cdot \mathcal{O}(N)$ to construct, where $N =$ sum of length of patterns. `find(x)` is $\mathcal{O}(N)$, where $N =$ length of x. `findAll` is $\mathcal{O}(NM)$.

f35677

```
struct AhoCorasick {
  enum {alpha = 26, first = 'A'}; // change this!
  struct Node {
    // (nmatches is optional)
    int back, next[alpha], start = -1, end = -1, nmatches =
      0;
    Node(int v) { memset(next, v, sizeof(next)); }
  };
  vector<Node> N;
  vi backp;
  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++;
  }
  AhoCorasick(vector<string>& pat) : N(1, -1) {
    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);
        }
      }
    }
  }
  vi find(string word) {
    int n = 0;
    vi res; // ll count = 0;
    for (char c : word) {
      n = N[n].next[c - first];
      res.push_back(N[n].end);
      // count += N[n].nmatches;
    }
    return res;
```

```
  }
  vector<vi> findAll(vector<string>& pat, string word) {
    vi r = find(word);
    vector<vi> 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;
  }
};
```

String hashing

Description: Yoinked from kactl. Self-explanatory methods for string hashing.

2d2a67

```
// 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 = (ll)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;}
```

Knuth-Morris-Pratt algorithm

Description: Yoinked from kactl. Finds all occurrences of a pattern in a string. `p[x]` computes the length of the longest prefix of `s` that ends at `x`, other than `s[0...x]` itself (`abacaba -> 0010123`).
Complexity: $\mathcal{O}(n)$.

d4375c

```
vi pi(const string& s) {
  vi 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;
}

vi match(const string& s, const string& pat) {
  vi p = pi(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;
}
```

Manacher

Description: Yoinked from kactl. For each position in a string, computes `p[0][i]` = half length of longest even palindrome around pos `i`, `p[1][i]` = longest odd (half rounded down).
Complexity: $\mathcal{O}(n)$.

e7ad79

```
array<vi, 2> manacher(const string& s) {
  int n = sz(s);
  array<vi,2> p = {vi(n+1), vi(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;
}
```

Min rotation

Description: Yoinked from kactl. Finds the lexicographically smallest rotation of a string.
Usage: `rotate(v.begin(), v.begin() + minRotation(v), v.end());`
Complexity: $\mathcal{O}(n)$.

d07a42

```
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;
}
```

Suffix array

Description: Yoinked from kactl. Builds suffix array for a string. `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`. The input string must not contain any zero bytes.
Complexity: $\mathcal{O}(n \log n)$ per update/query

38db9f

```
struct SuffixArray {
  vi sa, lcp;
  SuffixArray(string& s, int lim=256) { // or basic_string<
    int>
    int n = sz(s) + 1, k = 0, a, b;
    vi 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
    ++;
    }
    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++);
  }
};
```

Suffix automaton

Description: Standard suffix automaton. Does what you'd expect.
Usage: See example main function below. This was thrown in last minute from a working cses solution.
Complexity: $\mathcal{O}(\log n)$ per update/query

