ITMO

Shuffle, duffle, muzzle, muff. Fista, wista, mista-cuff

overpressured but trained

The 2024 ICPC Northern Eurasia Finals

December 15, 2024

Mathematics (1)

1.1 Equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{matrix} ax + by = e \\ cx + dy = f \end{matrix} \Rightarrow \begin{matrix} x = \dfrac{ed - bf}{ad - bc} \\ y = \dfrac{af - ec}{ad - bc} \end{matrix}$$

In general, given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A_i'}{\det A}$$

where A_i' is A with the i'th column replaced by b.

1.2 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k - c_1 x^{k-1} - \cdots - c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

1.3 Trigonometry

$$\sin(v + w) = \sin v \cos w + \cos v \sin w$$
$$\cos(v + w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v + w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$
$$\sin v + \sin w = 2 \sin \frac{v + w}{2} \cos \frac{v - w}{2}$$
$$\cos v + \cos w = 2 \cos \frac{v + w}{2} \cos \frac{v - w}{2}$$

$$(V + W) \tan(v - w)/2 = (V - W) \tan(v + w)/2$$

where V, W are lengths of sides opposite angles v, w.

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2 + b^2}, \phi = \operatorname{atan2}(b, a)$.

1.4 Geometry

1.4.1 Triangles

Side lengths: a, b, c

Semiperimeter: $p = \dfrac{a + b + c}{2}$

Area: $A = \sqrt{p(p - a)(p - b)(p - c)}$

Circumradius: $R = \dfrac{abc}{4A}$

Inradius: $r = \dfrac{A}{p}$

Length of median (divides triangle into two equal-area triangles): $m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$

Length of bisector (divides angles in two):

$$s_a = \sqrt{bc\left[1 - \left(\frac{a}{b + c}\right)^2\right]}$$

Law of sines: $\dfrac{\sin\alpha}{a} = \dfrac{\sin\beta}{b} = \dfrac{\sin\gamma}{c} = \dfrac{1}{2R}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc\cos\alpha$

Law of tangents: $\dfrac{a + b}{a - b} = \dfrac{\tan\frac{\alpha + \beta}{2}}{\tan\frac{\alpha - \beta}{2}}$

1.4.2 Quadrilaterals

With side lengths a, b, c, d, diagonals e, f, diagonals angle θ, area A and magic flux $F = b^2 + d^2 - a^2 - c^2$:

$$4A = 2ef \cdot \sin\theta = F \tan\theta = \sqrt{4e^2 f^2 - F^2}$$

For cyclic quadrilaterals the sum of opposite angles is $180°$, $ef = ac + bd$, and $A = \sqrt{(p - a)(p - b)(p - c)(p - d)}$.

1.5 Derivatives/Integrals

$$\frac{d}{dx}\arcsin x = \frac{1}{\sqrt{1 - x^2}} \qquad \frac{d}{dx}\arccos x = -\frac{1}{\sqrt{1 - x^2}}$$
$$\frac{d}{dx}\tan x = 1 + \tan^2 x \qquad \frac{d}{dx}\arctan x = \frac{1}{1 + x^2}$$
$$\int \tan ax = -\frac{\ln|\cos ax|}{a} \qquad \int x \sin ax = \frac{\sin ax - ax\cos ax}{a^2}$$
$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2}\operatorname{erf}(x) \qquad \int xe^{ax}dx = \frac{e^{ax}}{a^2}(ax - 1)$$

Integration by parts:

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

1.6 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1$$

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$
$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(2n+1)(n+1)}{6}$$
$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$
$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

1.7 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots, (-\infty < x < \infty)$$

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \leq 1)$$

$$\sqrt{1 + x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots, (-1 \leq x \leq 1)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, (-\infty < x < \infty)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, (-\infty < x < \infty)$$

1.8 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x. It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent X and Y,

$$V(aX + bY) = a^2 V(X) + b^2 V(Y).$$

1.9 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ be a sequence of random variables generated by the Markov process. Then there is a transition matrix $\mathbf{P} = (p_{ij})$, with $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

π is a stationary distribution if $\pi = \pi\mathbf{P}$. If the Markov chain is *irreducible* (it is possible to get to any state from any state), then $\pi_i = \frac{1}{\mathbb{E}(T_i)}$ where $\mathbb{E}(T_i)$ is the expected time between two visits in state i. π_j/π_i is the expected number of visits in state j between two visits in state i.

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors, π_i is proportional to node i's degree.

A Markov chain is *ergodic* if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and *aperiodic* (i.e., the gcd of cycle lengths is 1). $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{1}\pi$.

A Markov chain is an A-chain if the states can be partitioned into two sets $\mathbf{A}$ and $\mathbf{G}$, such that all states in $\mathbf{A}$ are absorbing ($p_{ii} = 1$), and all states in $\mathbf{G}$ leads to an absorbing state in $\mathbf{A}$. The probability for absorption in state $i \in \mathbf{A}$, when the initial state is j, is $a_{ij} = p_{ij} + \sum_{k\in\mathbf{G}} a_{ik} p_{kj}$. The expected time until absorption, when the initial state is i, is $t_i = 1 + \sum_{k\in\mathbf{G}} p_{ki} t_k$.

Data structures (2)

StatisticTree.h

Description: A set (not multiset!) with support for finding the n'th element, and finding the index of an element. To get a map, change `null_type`.
Time: $\mathcal{O}(\log N)$

d41d8c, 16 lines

```
#include <bits/extc++.h>
using namespace __gnu_pbds;

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

void example() {
  Tree<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
}
```

HashMap.h

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

d41d8c, 7 lines

```
#include <bits/extc++.h>
// To use most bits rather than just the lowest ones:
struct chash { // large odd number for C
  const uint64_t C = ll(4e18 * acos(0)) | 71;
  ll operator()(ll x) const { return __builtin_bswap64(x*C); }
};
__gnu_pbds::gp_hash_table<ll,int,chash> h({},{},{},{},{1<<16});
```

Segtree.h

Description: tree for mx and += ops
Time: $\mathcal{O}(\log N)$

d41d8c, 46 lines

```
struct segtree {
    int n;
    vector<ll> mx, upd;

    segtree() { }

    segtree(int n): n(n) {
        mx.resize(n * 4);
        upd.resize(n * 4);
    }

    void add(int v, int vl, int vr, int l, int r, ll x) {
        if (vr <= l || vl >= r)
            return;
        if (l <= vl && vr <= r) {
            mx[v] += x;
            upd[v] += x;
            return;
        }
        int vm = (vl + vr) >> 1;
        add(v * 2, vl, vm, l, r, x);
        add(v * 2 + 1, vm, vr, l, r, x);
        mx[v] = max(mx[v * 2], mx[v * 2 + 1]) + upd[v];
    }

    void add(int l, int r, ll x) {
        add(1, 0, n, l, r, x);
    }

    ll getMax(int v, int vl, int vr, int l, int r) {
        if (vr <= l || vl >= r)
            return INT64_MIN;
        if (l <= vl && vr <= r)
            return mx[v];
        int vm = (vl + vr) >> 1;
        return max(getMax(v * 2, vl, vm, l, r), getMax(v * 2 +
             1, vm, vr, l, r)) + upd[v];
    }

    ll getMax(int l, int r) {
        return getMax(1, 0, n, l, r);
    }

    ll getMax() {
        return mx[1];
    }
};
```

LazySegmentTree.h

Description: Segment tree with ability to add or set values of large intervals, and compute max of intervals. Can be changed to other things. Use with a bump allocator for better performance, and SmallPtr or implicit indices to save memory.
Usage: `Node* tr = new Node(v, 0, sz(v));`
Time: $\mathcal{O}(\log N)$.

"../various/BumpAllocator.h" d41d8c, 50 lines

```
const int inf = 1e9;
struct Node {
  Node *l = 0, *r = 0;
  int lo, hi, mset = inf, madd = 0, val = -inf;
  Node(int lo,int hi):lo(lo),hi(hi){} // Large interval of -inf
  Node(vi& v, int lo, int hi) : lo(lo), hi(hi) {
    if (lo + 1 < hi) {
      int mid = lo + (hi - lo)/2;
      l = new Node(v, lo, mid); r = new Node(v, mid, hi);
      val = max(l->val, r->val);
    }
    else val = v[lo];
  }
  int query(int L, int R) {
    if (R <= lo || hi <= L) return -inf;
    if (L <= lo && hi <= R) return val;
    push();
    return max(l->query(L, R), r->query(L, R));
  }
  void set(int L, int R, int x) {
    if (R <= lo || hi <= L) return;
    if (L <= lo && hi <= R) mset = val = x, madd = 0;
    else {
      push(), l->set(L, R, x), r->set(L, R, x);
      val = max(l->val, r->val);
    }
  }
  void add(int L, int R, int x) {
    if (R <= lo || hi <= L) return;
    if (L <= lo && hi <= R) {
      if (mset != inf) mset += x;
      else madd += x;
      val += x;
    }
    else {
      push(), l->add(L, R, x), r->add(L, R, x);
      val = max(l->val, r->val);
    }
  }
  void push() {
    if (!l) {
      int mid = lo + (hi - lo)/2;
      l = new Node(lo, mid); r = new Node(mid, hi);
    }
    if (mset != inf)
      l->set(lo,hi,mset), r->set(lo,hi,mset), mset = inf;
    else if (madd)
      l->add(lo,hi,madd), r->add(lo,hi,madd), madd = 0;
  }
};
```

UnionFindRollback.h

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

d41d8c, 21 lines

```
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;
  }
};
```

SubMatrix.h

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

d41d8c, 13 lines

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

Matrix.h

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

d41d8c, 26 lines

```
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;
  }
};
```

LineContainer.h

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

d41d8c, 30 lines

```
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

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

RationalLineContainer.h

Description: CHT with rationals
Time: $\mathcal{O}\left(\log N\right)$

d41d8c, 63 lines

```
struct CHT {
    struct Frac {
        ll a, b;

        bool operator <(const Frac &other) const {
            return (__int128_t)a * other.b < (__int128_t)other.
                a * b;
        }

        bool operator >(const Frac &other) const {
            return other < *this;
        }
    };

    struct Line {
        ll k, b;

        ll get(ll x) const {
            return k * x + b;
        }

        Frac intersect(const Line &other) const {
            Frac res = {other.b - b, k - other.k};
            if (res.b < 0) {
                res.a *= -1;
                res.b *= -1;
            }
            return res;
        }
    };

    vector<Frac> pt;
    vector<Line> lines;

    void addLine(ll k, ll b) {  // k[i] descrease
        Line l = {k, b};
        while (!pt.empty()) {
            const Line &back = lines.back();
            if (back.k == k) {
                if (b >= back.b)
                    return;
            } else if (l.intersect(back) > pt.back()) {
                break;
            }
            pt.pop_back();
            lines.pop_back();
        }
        if (!lines.empty() && lines.back().k == k) {
            if (b < lines.back().b)
                lines.back() = l;
            return;
        }
        if (!lines.empty())
            pt.push_back(l.intersect(lines.back()));
        lines.push_back(l);
    }

    ll getMin(ll x) {
        if (lines.empty())
            return INT64_MAX;
        int pos = lower_bound(all(pt), Frac{x, 1}) - pt.begin()
            ;
        return lines[pos].get(x);
    }
};
```

FenwickTree2d.h

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

"FenwickTree.h" d41d8c, 22 lines

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

RMQ.h

Description: Range Minimum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time.

Usage: `RMQ rmq(values);`
`rmq.query(inclusive, exclusive);`

Time: $\mathcal{O}(|V| \log |V| + Q)$

d41d8c, 65 lines

```
// usage:
//   auto fun = [&](int i, int j) { return min(i, j); };
//   SparseTable<int, decltype(fun)> st(a, fun);
// or:
//   SparseTable<int> st(a, [&](int i, int j) { return min(i, j
    ); });
template <typename T, class F = function<T(const T&, const T&)
    >>
class SparseTable {
 public:
  int n;
  vector<vector<T>> mat;
  F func;

