ITMO

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

doomed

The 2025 ICPC World Finals

September 4, 2025

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 $x = -b/2a$.

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

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: $\dfrac{a + b}{a - b} = \dfrac{\tan \dfrac{\alpha + \beta}{2}}{\tan \dfrac{\alpha - \beta}{2}}$

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 $A = \sqrt{(p - a)(p - b)(p - c)(p - d)}$.

1.4.3 Spherical coordinates

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

1.5 Derivatives/Integrals

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

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)

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

bc9cbb, 10 lines

```cpp
#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>;
/*
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).

2009a2, 18 lines

```cpp
#include <bits/extc++.h>
struct custom_int_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15; x = (x ^ (x >> 30)) * 0
        xbf58476d1ce4e5b9; x = (x ^ (x >> 27)) * 0
        x94d049bb133111eb;
    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);
  }
};

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 /* custom_int_hash */> h
    ({},{},{},{},{1<<16});
```

PersistentST.cpp

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

75e5b6, 30 lines

```cpp
struct SegtreeSum {
    int l, r, sum = 0;
    SegtreeSum* left = 0, *right = 0;
    SegtreeSum(int l , int r) : l(l), r(r) {
        int m = (l + r) / 2;
        if (r - l > 1) {
            left = new SegtreeSum(l, m);
            right = new SegtreeSum(m, r);
        }
    }
    void copyLeft() { if (left) left = new SegtreeSum(*left); }
    void copyRight() { if (right) right = new SegtreeSum(*right
        ); }
    void add(int idx, int val) {
        sum += val;
        int m = (l + r) / 2;
        if (r - l > 1) {
            if (idx < m) {
                copyLeft();
                left->add(idx, val);
            } else {
                copyRight();
                right->add(idx, val);
            }
        }
    }
};
// init:
SegtreeSum* init_version = new SegtreeSum(0, n);
SegtreeSum* version_with_update = new SegtreeSum(*init_version)
    ;
version_with_update->add(4, 7);
```

LazyST.h

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

703d3a, 50 lines

```cpp
class segtree {
  public:
  struct node {
    // don't forget to set default value (used for leaves) not
        necessarily neutral element!
    void apply(int l, int r, ... v) { }
  };
  node unite(const node &a, const node &b) const { node res; /*
      res = combine(a, b) */ return res; }
  inline void push(int x, int l, int r) { int y = (l + r) >> 1;
      int z = x + ((y - l + 1) << 1); }
  inline void pull(int x, int z) { tree[x] = unite(tree[x + 1],
      tree[z]); }
  int n; vector<node> tree;

  template <typename M> void build(int x, int l, int r, const
      vector<M> &v) {
    if (l == r) { tree[x].apply(l, r, v[l]); return; }
    int y = (l + r) >> 1; int z = x + ((y - l + 1) << 1);
    build(x + 1, l, y, v); build(z, y + 1, r, v); pull(x, z);
  }
  node get(int x, int l, int r, int ll, int rr) {
    if (ll <= l && r <= rr) { return tree[x]; }
    int y = (l + r) >> 1; int z = x + ((y - l + 1) << 1);
    push(x, l, r);
    node res{};
    if (rr <= y) res = get(x + 1, l, y, ll, rr);
    else if (ll > y) res = get(z, y + 1, r, ll, rr);
    else res = unite(get(x + 1, l, y, ll, rr), get(z, y + 1, r,
        ll, rr));
    pull(x, z);
    return res;
  }
  template <typename... M>
  void modify(int x, int l, int r, int ll, int rr, const M&...
      v) {
    if (ll <= l && r <= rr) { tree[x].apply(l, r, v...); return
        ; }
    int y = (l + r) >> 1; int z = x + ((y - l + 1) << 1);
    push(x, l, r);
    if (ll <= y) modify(x + 1, l, y, ll, rr, v...);
    if (rr > y) modify(z, y + 1, r, ll, rr, v...);
    pull(x, z);
  }

  template <typename M> segtree(const vector<M> &v) {
    n = v.size(); assert(n > 0);
    tree.resize(2 * n - 1); build(0, 0, n - 1, v);
  }
  node get(int ll, int rr) {
    assert(0 <= ll && ll <= rr && rr <= n - 1);
    return get(0, 0, n - 1, ll, rr);
  }
  template <typename... M> void modify(int ll, int rr, const M
      &... v) {
    assert(0 <= ll && ll <= rr && rr <= n - 1);
    modify(0, 0, n - 1, ll, rr, v...);
  }
};
```

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

b6e63c, 21 lines