3d234e

```
struct SA {
  struct State {
    int length;
    int link;
    int next[26];
    int cnt;
    bool is_clone;
    int first_pos;
    State(int _length, int _link) :
    length(_length),
    link(_link),
    cnt(0),
    is_clone(false),
    first_pos(-1)
    {
      memset(next, -1, sizeof(next));
    }
  };
  std::vector <State> states;
  int size;
  int last;
  bool did_init_count;
  int str_len;
  bool did_init_css;
  SA() :
  states(1, State(0, -1)),
  size(1),
  last(0),
  did_init_count(false),
  str_len(0),
  did_init_css(false)
  { }
  void push(char c) {
    str_len++;
    did_init_count = false;
    did_init_css = false;
    int cur = size;
```

```
    states.resize(++size, State(states[last].length + 1,
    -1));
    states[cur].first_pos = states[cur].length - 1;
    int p = last;
    while (p != -1 && states[p].next[c - 'a'] == -1) {
      states[p].next[c - 'a'] = cur;
      p = states[p].link;
    }
    if (p == -1) {
      states[cur].link = 0;
    } else {
      int q = states[p].next[c - 'a'];
      if (states[p].length + 1 == states[q].length) {
        states[cur].link = q;
      } else {
        int clone = size;
        states.resize(++size, State(states[p].length + 1,
     states[q].link));
        states[clone].is_clone = true;
        memcpy(states[clone].next, states[q].next, sizeof(
     State::next));
        states[clone].first_pos = states[q].first_pos;
        while (p != -1 && states[p].next[c - 'a'] == q) {
          states[p].next[c - 'a'] = clone;
          p = states[p].link;
        }
        states[q].link = states[cur].link = clone;
      }
    }
    last = cur;
  }
  bool exists(const std::string& pattern) {
    int node = 0;
    int index = 0;
    while (index < (int) pattern.length() && states[node].
     next[pattern[index] - 'a'] != -1) {
      node = states[node].next[pattern[index] - 'a'];
      index++;
    }
    return index == (int) pattern.size();
  }
  int count(const std::string& pattern) {
    if (!did_init_count) {
      did_init_count = true;
      for (int i = 1; i < size; i++) {
        states[i].cnt = !states[i].is_clone;
      }
      std::vector <std::vector <int>> of_length(str_len +
     1);
      for (int i = 0; i < size; i++) {
        of_length[states[i].length].push_back(i);
      }
      for (int l = str_len; l >= 0; l--) {
        for (int node : of_length[l]) {
          if (states[node].link != -1) {
            states[states[node].link].cnt += states[node].
     cnt;
          }
        }
      }
    }
    int node = 0;
    int index = 0;
    while (index < (int) pattern.length() && states[node].
     next[pattern[index] - 'a'] != -1) {
      node = states[node].next[pattern[index] - 'a'];
      index++;
    }
    return index == (int) pattern.size() ? states[node].cnt
      : 0;
  }
```

```
  int first_occ(const std::string& pattern) {
    int node = 0;
    int index = 0;
    while (index < (int) pattern.length() && states[node].
     next[pattern[index] - 'a'] != -1) {
      node = states[node].next[pattern[index] - 'a'];
      index++;
    }
    return index == (int) pattern.size() ? states[node].
     first_pos - (int) pattern.size() + 1 : -1;
  }
  size_t count_substrings() {
    static std::vector <size_t> dp;
    if (!did_init_css) {
      did_init_css = true;
      dp = std::vector <size_t> (size, 0);
      auto dfs = [&] (auto&& self, int node) -> size_t {
        if (node == -1) {
          return 0;
        }
        if (dp[node]) {
          return dp[node];
        }
        dp[node] = 1;
        for (int i = 0; i < 26; i++) {
          dp[node] += self(self, states[node].next[i]);
        }
        return dp[node];
      };
      dfs(dfs, 0);
    }
    return dp[0] - 1;
  }
};

// usage example: Repeating Substring submission on cses.fi
int main() {
  std::ios::sync_with_stdio(0); std::cin.tie(0);
  std::string s; std::cin >> s;
  int n; std::cin >> n;
  SA sa;
  for (char c : s) {
    sa.push(c);
  }
  sa.count("");
  int len = -1;
  int ind = -1;
  for (int i = 1; i < sa.size; i++) {
    if (sa.states[i].cnt > 1) {
      if (len < sa.states[i].length) {
        len = sa.states[i].length;
        ind = sa.states[i].first_pos - len + 1;
      }
    }
  }
  if (len == -1) {
    std::cout << "-1\n";
    return 0;
  }
  for (int i = 0; i < len; i++) {
    std::cout << s[i + ind];
  }
  std::cout << "\n";
}
```

Suffix tree

Description: Yoinked from kactl. 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).