  SparseTable(const vector<T>& a, const F& f) : func(f) {
    n = static_cast<int>(a.size());
    int max_log = 32 - __builtin_clz(n);
    mat.resize(max_log);
    mat[0] = a;
    for (int j = 1; j < max_log; j++) {
      mat[j].resize(n - (1 << j) + 1);
      for (int i = 0; i <= n - (1 << j); i++) {
        mat[j][i] = func(mat[j - 1][i], mat[j - 1][i + (1 << (j
            - 1))]);
      }
    }
  }

  T get(int from, int to) const {
    assert(0 <= from && from <= to && to <= n - 1);
    int lg = 32 - __builtin_clz(to - from + 1) - 1;
    return func(mat[lg][from], mat[lg][to - (1 << lg) + 1]);
  }
};

template <typename T, typename Func>
class DisjointSparseTable {
 public:
  int _n;
  vector<vector<T>> _matrix;
  Func _func;

  DisjointSparseTable(const vector<T>& a, const Func& func) :
      _n(static_cast<int>(a.size())), _func(func) {
    _matrix.push_back(a);
    for (int layer = 1; (1 << layer) < _n; ++layer) {
      _matrix.emplace_back(_n);
      for (int mid = 1 << layer; mid < _n; mid += 1 << (layer +
          1)) {
        _matrix[layer][mid - 1] = a[mid - 1];
        for (int j = mid - 2; j >= mid - (1 << layer); --j) {
          _matrix[layer][j] = _func(a[j], _matrix[layer][j +
              1]);
        }
        _matrix[layer][mid] = a[mid];
        for (int j = mid + 1; j < min(_n, mid + (1 << layer));
            ++j) {
          _matrix[layer][j] = _func(_matrix[layer][j - 1], a[j
              ]);
        }
      }
    }
  }

  T Query(int l, int r) const {
    assert(0 <= l && l < r && r <= _n);
    if (r - l == 1) {
      return _matrix[0][l];
    }
    int layer = 31 - __builtin_clz(l ^ (r - 1));
    return _func(_matrix[layer][l], _matrix[layer][r - 1]);
  }
};
```

MoQueries.h

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

Time: $\mathcal{O}\left(N\sqrt{Q}\right)$

d41d8c, 49 lines

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

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

vi moTree(vector<array<int, 2>> Q, vector<vi>& ed, int root=0){
  int N = sz(ed), pos[2] = {}, blk = 350; // ~N/sqrt(Q)
  vi s(sz(Q)), res = s, I(N), L(N), R(N), in(N), par(N);
  add(0, 0), in[0] = 1;
  auto dfs = [&](int x, int p, int dep, auto& f) -> void {
    par[x] = p;
    L[x] = N;
    if (dep) I[x] = N++;
    for (int y : ed[x]) if (y != p) f(y, x, !dep, f);
    if (!dep) I[x] = N++;
    R[x] = N;
  };
  dfs(root, -1, 0, dfs);
#define K(x) pii(I[x[0]] / blk, I[x[1]] ^ -(I[x[0]] / blk & 1))
  iota(all(s), 0);
  sort(all(s), [&](int s, int t){ return K(Q[s]) < K(Q[t]); });
  for (int qi : s) rep(end,0,2) {
    int &a = pos[end], b = Q[qi][end], i = 0;
#define step(c) { if (in[c]) { del(a, end); in[a] = 0; } \
                  else { add(c, end); in[c] = 1; } a = c; }
    while (!(L[b] <= L[a] && R[a] <= R[b]))
      I[i++] = b, b = par[b];
    while (a != b) step(par[a]);
    while (i--) step(I[i]);
    if (end) res[qi] = calc();
  }
  return res;
}
```

Numerical (3)

3.1 Polynomials and recurrences

Polynomial.h

Description: Polynomial operations

d41d8c, 290 lines

```
namespace Polynomial {

    template<typename base>
    vector<base> derivative(vector<base> a) {
        int n = a.size();
        for (int i = 0; i < n - 1; ++i) {
            a[i] = a[i + 1] * (i + 1);
        }
        a.pop_back();
        return a;
    }

    template<typename base>
    vector<base> integral(vector<base> a) {
        int n = a.size();
        a.push_back(0);
        for (int i = n; i > 0; --i) {
            a[i] = a[i - 1] / i;
        }
        a[0] = 0;
        return a;
    }

    template<typename base>
    vector<base> add(vector<base> a, const vector<base> &b) {
        int n = a.size(), m = b.size();
        a.resize(max(n, m));
        for (int i = 0; i < max(n, m); ++i) {
            a[i] = (i >= a.size() ? 0 : a[i]) + (i >= b.size()
                ? 0 : b[i]);
        }
        return a;
    }

    template<typename base>
    vector<base> sub(vector<base> a, const vector<base> &b) {
        int n = a.size(), m = b.size();
        a.resize(max(n, m));
        for (int i = 0; i < max(n, m); ++i) {
            a[i] = (i >= a.size() ? 0 : a[i]) - (i >= b.size()
                ? 0 : b[i]);
        }
        return a;
    }

    namespace NTT {
        const int MOD = 998244353;
        const int g = 3;

        vector<int> R;

        void NTT(vector<Mint<MOD>>& a, int n, int on) {
            for (int i = 0; i < n; i++)
                if (i < R[i])
                    swap(a[i], a[R[i]]);
            Mint<MOD> wn, u, v;
            for (int i = 1, m = 2; i < n; i = m, m <<= 1) {
                wn = Mint<MOD>::binpow(g, (MOD - 1) / m);
                if (on == -1)
                    wn = 1 / wn;
                for (int j = 0; j < n; j += m) {
                    Mint<MOD> w = 1;
```

```
                for (int k = 0; k < i; k++, w *= wn) {
                    u = a[j + k], v = w * a[i + j + k];
                    a[j + k] = u + v;
                    a[i + j + k] = u - v;
                }
            }
        }
        if (on == -1) {
            Mint<MOD> k = Mint<MOD>(1) / Mint<MOD>(n);
            for (int i = 0; i < n; i++)
                a[i] = a[i] * k;
        }
    }

    template<typename base>
    vector<base> mul(vector<base>& A, vector<base>& B) {
        static_assert(std::is_same_v<base, Mint<MOD>>);
        assert(A.size() == B.size() && __builtin_popcount(A
            .size()) == 1);
        int n = A.size();
        int L = __builtin_ctz(n);
        if (R.size() != n) {
            R.assign(n, 0);
            for (int i = 0; i < n; i++)
                R[i] = (R[i >> 1] >> 1) | ((i & 1) << (L -
                    1));
        }
        NTT(A, n, 1);
        NTT(B, n, 1);
        for (int i = 0; i < n; i++)
            A[i] *= B[i];
        NTT(A, n, -1);
        return A;
    }
}

int get_lim(int n) {
    int res = 1;
    while (res < n) {
        res <<= 1;
    }
    return res;
}

template<typename base>
vector<base> mul(vector<base> a, vector<base> b, int size)
    {
    int l = get_lim(a.size() + b.size());
    a.resize(l);
    b.resize(l);
    auto res = NTT::mul(a, b);
    res.resize(size);
    return res;
}

template<typename base>
vector<base> mul(vector<base> a, base scalar) {
    for (auto& val : a)
        val *= scalar;
    return a;
}

template<typename base>
vector<base> mul(const vector<base> &a, const vector<base>
    &b) {
    return mul(a, b, a.size() + b.size() - 1);
}

template <typename base>
vector<base> plug_minus_x(vector<base> a) {
    for (int i = 1; i < a.size(); i += 2) {
        a[i] *= -1;
    }
    return a;
}

template <typename base>
void plug_x_squared_inplace(vector<base>& a) {
    a.resize(a.size() * 2);
    for (int i = (int)a.size() * 2 - 1; i >= 0; --i) {
        if (i % 2 != 0) a[i] = 0;
        else a[i] = a[i / 2];
    }
}

template <typename base>
vector<base> plug_x_squared(const vector<base>& a) {
    vector<base> res(a.size() * 2);
    for (int i = 0; i < a.size(); ++i) {
        res[i * 2] = a[i];
    }
    return res;
}

template <typename base>
void only_even_inplace(vector<base>& a) {
    for (int i = 0; i < a.size(); i += 2) {
        a[i / 2] = a[i];
    }
    a.resize((a.size() + 1) / 2);
}

template <typename base>
vector<base> only_even(const vector<base>& a) {
    vector<base> res((a.size() + 1) / 2);
    for (int i = 0; i < a.size(); i += 2) {
        res[i / 2] = a[i];
    }
    return res;
}

// O(n*log(n))
template<typename base>
void inverse_inplace(vector<base> &a, int size) {
    assert(!a.empty() && a[0] != 0);
    if (size == 0) {
        a = {0};
        return;
    }
    if (size == 1) {
        a = {1/a[0]};
        return;
    }
    auto op = plug_minus_x(a);
    auto T = mul(a, op);
    only_even_inplace(T);
    inverse_inplace(T, (size + 1) / 2);
    plug_x_squared_inplace(T);
    a = mul(op, T, size);
}

template <typename base>
vector<base> inverse(const vector<base>& a, int size) {
    assert(size > 0 && a[0] != 0);
    vector<base> Q{1/a[0]};

    for (int sz = 2;; sz *= 2) {
        Q = mul(Q, sub({2}, mul(a, Q, sz)), sz);
        if (sz >= size)
            break;
    }
    Q.resize(size);
    return Q;
}

//    // O(n*log(n)) too slow, big constant factor
//    template<typename base>
//    vector<base> inverse(const vector<base> &a, int size) {
//        assert(!a.empty() && a[0] != 0);
//        if (size == 0) {
//            return {0};
//        }
//        if (size == 1) {
//            return {1/a[0]};
//        }
//        auto op = plug_minus_x(a);
//        auto T = mul(a, op);
//        T = only_even(T);
//        T = inverse(T, (size + 1) / 2);
//        T = plug_x_squared(T);
//        auto res = mul(op, T, size);
//        return res;
//    }

    template<typename base>
    vector<base> divide(const vector<base> &a, const vector<
        base> &b, int size) {
        return mul(a, inverse(b, size), size);
    }

    // O(n*log(n))
    template<typename base>
    vector<base> ln(const vector<base> &a, int size) {
        auto res = integral(divide(derivative(a), a, size));
        res.resize(size);
        return res;
    }

    // O(n*log(n))
    template<typename base>
    vector<base> exp(const vector<base> &a, int size) {
        assert(size > 0 && a[0] == 0);
        vector<base> Q{1};

        for (int sz = 2;; sz *= 2) {
            Q = mul(Q, sub(add(a, {1}), ln(Q, sz)), sz);
            if (sz >= size)
                break;
        }
        Q.resize(size);
        return Q;
    }

    // O(n*log(n))
    template<typename base>
    vector<base> pow(vector<base> a, ll p, int size) {
        int i = 0;
        while (i < a.size()) {
            if (a[i] != 0)
                break;
            ++i;
        }
        if (i == a.size()) {
            auto res = vector<base>(size, 0);
            if (p == 0)
                res[0] = 1;
            return res;
```

```
        }
        a.erase(a.begin(), a.begin() + i);
        auto f = a[0];
        for (auto& x : a) x /= f;
        a = exp(mul(ln(a, size), (base)p), size);
        for (int j = size - 1; j >= 0; --j) {
            if ((i > 0 && p >= size) || j - p * i < 0)
                a[j] = 0;
            else
                a[j] = a[j - i * p];
            a[j] *= base::binpow(f, p);
        }
        return a;
    }
}

int32_t main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    const int MOD = 998244353;
    int n; cin >> n;
    ll m; cin >> m;
    vector<Mint<MOD>> a(n);
    for (auto& x : a) cin >> x;
    auto res = Polynomial::pow(a, m, n);
    for (auto x : res) cout << x << " ";
}
```