```cpp
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;
  }
};
```

LineContainer.h

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

95af51, 29 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) { 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: linear CHT and persistent linear CHT for monotonic slopes
Time: $\mathcal{O}(\log N)$

"../../stress-tests/utilities/template.h" e258b1, 91 lines

```
constexpr int64_t CHT_LINE_INF = std::numeric_limits<int64_t>::
    max();

struct line {
  int a{0};
  int64_t b{CHT_LINE_INF};
  line() = default;
  line(int a, int64_t b) noexcept : a(a), b(b) {}
  friend __int128 cross(const line &left, const line &right) {
    return (__int128)left.a * right.b - (__int128)left.b *
        right.a;
  }
  friend line operator-(const line &left, const line &right) {
    return {left.a - right.a, left.b - right.b};
  }
  [[nodiscard]] int64_t evaluate(int x) const { return (int64_t
      )a * x + b; }
};

// query for min, inserting for increasing/decreasing slope
template <bool increasing_slope> struct incremental_CHT {
  vector<line> lines;
  void insert(const line &new_line) {
    while (lines.size() >= 2) {
      const line &line1 = lines[lines.size() - 2], line2 =
          lines.back(), line3 = new_line;
      if ((cross(line3 - line2, line2 - line1) < 0) ^
          increasing_slope) { // > 0 for max
        lines.pop_back();
      } else break;
    }

    if (!lines.empty()) {
      const line &line1 = lines.back(), line2 = new_line;
      if (line1.a == line2.a) {
        if (line1.b > line2.b) lines.pop_back(); // < for max
        else return;
      }
    }
    lines.push_back(new_line);
  }

  // have to perform queries with decreasing x
  int64_t query(int x) {
    assert(!lines.empty());
    while (lines.size() >= 2) {
      const line &line1 = lines[lines.size() - 2], line2 =
          lines.back();
      if (line1.evaluate(x) < line2.evaluate(x)) lines.pop_back
          (); // > for max
      else break;
    }
    return lines.back().evaluate(x);
  }
};

template <bool increasing_slope> struct
    persistent_incremental_CHT {
  vector<line> lines;
  vector<int> parents;
  int max_query_alive_vertex{};
  persistent_incremental_CHT() = default;

  int insert(const line &new_line) {
    parents.push_back(-1); lines.push_back(new_line);
        max_query_alive_vertex = lines.size() - 1;
    if (lines.size() == 1) return 0;
    int k = lines.size() - 1, j = lines.size() - 2, i = parents
        [j];
    parents[k] = j;
    while (i != -1) {
      const line &line1 = lines[i], line2 = lines[j], line3 =
          new_line;
      if ((cross(line3 - line2, line2 - line1) > 0) ^ !
          increasing_slope) { // < 0 for max
        parents[k] = i; j = i; i = parents[i];
      } else break;
    }

    while (j != -1) {
      const line &line1 = lines[j], line2 = new_line;
      if (line1.a == line2.a) {
        if (line1.b > line2.b) parents[k] = j = parents[j]; //
            < for max
        else return k;
      } else break;
    }
    return k;
  }

  int get(int root, int x) {
    if (root == -1 || parents[root] == -1) return root;
    const line &line1 = lines[parents[root]], line2 = lines[
        root];
    if (line1.evaluate(x) < line2.evaluate(x)) return parents[
        root] = get(parents[root], x);
    else return root;
  }

  // have to perform queries with increasing x if slope
      increases
  int64_t query(int root, int x) {
    assert(!lines.empty());
    int64_t answer = lines[root].evaluate(x);
    return min(answer, lines[get(root, x)].evaluate(x));
  }
};
```

RMQ.h

Description: Sparse tables.
Time: $\mathcal{O}(1)$ for query, $\mathcal{O}(n \log n)$ for build

b8d6b0, 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]);
  }
};
```

2.0.1 Segment Tree Beats

Ji Driver Segment Tree Beats. $min =, max =, sum?, min?max?$ – store max and second max

$min =, + =, gcd.$

store differences on segment, $a[i] - a[j]$, but only BST of differences,

$mod =, set, sum?$

break condiiton max ¿ mod tag condition max == min.