Complexity: $26 \cdot \mathcal{O}(n)$.

aae0b8

```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur position
  int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;

  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]; }
      if (q==r[m])  s[m]=v;  else s[m]=m+2;
      q=r[v]-(q-r[m]);  m+=2;  goto suff;
    }
  }

  SuffixTree(string a) : a(a) {
    fill(r,r+N,sz(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALPHA,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] =
      0;
    rep(i,0,sz(a)) ukkadd(i, toi(a[i]));
  }

  // example: find longest common substring (uses ALPHA =
    28)
  pii best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node]) :
      0;
    rep(c,0,ALPHA) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2)
    );
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
  }
};
```

Z function

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

Complexity: $\mathcal{O}(n)$.

ee09e2

```
vi Z(const string& S) {
  vi z(sz(S));
  int l = -1, r = -1;
  rep(i,1,sz(S)) {
```

```
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    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;
}
```

Various

Bump allocator

Description: Yoinked from kactl. 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

```
// 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*) {}
```

Fast integer input

Description: Yoinked from kactl. USE THIS IF TRYING TO CONSTANT TIME OPTIMIZE SOLUTION READING IN LOTS OF INTEGERS!!! Read an integer from stdin. Usage requires your program to pipe in input from file.
Usage: `./a.out < input.txt`
Complexity: About 5x as fast as cin/scanf.

7b3c70

```
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 - 480;
  return a - 48;
}
```

Fast knapsack

Description: Yoinked from kactl. Given N non-negative integer weights w and a non-negative target t, computes the maximum $S <= t$ such that S is the sum of some subset of the weights.
Complexity: $\mathcal{O}(N \max(w_i))$.

b20ccc

```
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;
}
```

Fast mod reduction

Description: Yoinked from kactl. Compute $a\%b$ about 5 times faster than usual, where b is constant but not known at compile time. Returns a value congruent to $a \pmod b$ in the range $[0, 2b)$. (proven correct)

751a02

```
typedef unsigned long long ull;
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;
  }
};
```

Interval container

Description: Yoinked from kactl. 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)`.
Complexity: $\mathcal{O}(\log n)$ per update/query

edce47

```
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);
}
```

Interval cover

Description: Yoinked from kactl. 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).
Complexity: $\mathcal{O}(n \log n)$.

9e9d8d

```
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;
}
```

Knuth DP optimization

Description: Yoinked from kactl. When doing DP on intervals: $a[i][j] = \min_{i<k<j}(a[i][k] + a[k][j]) + f(i, j)$, 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 $a \leq b \leq c \leq d$.
Complexity: $\mathcal{O}(N^2)$.

210610

```
// generic implementation frmo cp-algorithms:
int solve() {
    int N;
    ... // read N and input
    int dp[N][N], opt[N][N];
    auto C = [&](int i, int j) {
        ... // Implement cost function C.
    };
    for (int i = 0; i < N; i++) {
        opt[i][i] = i;
        ... // Initialize dp[i][i] according to the problem
    }
    for (int i = N-2; i >= 0; i--) {
        for (int j = i+1; j < N; j++) {
            int mn = INT_MAX;
            int cost = C(i, j);
            for (int k = opt[i][j-1]; k <= min(j-1, opt[i
 +1][j]); k++) {
                if (mn >= dp[i][k] + dp[k+1][j] + cost) {
                    opt[i][j] = k;
                    mn = dp[i][k] + dp[k+1][j] + cost;
                }
            }
            dp[i][j] = mn;
        }
    }
    cout << dp[0][N-1] << endl;
}
```

Longest increasing subsequence

Description: Yoinked from kactl. Computes the longest increasing subsequence of a sequence.
Complexity: $\mathcal{O}(n \log n)$.

2932a0

```
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;
}
```

Small ptr

Description: Yoinked from kactl. A 32-bit pointer that points into BumpAllocator memory.

780757
```
#include "Bump_allocator.h"

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; }
};
```

Xor Basis

Description: basis of vectors in Z_2^d

e54101
```
int basis[d]; // basis[i] keeps the mask of the vector
    whose f value is i
int sz; // Current size of the basis
void insertVector(int mask) {
  for (int i = 0; i < d; i++) {
    if ((mask & 1 << i) == 0) continue; // continue if i !=
     f(mask)

    if (!basis[i]) { // If there is no basis vector with
     the i'th bit set, then insert this vector into the
     basis
      basis[i] = mask;
      ++sz;

      return;
    }

    mask ^= basis[i]; // Otherwise subtract the basis
     vector from this vector
  }
}
```