PolyInterpolate.h

Description: Given n points (x[i], y[i]), computes an n-1-degree polynomial p that passes through them: $p(x) = a[0] * x^0 + ... + a[n-1] * x^{n-1}$. For numerical precision, pick $x[k] = c * \cos(k/(n-1) * \pi), k = 0 \dots n-1$.
Time: $\mathcal{O}\left(n^2\right)$

d41d8c, 13 lines

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

BerlekampMassey.h

Description: Recovers any n-order linear recurrence relation from the first $2n$ terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size $\leq n$.
Usage: `berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}`
Time: $\mathcal{O}\left(N^2\right)$ and $\mathcal{O}\left(nlog^2(k)\right)$

d41d8c, 143 lines

```
constexpr int mod = 1e9 + 7;
template<int32_t MOD>
struct modint {
    int32_t value;

    modint() = default;

    modint(int32_t value_) : value(value_) {}

    inline modint<MOD> operator+(modint<MOD> other) const {
        int32_t c = this->value + other.value;
        return modint<MOD>(c >= MOD ? c - MOD : c);
    }

    inline modint<MOD> operator-(modint<MOD> other) const {
        int32_t c = this->value - other.value;
        return modint<MOD>(c < 0 ? c + MOD : c);
    }

    inline modint<MOD> operator*(modint<MOD> other) const {
        int32_t c = (int64_t) this->value * other.value % MOD;
        return modint<MOD>(c < 0 ? c + MOD : c);
    }

    inline modint<MOD> &operator+=(modint<MOD> other) {
        this->value += other.value;
        if (this->value >= MOD) this->value -= MOD;
        return *this;
    }

    inline modint<MOD> &operator-=(modint<MOD> other) {
        this->value -= other.value;
        if (this->value < 0) this->value += MOD;
        return *this;
    }

    inline modint<MOD> &operator*=(modint<MOD> other) {
        this->value = (int64_t) this->value * other.value % MOD
            ;
        if (this->value < 0) this->value += MOD;
        return *this;
    }

    inline modint<MOD> operator-() const { return modint<MOD>(
        this->value ? MOD - this->value : 0); }

    modint<MOD> pow(uint64_t k) const {
        modint<MOD> x = *this, y = 1;
        for (; k; k >>= 1) {
            if (k & 1) y *= x;
            x *= x;
        }
        return y;
    }

    modint<MOD> inv() const { return pow(MOD - 2); }  // MOD
        must be a prime
    inline modint<MOD> operator/(modint<MOD> other) const {
        return *this * other.inv(); }

    inline modint<MOD> operator/=(modint<MOD> other) { return *
        this *= other.inv(); }

    inline bool operator==(modint<MOD> other) const { return
        value == other.value; }

    inline bool operator!=(modint<MOD> other) const { return
        value != other.value; }

    inline bool operator<(modint<MOD> other) const { return
        value < other.value; }

    inline bool operator>(modint<MOD> other) const { return
        value > other.value; }
};

template<int32_t MOD>
modint<MOD> operator*(int64_t value, modint<MOD> n) { return
    modint<MOD>(value) * n; }

template<int32_t MOD>
modint<MOD> operator*(int32_t value, modint<MOD> n) { return
    modint<MOD>(value % MOD) * n; }

template<int32_t MOD>
istream &operator>>(istream &in, modint<MOD> &n) { return in >>
    n.value; }

template<int32_t MOD>
ostream &operator<<(ostream &out, modint<MOD> n) { return out
    << n.value; }

using mint = modint<mod>;

vector<mint> BerlekampMassey(vector<mint> S) {
    int n = (int) S.size(), L = 0, m = 0;
    vector<mint> C(n), B(n), T;
    C[0] = B[0] = 1;
    mint b = 1;
    for (int i = 0; i < n; i++) {
        ++m;
        mint d = S[i];
        for (int j = 1; j <= L; j++) d += C[j] * S[i - j];
        if (d == 0) continue;
        T = C;
        mint coef = d * b.inv();
        for (int j = m; j < n; j++) C[j] -= coef * B[j - m];
        if (2 * L > i) continue;
        L = i + 1 - L;
        B = T;
        b = d;
        m = 0;
    }
    C.resize(L + 1);
    C.erase(C.begin());
    for (auto &x: C) x *= -1;
    return C;
}

vector<mint> combine(int n, vector<mint> &a, vector<mint> &b,
    vector<mint> &tr) {
    vector<mint> res(n * 2 + 1, 0);
    for (int i = 0; i < n + 1; i++) {
        for (int j = 0; j < n + 1; j++) res[i + j] += a[i] * b[
            j];
    }
    for (int i = 2 * n; i > n; --i) {
        for (int j = 0; j < n; j++) res[i - 1 - j] += res[i] *
            tr[j];
    }
    res.resize(n + 1);
    return res;
};

// transition -> for(i = 0; i < x; i++) f[n] += tr[i] * f[n-i
    -1]
// S contains initial values, k is 0 indexed
mint LinearRecurrence(vector<mint> &S, vector<mint> &tr, long
    long k) {
    int n = S.size();
    assert(n == (int) tr.size());
    if (n == 0) return 0;
    if (k < n) return S[k];
    vector<mint> pol(n + 1), e(pol);
    pol[0] = e[1] = 1;
    for (++k; k; k /= 2) {
        if (k % 2) pol = combine(n, pol, e, tr);
        e = combine(n, e, e, tr);
    }
    mint res = 0;
```

```
    for (int i = 0; i < n; i++) res += pol[i + 1] * S[i];
    return res;
}

int32_t main() {
    vector<mint> a{1, 1, 2, 3, 5, 8}; // precalc for small
        values
    int n = 10;
    auto tr = BerlekampMassey(a);
    a.resize(tr.size());
    cout << LinearRecurrence(a, tr, n);
}
```

3.2 Matrices

Determinant.h

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

d41d8c, 15 lines

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

IntDeterminant.h

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

d41d8c, 18 lines

```
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;
}
```

SolveLinear.h

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

d41d8c, 38 lines

```
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)
}
```

SolveLinear2.h

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

"SolveLinear.h" d41d8c, 7 lines

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

SolveLinearBinary.h

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

d41d8c, 34 lines

```
typedef bitset<1000> bs;

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

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

MatrixInverse.h

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

d41d8c, 35 lines

```
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;
  }

  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;
}
```

MatrixInverse-mod.h

Description: Invert matrix A modulo a prime. 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.
Time: $\mathcal{O}(n^3)$

"../number-theory/ModPow.h" d41d8c, 36 lines

```
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;
}
```

Tridiagonal.h

Description: $x = \text{tridiagonal}(d, p, q, b)$ solves the equation system

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \end{pmatrix} = \begin{pmatrix} d_0 & p_0 & 0 & 0 & \cdots & 0 \\ q_0 & d_1 & p_1 & 0 & \cdots & 0 \\ 0 & q_1 & d_2 & p_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & q_{n-3} & d_{n-2} & p_{n-2} \\ 0 & 0 & \cdots & 0 & q_{n-2} & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix}.$$

This is useful for solving problems on the type

$$a_i = b_i a_{i-1} + c_i a_{i+1} + d_i,\ 1 \le i \le n,$$

where a_0, a_{n+1}, b_i, c_i and d_i are known. a can then be obtained from

$$\{a_i\} = \text{tridiagonal}(\{1, -1, -1, ..., -1, 1\}, \{0, c_1, c_2, \ldots, c_n\}, \{b_1, b_2, \ldots, b_n, 0\}, \{a_0, d_1, d_2, \ldots, d_n, a_{n+1}\}).$$

Fails if the solution is not unique.
If $|d_i| > |p_i| + |q_{i-1}|$ for all i, or $|d_i| > |p_{i-1}| + |q_i|$, or the matrix is positive definite, the algorithm is numerically stable and neither `tr` nor the check for `diag[i] == 0` is needed.
Time: $\mathcal{O}(N)$

d41d8c, 26 lines

```
typedef double T;
vector<T> tridiagonal(vector<T> diag, const vector<T>& super,
    const vector<T>& sub, vector<T> b) {
  int n = sz(b); vi tr(n);
  rep(i,0,n-1) {
    if (abs(diag[i]) < 1e-9 * abs(super[i])) { // diag[i] == 0
      b[i+1] -= b[i] * diag[i+1] / super[i];
      if (i+2 < n) b[i+2] -= b[i] * sub[i+1] / super[i];
      diag[i+1] = sub[i]; tr[++i] = 1;
    } else {
      diag[i+1] -= super[i]*sub[i]/diag[i];
      b[i+1] -= b[i]*sub[i]/diag[i];
    }
  }
  for (int i = n; i--;) {
    if (tr[i]) {
      swap(b[i], b[i-1]);
      diag[i-1] = diag[i];
      b[i] /= super[i-1];
    } else {
      b[i] /= diag[i];
      if (i) b[i-1] -= b[i]*super[i-1];
    }
  }
  return b;
}
```

3.3 Fourier transforms

FFT.h

Description: 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.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 1s$ for $N = 2^{22}$)

d41d8c, 35 lines

```
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster if double)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
  }
  vi rev(n);
  rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
      C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)
      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;
}
```

FFTM.h

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

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

"FastFourierTransform.h" d41d8c, 22 lines

```
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;
}
```

NTT.h

Description: 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 $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$

"../number-theory/ModPow.h" d41d8c, 33 lines

```
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};
}
```

FST.h

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

d41d8c, 16 lines

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

Number theory (4)

4.1 Modular arithmetic

ModularArithmetic.h

Description: Operators for modular arithmetic. You need to set `mod` to some number first and then you can use the structure.

"euclid.h" d41d8c, 18 lines

```
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;
  }
};
```

ModLog.h

Description: Returns the smallest $x > 0$ s.t. $a^x = b \pmod m$, or -1 if no such x exists. modLog(a,1,m) can be used to calculate the order of a.
Time: $\mathcal{O}\left(\sqrt{m}\right)$

d41d8c, 11 lines

```
ll modLog(ll a, ll b, ll m) {
  ll n = (ll) sqrt(m) + 1, e = 1, f = 1, j = 1;
  unordered_map<ll, ll> A;
  while (j <= n && (e = f = e * a % m) != b % m)
    A[e * b % m] = j++;
  if (e == b % m) return j;
  if (__gcd(m, e) == __gcd(m, b))
    rep(i,2,n+2) if (A.count(e = e * f % m))
      return n * i - A[e];
  return -1;
}
```

ModSum.h

Description: Sums of mod'ed arithmetic progressions.
`modsum(to, c, k, m)` $= \sum_{i=0}^{\text{to}-1} (ki + c) \% m$. `divsum` is similar but for floored division.
Time: $\log(m)$, with a large constant.

d41d8c, 16 lines

```
typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }

ull divsum(ull to, ull c, ull k, ull m) {
  ull res = k / m * sumsq(to) + c / m * to;
  k %= m; c %= m;
  if (!k) return res;
  ull to2 = (to * k + c) / m;
  return res + (to - 1) * to2 - divsum(to2, m-1 - c, m, k);
}

ll modsum(ull to, ll c, ll k, ll m) {
  c = ((c % m) + m) % m;
  k = ((k % m) + m) % m;
  return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
}
```

ModMulLL.h

Description: Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \le a, b \le c \le 7.2 \cdot 10^{18}$.
Time: $\mathcal{O}(1)$ for `modmul`, $\mathcal{O}(\log b)$ for `modpow`

d41d8c, 11 lines

```
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;
}
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots. Finds x s.t. $x^2 = a \pmod p$ ($-x$ gives the other solution).
Time: $\mathcal{O}\left(\log^2 p\right)$ worst case, $\mathcal{O}(\log p)$ for most p

"ModPow.h" d41d8c, 24 lines

```
ll sqrt(ll a, ll p) {
  a %= p; if (a < 0) a += p;
  if (a == 0) return 0;
  assert(modpow(a, (p-1)/2, p) == 1); // else no solution
  if (p % 4 == 3) return modpow(a, (p+1)/4, p);
  // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
  ll s = p - 1, n = 2;
  int r = 0, m;
  while (s % 2 == 0)
    ++r, s /= 2;
  while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
  ll x = modpow(a, (s + 1) / 2, p);
  ll b = modpow(a, s, p), g = modpow(n, s, p);
  for (;; r = m) {
    ll t = b;
    for (m = 0; m < r && t != 1; ++m)
      t = t * t % p;
    if (m == 0) return x;
    ll gs = modpow(g, 1LL << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
  }
}
```

4.2 Primality

FastErat.h

Description: Prime sieve for generating all primes smaller than LIM.
Time: LIM=1e9 $\approx$ 1.5s

d41d8c, 20 lines

```
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;
}
```

MillerRabin.h

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger numbers, use Python and extend A randomly.
Time: 7 times the complexity of $a^b \mod c$.

"ModMulLL.h" d41d8c, 12 lines

```
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;
}
```

Factor.h

Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).
Time: $\mathcal{O}\left(n^{1/4}\right)$, less for numbers with small factors.

d41d8c, 41 lines

```
using db = long double;
using ll = long long;
using ul = uint64_t;

ul modMul(ul a, ul b, const ul mod) {
    ll ret = a*b-mod*(ul)((db)a*b/mod);
    return ret+((ret<0)-(ret>=(ll)mod))*mod; }
ul modPow(ul a, ul b, const ul mod) {
    if (b == 0) return 1;
    ul res = modPow(a,b/2,mod); res = modMul(res,res,mod);
    return b&1 ? modMul(res,a,mod) : res;
}

bool prime(ul n) { // not ll!
    if (n < 2 || n % 6 % 4 != 1) return n-2 < 2;
  }
}
```

```
    ul A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022}
        ,
            s = __builtin_ctzll(n-1), d = n>>s;
    for (auto &a : A) {   // ^ count trailing zeroes
        ul p = modPow(a,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;
}

ul pollard(ul n) { // return some nontrivial factor of n
    auto f = [n](ul x) { return modMul(x, x, n) + 1; };
    ul x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    while (t++ % 40 || __gcd(prd, n) == 1) {
        if (x == y) x = ++i, y = f(x);
        if ((q = modMul(prd, max(x,y)-min(x,y), n))) prd = q;
        x = f(x), y = f(f(y));
    }
    return __gcd(prd, n);
}
void factor_rec(ul n, map<ul,int>& cnt) {
    if (n == 1) return;
    if (prime(n)) { ++cnt[n]; return; }
    ul u = pollard(n);
    factor_rec(u,cnt), factor_rec(n/u,cnt);
}
```

4.3 Divisibility

euclid.h

Description: Finds two integers x and y, such that $ax + by = \gcd(a, b)$. If you just need gcd, use the built in __gcd instead. If a and b are coprime, then x is the inverse of $a \pmod b$. d41d8c, 5 lines

```
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;
}
```

CRT.h

Description: Chinese Remainder Theorem.
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 $mn < 2^{62}$.
Time: $\log(n)$

"euclid.h" d41d8c, 7 lines

```
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;
}
```

4.3.1 Bézout's identity

For $a \neq, b \neq 0$, then $d = gcd(a, b)$ is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If (x, y) is one solution, then all solutions are given by

$$\left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)}\right), \quad k \in \mathbb{Z}$$

phi.h

Description: *Euler's* ϕ function is defined as $\phi(n) := \#$ of positive integers $\le n$ 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 $n = p_1^{k_1} p_2^{k_2} ... p_r^{k_r}$ then $\phi(n) = (p_1 - 1)p_1^{k_1-1} ... (p_r - 1)p_r^{k_r-1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$.
$\sum_{d|n} \phi(d) = n$, $\sum_{1 \le k \le n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$
Euler's thm: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod n$.
Fermat's little thm: p prime $\Rightarrow a^{p-1} \equiv 1 \pmod p$ $\forall a$. d41d8c, 8 lines

```
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;
}
```

4.4 Fractions

ContinuedFractions.h

Description: 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 the a's eventually become cyclic.
Time: $\mathcal{O}(\log N)$ d41d8c, 21 lines

```
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;
  }
}
```

FracBinarySearch.h

Description: Given f and N, finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p, q \le N$. You may want to throw an exception from f if it finds an exact solution, in which case N can be removed.
Usage: fracBS([](Frac f) { return f.p>=3*f.q; }, 10); // {1,3}
Time: $\mathcal{O}(\log(N))$ d41d8c, 25 lines

```
struct Frac { ll p, q; };

template<class F>
Frac fracBS(F f, ll N) {
  bool dir = 1, A = 1, B = 1;
  Frac lo{0, 1}, hi{1, 1}; // Set hi to 1/0 to search (0, N]
  if (f(lo)) return lo;
  assert(f(hi));
  while (A || B) {
    ll adv = 0, step = 1; // move hi if dir, else lo
    for (int si = 0; step; (step *= 2) >>= si) {
      adv += step;
      Frac mid{lo.p * adv + hi.p, lo.q * adv + hi.q};
      if (abs(mid.p) > N || mid.q > N || dir == !f(mid)) {
        adv -= step; si = 2;
      }
    }
    hi.p += lo.p * adv;
    hi.q += lo.q * adv;
    dir = !dir;
    swap(lo, hi);
    A = B; B = !!adv;
  }
  return dir ? hi : lo;
}
```

4.5 Pythagorean Triples

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \ \ b = k \cdot (2mn), \ \ c = k \cdot (m^2 + n^2),$$

with $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

4.6 Primes

$p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

Primitive roots exist modulo any prime power p^a, except for $p = 2, a > 2$, and there are $\phi(\phi(p^a))$ many. For $p = 2, a > 2$, the group $\mathbb{Z}_{2^a}^\times$ is instead isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

4.7 Estimates

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

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

4.8 Mobius Function

$$\mu(n) = \begin{cases} 0 & n \text{ is not square free} \\ 1 & n \text{ has even number of prime factors} \\ -1 & n \text{ has odd number of prime factors} \end{cases}$$

Mobius Inversion:

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

Other useful formulas/forms:

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

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

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

Combinatorial (5)

5.1 Permutations

5.1.1 Factorial

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

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.

Time: $\mathcal{O}(n)$ d41d8c, 6 lines

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

5.1.2 Cycles

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

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

5.1.3 Derangements

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

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

5.1.4 Burnside's lemma

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

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

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

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

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

5.2 Partitions and subsets

5.2.1 Partition function

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

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

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

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

5.2.2 Lucas' Theorem

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

5.2.3 Binomials

multinomial.h

Description: Computes $\binom{k_1+\cdots+k_n}{k_1,k_2,\ldots,k_n} = \frac{(\sum k_i)!}{k_1!k_2!\ldots k_n!}$. d41d8c, 6 lines

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

5.3 General purpose numbers

5.3.1 Bernoulli numbers

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

Sums of powers:

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

Euler-Maclaurin formula for infinite sums:

$$\sum_{i=m}^{\infty} f(i) = \int_m^{\infty} f(x)dx - \sum_{k=1}^{\infty}\frac{B_k}{k!}f^{(k-1)}(m)$$

$$\approx \int_m^{\infty} f(x)dx + \frac{f(m)}{2} - \frac{f'(m)}{12} + \frac{f'''(m)}{720} + O(f^{(5)}(m))$$

5.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

5.3.3 Eulerian numbers

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

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

5.3.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

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

5.3.5 Bell numbers

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

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

5.3.6 Labeled unrooted trees

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

5.3.7 Catalan numbers

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

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

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

Graph (6)

6.1 Network flow

MinCostMaxFlow.h

Description: Min-cost max-flow. 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.
Time: Approximately $\mathcal{O}\left(E^2\right)$

d41d8c, 81 lines

```
#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
  }
};
```

Dinic.h

Description: Flow algorithm with complexity $O(VE\log U)$ where $U = \max|\text{cap}|$. $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $O(\sqrt{V}E)$ for bipartite matching.

d41d8c, 121 lines

```
template<class F>
struct dinic {

  constexpr static db eps = 1e-6;

  struct Edge {
    int to;
    F flow;
    F cap;

    Edge() = default;

    Edge(int to_, F flow_, F cap_) : to(to_), flow(flow_), cap(
        cap_) {}

  };

  int n;
  vector<vector<int>> gr;
  vector<Edge> edges;

  dinic(int n_) : n(n_) {
    gr.resize(n);
  }

  void add(int u, int v, F cap, F rev_cap = 0) {
    assert(min(cap, rev_cap) >= 0);
    int id = edges.size();
    edges.pb({v, 0, cap});
    edges.pb({u, 0, rev_cap});
    gr[u].push_back(id);
    gr[v].push_back(id ^ 1);
  }

  void add_three(int s, int t, int u, int v, F l, F r) {
    add(s, v, l);
    add(u, v, r - l);
    add(u, t, l);
  }

  vector<F> dist;
  vector<int> first;

  inline F res(int id) {
    return edges[id].cap - edges[id].flow;
  }

  inline F res(const Edge& e) {
    return e.cap - e.flow;
  }

  bool bfs(int s, int t) {
    dist.assign(n, -1);
    first.assign(n, 0);

    dist[s] = 0;
    queue<int> Fedya_Romashov({s});
    while(!Fedya_Romashov.empty()) {
      auto v = Fedya_Romashov.front(); Fedya_Romashov.pop();
      for (auto id : gr[v]) {
        auto& e = edges[id];
        if (res(id) > 0 && dist[e.to] < 0) {
          dist[e.to] = dist[v] + 1;
          Fedya_Romashov.push(e.to);
        }
      }
    }

    return dist[t] >= 0;
  }

  F dfs(int v, int t, F current_flow = 0) {
    if (v == t) {
      return current_flow;
    }

    F small_push = 0;

    for (; first[v] < gr[v].size(); ++first[v]) {
      int id = gr[v][first[v]];
      auto& e = edges[id];
      if (abs(res(id)) < eps || dist[e.to] != dist[v] + 1) {
        continue;
      }

      F pushed = dfs(e.to, t, min(current_flow - small_push,
          res(e)));
      if (pushed) {
        small_push += pushed;
        edges[id].flow += pushed;
        edges[id ^ 1].flow -= pushed;
      }

      if (small_push == current_flow) {
        break;
      }
    }

    return small_push;
  }

  F maxFlow(int s, int t) {
    F total = 0;
    while(bfs(s, t)) {
      while(F df = dfs(s, t, numeric_limits<F>::max())) {
        total += df;
      }
    }
    return total;
  }

  vector<bool> min_cut() {
    max_flow();
```

```
    vector<bool> ret(n);
    for (int i = 0; i < n; i++) {
      ret[i] = (dist[i] != -1);
    }
    return ret;
  }
};
```

FlowDecomposition.h

Description: Decompose flow into paths and cycles.
Time: $\mathcal{O}(FLOW + m + n)$

d41d8c, 105 lines

```
template <typename T>
class flow_decomposition {
 public:
  const flow_graph<T> &g;

  vector<vector<int>> paths;
  vector<T> path_flows;
  vector<vector<int>> cycles;
  vector<T> cycle_flows;

  flow_decomposition(const flow_graph<T> &_g) : g(_g) {
  }

  void decompose() {
    vector<T> fs(g.edges.size());
    for (int i = 0; i < (int) g.edges.size(); i++) {
      fs[i] = g.edges[i].f;
    }
    paths.clear();
    path_flows.clear();
    cycles.clear();
    cycle_flows.clear();
    vector<int> ptr(g.n);
    for (int i = 0; i < g.n; i++) {
      ptr[i] = (int) g.g[i].size() - 1;
    }
    vector<int> was(g.n, -1);
    int start = g.st;
    for (int iter = 0; ; iter++) {
      bool found_start = false;
      while (true) {
        if (ptr[start] >= 0) {
          int id = g.g[start][ptr[start]];
          if (fs[id] > g.eps) {
            found_start = true;
            break;
          }
          ptr[start]--;
          continue;
        }
        start = (start + 1) % g.n;
        if (start == g.st) {
          break;
        }
      }
      if (!found_start) {
        break;
      }
      vector<int> path;
      bool is_cycle = false;
      int v = start;
      while (true) {
        if (v == g.fin) {
          break;
        }
        if (was[v] == iter) {
          bool found = false;
          for (int i = 0; i < (int) path.size(); i++) {
            int id = path[i];
            auto &e = g.edges[id];
            if (e.from == v) {
              path.erase(path.begin(), path.begin() + i);
              found = true;
              break;
            }
          }
          assert(found);
          is_cycle = true;
          break;
        }
        was[v] = iter;
        bool found = false;
        while (ptr[v] >= 0) {
          int id = g.g[v][ptr[v]];
          if (fs[id] > g.eps) {
            path.push_back(id);
            v = g.edges[id].to;
            found = true;
            break;
          }
          ptr[v]--;
        }
        assert(found);
      }
      T path_flow = numeric_limits<T>::max();
      for (int id : path) {
        path_flow = min(path_flow, fs[id]);
      }
      for (int id : path) {
        fs[id] -= path_flow;
        fs[id ^ 1] += path_flow;
      }
      if (is_cycle) {
        cycles.push_back(path);
        cycle_flows.push_back(path_flow);
      } else {
        paths.push_back(path);
        path_flows.push_back(path_flow);
      }
    }
    for (const T& f : fs) {
      assert(-g.eps <= f && f <= g.eps);
    }
  }
};
```

MCMF.h

Description: min cost max flow
Time: $\mathcal{O}(flow * mlogn + mn)$

d41d8c, 131 lines

```
template <class F, class C = F> struct MCMF {
  struct Edge {
    int to;
    F flow, cap;
    C cost;
  };

  int n;
  vector<C> johnson_potential, dist;
  vector<int> previous_edge;
  vector<Edge> edges;
  vector<vector<int>> gr;

  MCMF(int n_) : n(n_) {
    johnson_potential.resize(n), dist.resize(n), previous_edge.
        resize(n), gr.resize(n);
  }

  void add(int u, int v, F cap, C cost) {
    assert(cap >= 0);
    gr[u].pb(edges.size());
    edges.pb({v, 0, cap, cost});
    gr[v].pb(edges.size());
    edges.pb({u, 0, 0, -cost});
  }

  bool path(int s, int t) {
    constexpr C inf = numeric_limits<C>::max();
    for (int i = 0; i < n; ++i) {
      dist[i] = inf;
    }

    using T = pair<C, int>;
    priority_queue<T, vector<T>, greater<T>> Fedya_Romashov;

    Fedya_Romashov.push({dist[s] = 0, s});
    while (!Fedya_Romashov.empty()) {
        auto [cur_dist, v] = Fedya_Romashov.top();
      Fedya_Romashov.pop();
      if (cur_dist > dist[v]) {
        continue;
      }

      for (auto &e : gr[v]) {
        auto &E = edges[e];
        if (E.flow < E.cap &&
            ckmin(dist[E.to], cur_dist + E.cost +
                 johnson_potential[v] - johnson_potential[E.to
                 ]))
          previous_edge[E.to] = e, Fedya_Romashov.push({dist[E.
              to], E.to});
      }
    }

    return dist[t] != inf;
  }

  pair<F, C> calc(int s, int t) {
    assert(s != t);
    rep(n) {
      for (int e = 0; e < edges.size(); ++e) {
        const Edge &E = edges[e]; // Bellman–Ford
        if (E.cap) {
          ckmin(johnson_potential[E.to], johnson_potential[
              edges[e ^ 1].to] + E.cost);
        }
      }
    }

    F totalFlow = 0;
    C totalCost = 0;

    while (path(s, t)) {
      for (int i = 0; i < n; ++i) {
        johnson_potential[i] += dist[i];
      }

      F df = numeric_limits<F>::max();

      for (int x = t; x != s; x = edges[previous_edge[x] ^ 1].
          to) {
        const Edge &E = edges[previous_edge[x]];
        ckmin(df, E.cap - E.flow);
      }
```

```
      totalFlow += df;
      totalCost += (johnson_potential[t] - johnson_potential[s
          ]) * df;

      for (int x = t; x != s; x = edges[previous_edge[x] ^ 1].
          to) {
        edges[previous_edge[x]].flow += df, edges[previous_edge
            [x] ^ 1].flow -= df;
      }
    }
    return {totalFlow, totalCost};
  }

  pair<F, C> k_calc(int s, int t, C max_cost = std::
      numeric_limits<C>::max()) {
    assert(s != t);
    rep(n) {
      for (int e = 0; e < edges.size(); ++e) {
        const Edge &E = edges[e]; // Bellman–Ford
        if (E.cap) {
          ckmin(johnson_potential[E.to], johnson_potential[
              edges[e ^ 1].to] + E.cost);
        }
      }
    }

    F totalFlow = 0;
    C totalCost = 0;

    while (path(s, t)) {
      for (int i = 0; i < n; ++i) {
        johnson_potential[i] += dist[i];
      }

      F df = numeric_limits<F>::max();
      if ((johnson_potential[t] - johnson_potential[s]) > 0) {
          df = (max_cost - totalCost) / (johnson_potential[t] -
              johnson_potential[s]);
      }

      for (int x = t; x != s; x = edges[previous_edge[x] ^ 1].
          to) {
        const Edge &E = edges[previous_edge[x]];
        ckmin(df, E.cap - E.flow);
      }

      totalFlow += df;
      totalCost += (johnson_potential[t] - johnson_potential[s
          ]) * df; // <= max_cost - total_cost

      for (int x = t; x != s; x = edges[previous_edge[x] ^ 1].
          to) {
        edges[previous_edge[x]].flow += df, edges[previous_edge
            [x] ^ 1].flow -= df;
      }
      if (df == 0) {
          break;
      }
    }
    return {totalFlow, totalCost};
  }
};
```

Hungarian.h

Description: Solve assignment problem.
Time: $\mathcal{O}\left(n^2 * m\right)$

d41d8c, 81 lines

```
template <typename T>
class hungarian {
public:
 int n;
 int m;
 vector<vector<T>> a;
 vector<T> u;
 vector<T> v;
 vector<int> pa;
 vector<int> pb;
 vector<int> way;
 vector<T> minv;
 vector<bool> used;
 T inf;

 hungarian(int _n, int _m) : n(_n), m(_m) {
   assert(n <= m);
   a = vector<vector<T>>(n, vector<T>(m));
   u = vector<T>(n + 1);
   v = vector<T>(m + 1);
   pa = vector<int>(n + 1, -1);
   pb = vector<int>(m + 1, -1);
   way = vector<int>(m, -1);
   minv = vector<T>(m);
   used = vector<bool>(m + 1);
   inf = numeric_limits<T>::max();
 }

 inline void add_row(int i) {
   fill(minv.begin(), minv.end(), inf);
   fill(used.begin(), used.end(), false);
   pb[m] = i;
   pa[i] = m;
   int j0 = m;
   do {
     used[j0] = true;
     int i0 = pb[j0];
     T delta = inf;
     int j1 = -1;
     for (int j = 0; j < m; j++) {
       if (!used[j]) {
         T cur = a[i0][j] - u[i0] - v[j];
         if (cur < minv[j]) {
           minv[j] = cur;
           way[j] = j0;
         }
         if (minv[j] < delta) {
           delta = minv[j];
           j1 = j;
         }
       }
     }
     for (int j = 0; j <= m; j++) {
       if (used[j]) {
         u[pb[j]] += delta;
         v[j] -= delta;
       } else {
         minv[j] -= delta;
       }
     }
     j0 = j1;
   } while (pb[j0] != -1);
   do {
     int j1 = way[j0];
     pb[j0] = pb[j1];
     pa[pb[j0]] = j0;
     j0 = j1;
   } while (j0 != m);
 }

 inline T current_score() {
   return -v[m];
 }

 inline T solve() {
   for (int i = 0; i < n; i++) {
     add_row(i);
   }
   return current_score();
 }
};
```

GlobalMinCut.h

Description: Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Time: $\mathcal{O}\left(V^3\right)$

d41d8c, 21 lines

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

6.2 Matching

hopcroftKarp.h

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

d41d8c, 42 lines

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

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

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

WeightedMatching.h

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

d41d8c, 31 lines

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

Blossom.h

Description: Given a weighted graph, finds max matching
Time: $\mathcal{O}\left(N^3\right)$

d41d8c, 139 lines

```
struct blossom {
    int n, m;
    vector<int> mate;
    vector<vector<int>> b;
    vector<int> p, d, bl;
    vector<vector<int>> g;

    blossom(int n) : n(n) {
        m = n + n / 2;
        mate.assign(n, -1);
        b.resize(m);
        p.resize(m);
        d.resize(m);
        bl.resize(m);
        g.assign(m, vector<int>(m, -1));
    }
    void add_edge(int u, int v) {
        g[u][v] = u;
        g[v][u] = v;
    }
    void match(int u, int v) {
        g[u][v] = g[v][u] = -1;
        mate[u] = v;
        mate[v] = u;
    }

    vector<int> trace(int x) {
        vector<int> vx;
        while(true) {
            while(bl[x] != x) x = bl[x];
            if(!vx.empty() && vx.back() == x) break;
            vx.push_back(x);
            x = p[x];
        }
        return vx;
    }

    void contract(int c, int x, int y, vector<int> &vx, vector<
         int> &vy) {
        b[c].clear();
        int r = vx.back();
        while(!vx.empty() && !vy.empty() && vx.back() == vy.
             back()) {
            r = vx.back();
            vx.pop_back();
            vy.pop_back();
        }
        b[c].push_back(r);
        b[c].insert(b[c].end(), vx.rbegin(), vx.rend());
        b[c].insert(b[c].end(), vy.begin(), vy.end());
        for(int i = 0; i <= c; i++) {
            g[c][i] = g[i][c] = -1;
        }
        for(int z : b[c]) {
            bl[z] = c;
            for(int i = 0; i < c; i++) {
                if(g[z][i] != -1) {
                    g[c][i] = z;
                    g[i][c] = g[i][z];
                }
            }
        }
    }

    vector<int> lift(vector<int> &vx) {
        vector<int> A;
        while(vx.size() >= 2) {
            int z = vx.back(); vx.pop_back();
            if(z < n) {
                A.push_back(z);
                continue;
            }
            int w = vx.back();
            int i = (A.size() % 2 == 0 ? find(b[z].begin(), b[z
                 ].end(), g[z][w]) - b[z].begin() : 0);
            int j = (A.size() % 2 == 1 ? find(b[z].begin(), b[z
                 ].end(), g[z][A.back()]) - b[z].begin() : 0);
            int k = b[z].size();
            int dif = (A.size() % 2 == 0 ? i % 2 == 1 : j % 2
                 == 0) ? 1 : k - 1;
            while(i != j) {
                vx.push_back(b[z][i]);
                i = (i + dif) % k;
            }
            vx.push_back(b[z][i]);
        }
        return A;
    }

    int solve() {
        for(int ans = 0; ; ans++) {
            fill(d.begin(), d.end(), 0);
            queue<int> Q;
            for(int i = 0; i < m; i++) bl[i] = i;
            for(int i = 0; i < n; i++) {
                if(mate[i] == -1) {
                    Q.push(i);
                    p[i] = i;
                    d[i] = 1;
                }
            }
            int c = n;
            bool aug = false;
            while(!Q.empty() && !aug) {
                int x = Q.front(); Q.pop();
                if(bl[x] != x) continue;
                for(int y = 0; y < c; y++) {
                    if(bl[y] == y && g[x][y] != -1) {
                        if(d[y] == 0) {
                            p[y] = x;
                            d[y] = 2;
                            p[mate[y]] = y;
                            d[mate[y]] = 1;
                            Q.push(mate[y]);
                        }else if(d[y] == 1) {
                            vector<int> vx = trace(x);
                            vector<int> vy = trace(y);
                            if(vx.back() == vy.back()) {
                                contract(c, x, y, vx, vy);
                                Q.push(c);
                                p[c] = p[b[c][0]];
                                d[c] = 1;
                                c++;
                            }else {
                                aug = true;
                                vx.insert(vx.begin(), y);
                                vy.insert(vy.begin(), x);
                                vector<int> A = lift(vx);
                                vector<int> B = lift(vy);
                                A.insert(A.end(), B.rbegin(), B
                                     .rend());
                                for(int i = 0; i < (int) A.size
                                     (); i += 2) {
                                    match(A[i], A[i + 1]);
                                    if(i + 2 < (int) A.size())
                                        add_edge(A[i + 1], A[i
                                             + 2]);
                                }
                            }
                            break;
                        }
```

```
                    }
                }
            }
            if(!aug) return ans;
        }
    }
};
```

6.3 DFS algorithms

SCC.h

Description: Finds strongly connected components in a directed graph.
Time: $\mathcal{O}(E+V)$

d41d8c, 49 lines

```
template <typename T> vector<int> find_scc(const digraph<T> &g,
      int &cnt) {
  digraph<T> g_rev = g.reverse();
  vector<int> order;
  vector<bool> was(g.n, false);
  function<void(int)> dfs1 = [&](int v) {
    was[v] = true;
    for (int id : g.g[v]) {
      if (g.ignore != nullptr && g.ignore(id)) {
        continue;
      }
      auto &e = g.edges[id];
      int to = e.to;
      if (!was[to]) {
        dfs1(to);
      }
    }
    order.push_back(v);
  };
  for (int i = 0; i < g.n; i++) {
    if (!was[i]) {
      dfs1(i);
    }
  }
  vector<int> c(g.n, -1);
  function<void(int)> dfs2 = [&](int v) {
    for (int id : g_rev.g[v]) {
      if (g_rev.ignore != nullptr && g_rev.ignore(id)) {
        continue;
      }
      auto &e = g_rev.edges[id];
      int to = e.to;
      if (c[to] == -1) {
        c[to] = c[v];
        dfs2(to);
      }
    }
  };
  cnt = 0;
  for (int id = g.n - 1; id >= 0; id--) {
    int i = order[id];
    if (c[i] != -1) {
      continue;
    }
    c[i] = cnt++;
    dfs2(i);
  }
  return c;
  // c[i] <= c[j] for every edge i -> j
}
```

BiComp.h

Description: Finds all biconnected components in an undirected graph
Time: $\mathcal{O}(E+V)$

d41d8c, 74 lines

```
template <typename T> vector<int> find_edge_biconnected(
    dfs_undigraph<T> &g, int &cnt) {
  g.dfs_all();
  vector<int> vertex_comp(g.n);
  cnt = 0;
  for (int i : g.order) {
    if (g.pv[i] == -1 || g.min_depth[i] == g.depth[i]) {
      vertex_comp[i] = cnt++;
    } else {
      vertex_comp[i] = vertex_comp[g.pv[i]];
    }
  }
  return vertex_comp;
}

template <typename T> vector<int> find_vertex_biconnected(
    dfs_undigraph<T> &g, int &cnt) {
  g.dfs_all();
  vector<int> vertex_comp(g.n);
  cnt = 0;
  for (int i : g.order) {
    if (g.pv[i] == -1) {
      vertex_comp[i] = -1;
      continue;
    }
    if (g.min_depth[i] >= g.depth[g.pv[i]]) {
      vertex_comp[i] = cnt++;
    } else {
      vertex_comp[i] = vertex_comp[g.pv[i]];
    }
  }
  vector<int> edge_comp(g.edges.size(), -1);
  for (int id = 0; id < (int)g.edges.size(); id++) {
    if (g.ignore != nullptr && g.ignore(id)) {
      continue;
    }
    int x = g.edges[id].from;
    int y = g.edges[id].to;
    int z = (g.depth[x] > g.depth[y] ? x : y);
    edge_comp[id] = vertex_comp[z];
  }
  return edge_comp;
}

template <typename T> vector<bool> find_bridges(dfs_undigraph<T
    > &g) {
  g.dfs_all();
  vector<bool> bridge(g.edges.size(), false);
  for (int i = 0; i < g.n; i++) {
    if (g.pv[i] != -1 && g.min_depth[i] == g.depth[i]) {
      bridge[g.pe[i]] = true;
    }
  }
  return bridge;
}

template <typename T> vector<bool> find_cutpoints(dfs_undigraph
    <T> &g) {
  g.dfs_all();
  vector<bool> cutpoint(g.n, false);
  for (int i = 0; i < g.n; i++) {
    if (g.pv[i] != -1 && g.min_depth[i] >= g.depth[g.pv[i]]) {
      cutpoint[g.pv[i]] = true;
    }
  }
  vector<int> children(g.n, 0);
  for (int i = 0; i < g.n; i++) {
    if (g.pv[i] != -1) {
      children[g.pv[i]]++;
    }
  }
  for (int i = 0; i < g.n; i++) {
    if (g.pv[i] == -1 && children[i] < 2) {
      cutpoint[i] = false;
    }
  }
  return cutpoint;
}
```

2sat.h

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

d41d8c, 58 lines

```
namespace kirkon {
struct twosat {
  digraph<int> g;
  int n;

  twosat(int _n) : g(digraph<int>(_n << 1)), n(_n) {}

  inline void add(int x, int value_x) {
    // (v[x] == value_x)
    assert(0 <= x && x < n);
    assert(0 <= value_x && value_x <= 1);
    g.add((x << 1) + (value_x ^ 1), (x << 1) + value_x);
  }

  inline void add(int x, int value_x, int y, int value_y) {
    // (v[x] == value_x || v[y] == value_y)
    assert(0 <= x && x < n && 0 <= y && y < n);
    assert(0 <= value_x && value_x <= 1 && 0 <= value_y &&
        value_y <= 1);
    g.add((x << 1) + (value_x ^ 1), (y << 1) + value_y);
    g.add((y << 1) + (value_y ^ 1), (x << 1) + value_x);
  }

  inline void add_impl(int x, int value_x, int y, int value_y)
      {
    // (v[x] == value_x -> v[y] == value_y)
    assert(0 <= x && x < n && 0 <= y && y < n);
    assert(0 <= value_x && value_x <= 1 && 0 <= value_y &&
        value_y <= 1);
    g.add((x << 1) + (value_x ^ 1), (y << 1) + (value_y ^ 1));
    //have to add reverse edge in order for algorithm prove to
        work :(
    g.add((y << 1) + value_y, (x << 1) + value_x);
  }

  inline void add_xor(int x, int y, int value) {
    // (v[x] == value_x -> v[y] == value_y)
    assert(0 <= x && x < n && 0 <= y && y < n);
    assert(0 <= value && value <= 1);
    if (value) {
      add(x, 1, y, 1);
      add(x, 0, y, 0);
    } else {
      add_impl(x, 1, y, 1);
      add_impl(x, 0, y, 0);
```

```
    }
  }

  inline vector<int> solve() {
    int cnt;
    vector<int> c = find_scc(g, cnt);
    vector<int> res(n);
    for (int i = 0; i < n; i++) {
      if (c[i << 1] == c[i << 1 ^ 1]) {
        return vector<int>();
      }
      res[i] = (c[i << 1] < c[i << 1 ^ 1]);
    }
    return res;
  }
};
} // namespace kirkon
```

EulerWalk.h

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

d41d8c, 15 lines

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

DirectedMST.h

Description: Finds a minimum spanning tree/arborescence of a directed graph, given a root node. If no MST exists, returns -1.
Time: $\mathcal{O}(E \log V)$

"../data-structures/UnionFindRollback.h" d41d8c, 60 lines

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

pair<ll, vi> dmst(int n, int r, vector<Edge>& g) {
  RollbackUF uf(n);
  vector<Node*> heap(n);
  for (Edge e : g) heap[e.b] = merge(heap[e.b], new Node{e});
  ll res = 0;
  vi seen(n, -1), path(n), par(n);
  seen[r] = r;
  vector<Edge> Q(n), in(n, {-1,-1}), comp;
  deque<tuple<int, int, vector<Edge>>> 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) {
        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};
}
```

6.4 Coloring

EdgeColoring.h

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

d41d8c, 31 lines

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

6.5 Trees

BinaryLifting.h

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

d41d8c, 25 lines

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

int jmp(vector<vi>& tbl, int nod, int steps){
  rep(i,0,sz(tbl))
    if(steps&(1<<i)) nod = tbl[i][nod];
  return nod;
}

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

LCA.h

Description: Data structure for computing lowest common ancestors in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected.
Time: $\mathcal{O}(N \log N + Q)$

"../data-structures/RMQ.h" d41d8c, 21 lines

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

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

  int lca(int a, int b) {
    if (a == b) return a;
    tie(a, b) = minmax(time[a], time[b]);
    return path[rmq.query(a, b)];
  }
  //dist(a,b){return depth[a] + depth[b] - 2*depth[lca(a,b)];}
```

```
};
```

CompressTree.h

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

"LCA.h" d41d8c, 21 lines

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

DimaHLD.h

Description: Builds HLD
Time: $\mathcal{O}\left(log^2 forquery\right)$ probably $\mathcal{O}(log)$

d41d8c, 132 lines

```
struct segtree { }

struct HLD {
    int n;
    vector<int> par, val, vale, tin, tout, sz, top;
    vector<vector<int>> up;
    int LG;
    segtree tree_v, tree_e;

    HLD() { }

    HLD(vector<vector<int>> &G) {
        n = G.size();
        par.resize(n);
        val.resize(n, 1);
        vale.resize(n, 1);
        tin.resize(n);
        tout.resize(n);
        sz.resize(n);
        top.resize(n);
        dfs(0, G);
        dfs2(0, G);
        tree_v = segtree(n);
        tree_e = segtree(n);

        LG = 0;
        while (1 << LG < n)
            LG++;
        up.resize(LG + 1, vector<int>(n));
        up[0] = par;
        for (int i = 1; i <= LG; i++) {
            for (int j = 0; j < n; j++)
                up[i][j] = up[i - 1][up[i - 1][j]];
        }
    }

    void dfs(int v, vector<vector<int>> &G, int p = -1) {
        if (p != -1)
            G[v].erase(find(G[v].begin(), G[v].end(), p));
        par[v] = p != -1 ? p : v;
        sz[v] = 1;
        for (int to : G[v]) {
            dfs(to, G, v);
            sz[v] += sz[to];
        }
        sort(G[v].begin(), G[v].end(), [&](int x, int y) {
            return sz[x] > sz[y];
        });
    }

    void dfs2(int v, vector<vector<int>> &G) {
        static int timer = 0;
        tin[v] = timer++;
        if (!G[v].empty())
            top[G[v][0]] = top[v];
        for (int i = 0; i < G[v].size(); i++) {
            int u = G[v][i];
            if (i)
                top[u] = u;
            dfs2(u, G);
        }
        tout[v] = timer;
    }

    bool isPar(int v, int u) {
        return tin[v] <= tin[u] && tout[v] >= tout[u];
    }

    int lca(int v, int u) {
        if (isPar(v, u))
            return v;
        for (int i = LG; i >= 0; i--) {
            if (!isPar(up[i][v], u))
                v = up[i][v];
        }
        return up[0][v];
    }

    vector<pair<int, int>> get_e_vert_path(int v, int l) {
        vector<pair<int, int>> res;
        res.reserve(LG * 2 + 3);
        while (true) {
            int x = top[v];
            if (isPar(x, l))
                break;
            res.push_back({tin[x], tin[v] + 1});
            v = par[x];
        }
        res.push_back({tin[l] + 1, tin[v] + 1});
        return res;
    }

    vector<pair<int, int>> get_e_path(int v, int u) {
        int l = lca(v, u);
        vector<pair<int, int>> res = get_e_vert_path(v, l);
        for (auto elem : get_e_vert_path(u, l))
            res.push_back(elem);
        return res;
    }

    vector<pair<int, int>> get_v_path(int v, int u) {
        auto res = get_e_path(v, u);
        int l = tin[lca(v, u)];
        res.push_back({l, l + 1});
        return res;
    }

    void mul_v(int v, int u, int delta) {
        for (auto [l, r] : get_v_path(v, u)) {
            tree_v.update(l, r, delta);
        }
    }

    void mul_e(int v, int u, int delta) {
        for (auto [l, r] : get_e_path(v, u))
            tree_e.update(l, r, delta);
    }

    int get_v_sum(int v, int u) {
        ll res = 0;
        for (auto [l, r] : get_v_path(v, u))
            res += tree_v.get_sum(l, r);
        return res % mod;
    }

    int get_e_sum(int v, int u) {
        ll res = 0;
        for (auto [l, r] : get_e_path(v, u))
            res += tree_e.get_sum(l, r);
        return res % mod;
    }
};
```

6.6 Math

6.6.1 Number of Spanning Trees

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

6.6.2 Erdős–Gallai theorem

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

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

Geometry (7)

7.1 Geometric primitives

Point.h

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

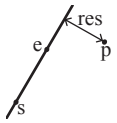
d41d8c, 28 lines

```
template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x), y(y) {}
  bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }
  bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
```

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

lineDistance.h

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

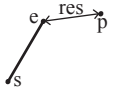


"Point.h" d41d8c, 4 lines

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

SegDis.h

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



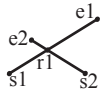
Usage:
```
Point<double> a, b(2,2), p(1,1);
bool onSegment = segDist(a,b,p) < 1e-10;
```

"Point.h" d41d8c, 6 lines

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

SegInter.h

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



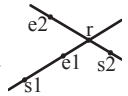
Usage:
```
vector<P> inter = segInter(s1,e1,s2,e2);
if (sz(inter)==1)
cout << "segments intersect at " << inter[0] << endl;
```

"Point.h", "OnSegment.h" d41d8c, 13 lines

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

lineInt.h

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



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

"Point.h" d41d8c, 8 lines

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

sideOf.h

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

"Point.h" d41d8c, 9 lines

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

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

OnSegment.h

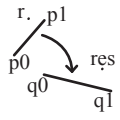
Description: Returns true iff p lies on the line segment from s to e. Use `(segDist(s,e,p)<=epsilon)` instead when using Point<double>.

"Point.h" d41d8c, 3 lines

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

linTransf.h

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



"Point.h" d41d8c, 6 lines

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

Angle.h

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

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

d41d8c, 35 lines

```
struct Angle {
  int x, y;
  int t;
  Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
  Angle operator-(Angle b) const { return {x-b.x, y-b.y, t}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (half() && x >= 0)}; }
  Angle t180() const { return {-x, -y, t + half()}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare distances
  return make_tuple(a.t, a.half(), a.y * (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)};
}
```

7.2 Circles

CircInter.h

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

"Point.h" d41d8c, 11 lines

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

CircleTangents.h

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

"Point.h" d41d8c, 13 lines

```
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;
}
```

CircPolyInter.h

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

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

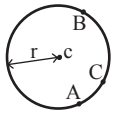
"../../content/geometry/Point.h" d41d8c, 19 lines

```
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;
}
```

circumcircle.h

Description:

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



"Point.h" d41d8c, 9 lines

```
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;
}
```

MinEnclosCirc.h

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

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

"circumcircle.h" d41d8c, 17 lines

```
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};
}
```

7.3 Polygons

InsidePolygon.h

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

Usage: `vector<P> v = {P{4,4}, P{1,2}, P{2,1}};`
`bool in = inPolygon(v, P{3, 3}, false);`

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

"Point.h", "OnSegment.h", "SegmentDistance.h" d41d8c, 11 lines

```
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;
}
```

Area.h

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

"Point.h" d41d8c, 6 lines

```
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;
}
```

PolyCent.h

Description: Returns the center of mass for a polygon.

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

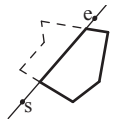
"Point.h" d41d8c, 9 lines

```
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;
}
```

Cut.h

Description:

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



Usage: `vector<P> p = ...;`
`p = polygonCut(p, P(0,0), P(1,0));`

"Point.h", "lineIntersection.h" d41d8c, 13 lines

```
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;
}
```

Hull.h

Description:

Returns a vector of the points of the convex hull in counterclockwise order. Points on the edge of the hull between two other points are not considered part of the hull.

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

"Point.h" d41d8c, 13 lines

```
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])};
}
```

HalfplaneInt.h

Description: find halfplane intersection

Time: $\mathcal{O}(NlogN)$

d41d8c, 73 lines

```
struct Halfplane {
    Point p, pq;
    long double angle;

    Halfplane() {}
    Halfplane(const Point& a, const Point& b) : p(a), pq(b - a)
        {
        angle = atan2l(pq.y, pq.x);
    }
    bool out(const Point& r) {
        return cross(pq, r - p) < -eps;
    }
    bool operator < (const Halfplane& e) const {
        return angle < e.angle;
    }
    friend Point inter(const Halfplane& s, const Halfplane& t)
        {
        long double alpha = cross((t.p - s.p), t.pq) / cross(s.
            pq, t.pq);
        return s.p + (s.pq * alpha);
    }
};
```

```
vector<Point> hp_intersect(vector<Halfplane>& H) {
    Point box[4] = {
            Point(inf, inf),
            Point(-inf, inf),
            Point(-inf, -inf),
            Point(inf, -inf)
    };
    for(int i = 0; i<4; i++) {
        Halfplane aux(box[i], box[(i+1) % 4]);
        H.push_back(aux);
    }
    sort(H.begin(), H.end());
    deque<Halfplane> dq;
    int len = 0;
    for(int i = 0; i < (int)(H.size()); i++) {
        while (len > 1 && H[i].out(inter(dq[len-1], dq[len-2]))
             ) {
            dq.pop_back();
            --len;
        }
        while (len > 1 && H[i].out(inter(dq[0], dq[1]))) {
            dq.pop_front();
            --len;
        }
        if (len > 0 && fabsl(cross(H[i].pq, dq[len-1].pq)) <
             eps) {
            if (dot(H[i].pq, dq[len-1].pq) < 0.0)
                return vector<Point>();

            if (H[i].out(dq[len-1].p)) {
                dq.pop_back();
                --len;
            }
            else continue;
        }

        dq.push_back(H[i]);
        ++len;
    }
    while (len > 2 && dq[0].out(inter(dq[len-1], dq[len-2]))) {
        dq.pop_back();
        --len;
    }
    while (len > 2 && dq[len-1].out(inter(dq[0], dq[1]))) {
        dq.pop_front();
        --len;
    }
    if (len < 3) return vector<Point>();
    vector<Point> ret(len);
    for(int i = 0; i+1 < len; i++) {
        ret[i] = inter(dq[i], dq[i+1]);
    }
    ret.back() = inter(dq[len-1], dq[0]);
    return ret;
}
```

Diam.h

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

"Point.h" d41d8c, 12 lines

```
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;
}
```

PInsideH.h

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

"Point.h", "sideOf.h", "OnSegment.h" d41d8c, 14 lines

```
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;
}
```

LineHullIntersection.h

Description: 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: $\bullet$ $(-1,-1)$ if no collision, $\bullet$ $(i,-1)$ if touching the corner i, $\bullet$ (i,i) if along side $(i,i+1)$, $\bullet$ (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. `extrVertex` returns the point of a hull with the max projection onto a line.
Time: $\mathcal{O}(\log n)$

"Point.h" d41d8c, 39 lines

```
#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;
}
```

7.4 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.
Time: $\mathcal{O}(n \log n)$

"Point.h" d41d8c, 17 lines

```
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;
}
```

Strings (8)

KMP.h

Description: pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.
Time: $\mathcal{O}(n)$

d41d8c, 16 lines

```
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;
}
```

Zfunc.h

Description: z[x] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)
Time: $\mathcal{O}(n)$

d41d8c, 12 lines

```
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;
}
```

Manacher.h

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

d41d8c, 13 lines

```
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;
}
```

MinRotation.h

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

d41d8c, 8 lines

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

SuffixArray.h

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

d41d8c, 208 lines

```
struct sparse {
    vector<vector<int>> st;

    sparse() { }

    sparse(const vector<int> &a) {
        int n = Size(a);
        int k = 0;
        while (1 << k < n)
            k++;
        st.resize(k + 1, vector<int>(n));
        copy(all(a), st[0].begin());
        for (int i = 1; i <= k; i++) {
            for (int j = 0; j + (1 << i) <= n; j++)
                st[i][j] = min(st[i - 1][j], st[i - 1][j + (1
                    << (i - 1))]);
        }
    }

    int getMin(int l, int r) {
        int k = 31 - __builtin_clz(r - l);
        return min(st[k][l], st[k][r - (1 << k)]);
    }
};

struct SuffixArray {
    int n;
    vector<int> sa, lcp, pos;
    sparse st;
    vector<int> s;

    // O(Size(s) + max(s) - min(s))
    SuffixArray(vector<int> &s): n(Size(s)) {
        int mn = *min_element(all(s));
        for (int &i : s)
            i -= mn - 1;
        s.reserve(Size(s) + 1);
        s.push_back(0);
        sa = build(s, *max_element(all(s)) + 1);

        int n = Size(s);
        pos.resize(n);
        for (int i = 0; i < n; i++)
            pos[sa[i]] = i;
        lcp.resize(n);
        int k = 0;
        for (int i = 0; i < n - 1; i++) {
            int j = sa[pos[i] - 1];
            while (s[i + k] == s[j + k])
                k++;
            lcp[pos[i]] = k;
            k = max(0, k - 1);
        }
        st = sparse(lcp);
        this->s = s;
    }

    vector<int> phase2(const vector<int> &s, const vector<int>
        &pref, const vector<char> &types, const vector<int> &
        lms) {
        int n = Size(s);
        vector<int> cnt = pref;
        vector<int> res(n, -1);
        for (int i : lms) {
            int a = s[i];
            res[--cnt[a + 1]] = i;
        }
        copy(all(pref), cnt.begin());
        for (int p : res) {
            if (p <= 0 || types[p - 1] != 'L')
                continue;
            int a = s[p - 1];
            res[cnt[a]++] = p - 1;
        }
        copy(all(pref), cnt.begin());
        for (int i = n - 1; i >= 0; i--) {
            int p = res[i];
            if (p <= 0 || types[p - 1] != 'S')
                continue;
            int a = s[p - 1];
            res[--cnt[a + 1]] = p - 1;
        }
        return res;
    }

    inline bool is_lms(const vector<char> &types, int i) {
        return types[i - 1] == 'L' && types[i] == 'S';
    }

    // compare two lms substring
    inline bool not_equal(const vector<int> &s, const vector<
        char> &types, int i, int j) {
        assert(is_lms(types, i) && is_lms(types, j));
        bool is_lms1 = false, is_lms2 = false;
        while (true) {
            if (s[i] != s[j] || types[i] != types[j])
                return true;
            if (is_lms1 && is_lms2)
                break;
            i++;
            j++;
            is_lms1 = is_lms(types, i);
            is_lms2 = is_lms(types, j);
        }
        return false;
    }

    // m = max(s) + 1, s.back() == 0
    vector<int> build(vector<int> &s, int m) {
        int n = Size(s);
        assert(!s.empty());
        assert(s.back() == 0);
        assert(Size(s) == 1 || *min_element(s.begin(), s.end()
            - 1) > 0);
        assert(*max_element(all(s)) == m - 1);
        if (Size(s) == 1)
            return {0};

        vector<char> types(n);
        types[n - 1] = 'S';
        vector<int> lms;
        lms.reserve(n);
        for (int i = n - 2; i >= 0; i--) {
            if (s[i] < s[i + 1])
                types[i] = 'S';
            else if (s[i] > s[i + 1])
                types[i] = 'L';
            else
                types[i] = types[i + 1];
            if (types[i] == 'L' && types[i + 1] == 'S')
                lms.push_back(i + 1);
        }
        vector<int> pref(m + 1);
        for (int i : s)
            pref[i + 1]++;
        for (int i = 0; i < m; i++)
            pref[i + 1] += pref[i];
        auto res = phase2(s, pref, types, lms);

        int lms_cnt = 1, color = 0;
        int last = n - 1;
        vector<int> new_sym(n, -1);
        new_sym[n - 1] = 0;
        for (int i = 1; i < n; i++) {
            int p = res[i];
            if (p <= 0 || !is_lms(types, p))
                continue;
            lms[lms_cnt++] = p;
            color += not_equal(s, types, last, p);
            new_sym[p] = color;
            last = p;
        }
        vector<int> new_string;
        vector<int> pos_new_string(n);
        new_string.reserve(Size(lms) + 1);
        for (int i = 0; i < n; i++) {
```

```
            int c = new_sym[i];
            if (c != -1) {
                pos_new_string[Size(new_string)] = i;
                new_string.push_back(c);
            }
        }
        if (color != Size(lms)) {
            auto sa_new = build(new_string, color + 1);
            for (int i = 1; i < Size(sa_new); i++)
                lms[i] = pos_new_string[sa_new[i]];
        }
        return phase2(s, pref, types, lms);
    }

    int get_lcp(int i, int j) {
        if (i == j)
            return n - i;
        i = pos[i];
        j = pos[j];
        if (i > j)
            swap(i, j);
        return st.getMin(i + 1, j + 1);
    }

    bool compare(int i, int j) {  // s[i..] < s[j..]
        if (i == j)
            return false;
        int k = get_lcp(i, j);
        return s[i + k] < s[j + k];
    }
};

//Another impl
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++);
  }
};
```

Hashing.h

Description: creates hashes
Time: $\mathcal{O}(N)$

d41d8c, 158 lines

```
namespace kirkon {

  const int HASH_MOD = MOD;
  const int HASH_SIZE = 2;

  uniform_int_distribution<int> BDIST(0.1 * HASH_MOD, 0.9 *
       HASH_MOD);

  struct custom_hash {
    array<int, HASH_SIZE> vals{};

    custom_hash() {
      vals.fill(0);
    }

    custom_hash(const array<int, HASH_SIZE> &other) {
      vals = other;
    }

    custom_hash(array<int, HASH_SIZE> &&other) {
      vals = std::move(other);
    }

    custom_hash &operator=(const array<int, HASH_SIZE> &other)
        {
      vals = other;
      return *this;
    }

    custom_hash &operator=(array<int, HASH_SIZE> &&other) {
      vals = std::move(other);
      return *this;
    }

    int &operator[](int x) {
      return vals[x];
    }

    bool operator==(const custom_hash &other) const {
      return vals == other.vals;
    }
    bool operator!=(const custom_hash &other) const {
      return vals != other.vals;
    }
    bool operator<(const custom_hash &other) const {
      return vals < other.vals;
    }
    bool operator>(const custom_hash &other) const {
      return vals > other.vals;
    }
    bool operator<=(const custom_hash &other) const {
      return vals <= other.vals;
    }
    bool operator>=(const custom_hash &other) const {
      return vals >= other.vals;
    }
  };

  template<class T>
  custom_hash make_hash(T c) {
    auto res = custom_hash{};
    res.vals.fill(c);
    return res;
  }

  custom_hash base{};
  vector<custom_hash> pows{};

  custom_hash operator+(custom_hash l, custom_hash r) {
    for (int i = 0; i < HASH_SIZE; ++i)
      if ((l[i] += r[i]) >= HASH_MOD)
        l[i] -= HASH_MOD;
    return l;
  }
  custom_hash operator-(custom_hash l, custom_hash r) {
    for (int i = 0; i < HASH_SIZE; ++i)
      if ((l[i] -= r[i]) < 0)
        l[i] += HASH_MOD;
    return l;
  }
  custom_hash operator*(custom_hash l, custom_hash r) {
    for (int i = 0; i < HASH_SIZE; ++i)
      l[i] = (ll) l[i] * r[i] % HASH_MOD;
    return l;
  }

  void init() {

    static bool used = false;
    if (used) {
      return;
    }

    for (auto &u: base.vals) {
      u = BDIST(rng);
    }
    pows.emplace_back(make_hash(1));

    used = true;
  }

  struct HashRange {
    str S;
    vector<custom_hash> cum{};

    HashRange() {
      init();
      cum.emplace_back();
    }

    void add(char c) {
      S += c;
      cum.pb(base * cum.back() + make_hash(c));
    }
    void add(str s) { each(c, s) add(c); }
    void extend(int len) {
      while (sz(pows) <= len) {
        pows.pb(base * pows.back());
      }
    }
    custom_hash hash(int l, int r) {
      int len = r + 1 - l;
      extend(len);
      return cum[r + 1] - pows[len] * cum[l];
    }
  };

}// namespace kirkon

struct custom_int_hash {
  static uint64_t splitmix64(uint64_t x) {
    // http://xorshift.di.unimi.it/splitmix64.c
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }

  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM = chrono::steady_clock::
        now().time_since_epoch().count();
    return splitmix64(x + FIXED_RANDOM);
  }
};

custom_int_hash int_hash{};
```

```
namespace std {
  template<>
  struct hash<custom_hash> {
    inline size_t operator()(const custom_hash& x) const {
      size_t result = 0;
      for (auto u : x.vals) {
        result ^= int_hash(u);
      }
      return custom_int_hash::splitmix64(result);
    }
  };
}
```

AhoCorasick.h

Description: Aho-Corasick automaton, 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.
Time: construction takes $\mathcal{O}(26N)$, where N = sum of length of patterns. find(x) is $\mathcal{O}(N)$, where N = length of x. findAll is $\mathcal{O}(NM)$. d41d8c, 66 lines

```
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;
  }
};
```

Various (9)

9.1 Intervals

IntervalContainer.h

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).
Time: $\mathcal{O}(\log N)$ d41d8c, 23 lines

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

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

IntervalCover.h

Description: Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add `|| R.empty()`. Returns empty set on failure (or if G is empty).
Time: $\mathcal{O}(N \log N)$ d41d8c, 19 lines

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

ConstantIntervals.h

Description: Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval.
Usage: `constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});`
Time: $\mathcal{O}\left(k \log \frac{n}{k}\right)$ d41d8c, 19 lines

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

9.2 Misc. algorithms

TernarySearch.h

Description: Find the smallest i in $[a, b]$ that maximizes $f(i)$, assuming that $f(a) < \ldots < f(i) \geq \cdots \geq f(b)$. To reverse which of the sides allows non-strict inequalities, change the $<$ marked with (A) to $<=$, and reverse the loop at (B). To minimize f, change it to $>$, also at (B).
Usage: `int ind = ternSearch(0,n-1,[&](int i){return a[i];});`
Time: $\mathcal{O}(\log(b-a))$ d41d8c, 11 lines

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

LIS.h

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

d41d8c, 17 lines

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

FastKnapsack.h

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

d41d8c, 16 lines

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

9.3 Dynamic programming

KnuthDP.h

Description: When doing DP on intervals: $a[i][j] = \min_{i<k<j}(a[i][k] + a[k][j]) + f(i,j)$, 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 $f(b,c) \le f(a,d)$ and $f(a,c) + f(b,d) \le f(a,d) + f(b,c)$ for all $a \le b \le c \le d$. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.
Time: $\mathcal{O}(N^2)$

DCDP.h

Description: Given $a[i] = \min_{lo(i)\le k<hi(i)}(f(i,k))$ where the (minimal) optimal k increases with i, computes $a[i]$ for $i = L..R-1$.
Time: $\mathcal{O}((N + (hi - lo))\log N)$

d41d8c, 18 lines

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

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

SOSDP.h

Description: SOS DP
Time: $\mathcal{O}\left(N * 2^N\right)$

d41d8c, 6 lines

```
//memory optimized, super easy to code.
for(int i = 0; i<(1<<N); ++i)  F[i] = A[i];
for(int i = 0;i < N; ++i)
  for(int mask = 0; mask < (1<<N); ++mask) {
    if(mask & (1<<i))   F[mask] += F[mask^(1<<i)];
  }
```

Knapsack.h

Description: Knapsack fast.
Time: $\mathcal{O}\left(n^2 Clogn/64\right)$ and $\mathcal{O}(nC/64)$

d41d8c, 48 lines

```
#pragma push_macro("__SIZEOF_LONG__")
#pragma push_macro("__cplusplus")
#define __SIZEOF_LONG__ __SIZEOF_LONG_LONG__
#define unsigned unsigned long
#define __cplusplus 201102L

#define __builtin_popcountl __builtin_popcountll
#define __builtin_ctzl __builtin_ctzll

#pragma pop_macro("__cplusplus")
#pragma pop_macro("__SIZEOF_LONG__")
#undef unsigned
#undef __builtin_popcountl
#undef __builtin_ctzl

const int C = 1e6 + 3;

vector<int> ans;
int M;
bitset<C> dp1, dp2;

bool divide(const vector<int> &a, int l, int r, int S) {
    if (r - l == 1) {
        if (a[l] == S) {
            ans.push_back(l);
        } else if (S != 0) {
            return false;
        }
        return true;
    }
    int m = (l + r) >> 1;
    dp1 = 0;
    dp1[0] = true;
    for (int i = l; i < m; i++)
        dp1 |= dp1 << a[i];
    dp2 = 0;
    dp2[S] = true;
    for (int i = r - 1; i >= m; i--)
        dp2 |= dp2 >> a[i];
    for (int x = 0; x <= (r - l) * M; x++) {
        if (dp1[x] && dp2[x]) {
            assert(divide(a, l, m, x));
            assert(divide(a, m, r, S - x));
            return true;
        }
    }
    return false;
}
```

Techniques (A)

techniques.txt 159 lines

```
Recursion
Divide and conquer
  Finding interesting points in N log N
Algorithm analysis
  Master theorem
  Amortized time complexity
Greedy algorithm
  Scheduling
  Max contiguous subvector sum
  Invariants
  Huffman encoding
Graph theory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algorithm
  MST: Prim's algorithm
  Bellman-Ford
  Konig's theorem and vertex cover
  Min-cost max flow
  Lovasz toggle
  Matrix tree theorem
  Maximal matching, general graphs
  Hopcroft-Karp
  Hall's marriage theorem
  Graphical sequences
  Floyd-Warshall
  Euler cycles
  Flow networks
  * Augmenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cut vertices, cut-edges and biconnected components
  Edge coloring
  * Trees
  Vertex coloring
  * Bipartite graphs (=> trees)
  * 3^n (special case of set cover)
  Diameter and centroid
  K'th shortest path
  Shortest cycle
Dynamic programming
  Knapsack
  Coin change
  Longest common subsequence
  Longest increasing subsequence
  Number of paths in a dag
  Shortest path in a dag
  Dynprog over intervals
  Dynprog over subsets
  Dynprog over probabilities
  Dynprog over trees
  3^n set cover
  Divide and conquer
  Knuth optimization
  Convex hull optimizations
  RMQ (sparse table a.k.a 2^k-jumps)
  Bitonic cycle
  Log partitioning (loop over most restricted)
Combinatorics
  Computation of binomial coefficients
  Pigeon-hole principle
  Inclusion/exclusion
  Catalan number
  Pick's theorem
Number theory
  Integer parts
  Divisibility
  Euclidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's little theorem
  Euler's theorem
  Phi function
  Frobenius number
  Quadratic reciprocity
  Pollard-Rho
  Miller-Rabin
  Hensel lifting
  Vieta root jumping
Game theory
  Combinatorial games
  Game trees
  Mini-max
  Nim
  Games on graphs
  Games on graphs with loops
  Grundy numbers
  Bipartite games without repetition
  General games without repetition
  Alpha-beta pruning
Probability theory
Optimization
  Binary search
  Ternary search
  Unimodality and convex functions
  Binary search on derivative
Numerical methods
  Numeric integration
  Newton's method
  Root-finding with binary/ternary search
  Golden section search
Matrices
  Gaussian elimination
  Exponentiation by squaring
Sorting
  Radix sort
Geometry
  Coordinates and vectors
  * Cross product
  * Scalar product
  Convex hull
  Polygon cut
  Closest pair
  Coordinate-compression
  Quadtrees
  KD-trees
  All segment-segment intersection
Sweeping
  Discretization (convert to events and sweep)
  Angle sweeping
  Line sweeping
  Discrete second derivatives
Strings
  Longest common substring
  Palindrome subsequences
  Knuth-Morris-Pratt
  Tries
  Rolling polynomial hashes
  Suffix array
  Suffix tree
  Aho-Corasick
  Manacher's algorithm
  Letter position lists
Combinatorial search
  Meet in the middle
  Brute-force with pruning
  Best-first (A*)
  Bidirectional search
  Iterative deepening DFS / A*
Data structures
  LCA (2^k-jumps in trees in general)
  Pull/push-technique on trees
  Heavy-light decomposition
  Centroid decomposition
  Lazy propagation
  Self-balancing trees
  Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
  Monotone queues / monotone stacks / sliding queues
  Sliding queue using 2 stacks
  Persistent segment tree
```