$sqrt =, + =, sum?max?min?$

store sum, max, min, break condition just standard segtree $qr <= l||r <= qr$ tag condition $max - min <= 1$

$div =, + =, sum?max?min?$

store sum, max, min, break condition just standard segtree $qr <= l||r <= qr$ tag condition $max - min <= 1$

$\& =, | =, max?$

$C = 2^k$ is important the upper bound for numbers. store $max, pushand, pushor, and_o n_s eg, or_o n_s eg$ break condition: standard segtree $qr <= l||r <= ql$ tag condition for and = $ql <= l\&\&r <= qr\&\&((and_o n_s eg[v]^o r_o n_s eg[v])\& x) == 0$ tag condition for or = $ql <= l\&\&r <= qr\&\&((and_o n_s eg[v]^o r_o n_s eg[v])\&y) == 0$

Numerical (3)

3.1 Polynomials and recurrences

Polynomial.h

Description: Polynomial operations 7e61ac, 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: $\mathcal{O}(n^2)$

518501, 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: $\mathcal{O}(N^2)$ and $\mathcal{O}(nlog^2(k))$

863929, 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 Optimization

Integrate.h

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

369aa2, 7 lines

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

Simplex.h

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

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

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

afb5a2, 68 lines

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

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

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

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

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

```
    rep(j,0,n+2) if (j != s) D[r][j] *= inv;
    rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

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

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

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 $N = 2^{22}$) 781fb8, 35 lines

```
using C = complex<double>;
using vd = vector<double>;
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 $N \log_2 N \cdot \mathrm{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \mathrm{mod})$.
Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT) a0b3a7, 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 $g = \mathrm{root}^{(mod-1)/N}$. 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 $[0, \mathrm{mod})$.
Time: $\mathcal{O}(N \log N)$
"../number-theory/ModPow.h" d3c419, 99 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};
}

using Mat = vector<vl>;

//a is NxN, b is MxM
Mat conv2d_ntt(const Mat& A, const Mat& B) {
  int ha = (int)A.size(); int wa = ha ? (int)A[0].size() : 0;
  int hb = (int)B.size(); int wb = hb ? (int)B[0].size() : 0;
  if (!ha || !wa || !hb || !wb) return {};
  int H  = ha + hb - 1;
  int wr = wa + wb - 1;

  // Flatten with stride = wr so column indices never alias
      between rows.
  vl A1(ha * wr, 0), B1(hb * wr, 0);
  rep(i, 0, ha) rep(j, 0, wa) {
    ll x = A[i][j] % mod;
    A1[i * wr + j] = x + (x < 0) * mod;
  }

  rep(i, 0, hb) rep(j, 0, wb) {
    ll x = B[i][j] % mod;
    B1[i * wr + j] = x + (x < 0) * mod;
  }

  vl C1 = conv(A1, B1);
  Mat R(H, vl(wr, 0));
  rep(i, 0, H) rep(j, 0, wr) R[i][j] = C1[i * wr + j] % mod;
  return R;
}

Mat conv2d_brute(Mat a, Mat b) {
  int n = a.size(), m = b.size();
  Mat ans(n + m - 1, vl(n + m - 1, 0));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int r = 0; r < m; r++) {
        for (int c = 0; c < m; c++) {
          ans[i + r][j + c] += 1LL * a[i][j] * b[r][c] % mod;
          ans[i + r][j + c] %= mod;
        }
      }
    }
  }
  return ans;
}
```

```
Mat rot180(const Mat& K) {
    int h = (int)K.size(), w = h ? (int)K[0].size() : 0;
    Mat R(h, vl(w, 0));
    rep(i, 0, h) rep(j, 0, w)
      R[h - 1 - i][w - 1 - j] = (K[i][j] % MOD + MOD) % MOD;
    return R;
}

Mat correlate2d_ntt(const Mat& A, const Mat& P) {
    return conv2d_ntt(A, rot180(P));
}

// Crop the "valid" region (top-left placements of pattern in
     grid)
Mat crop_valid(const Mat& Cfull, int ha, int wa, int hb, int wb
     ) {
    int H = ha - hb + 1, W = wa - wb + 1;
    int offi = hb - 1, offj = wb - 1;
    Mat R(H, vl(W, 0));
    rep(i, 0, H) rep(j, 0, W)
      R[i][j] = Cfull[i + offi][j + offj];
    return R;
}
```

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

503b23, 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" b81014, 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;
  }
};
```

ModOps.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

8bb07c, 67 lines

```
using ull = unsigned long long;
using ll = long long;

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;
}

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);
}

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;
}

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

fbb85f, 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" 939d71, 10 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.

b2e2fb, 34 lines

```
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 $a \pmod b$.

f391ae, 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 $0 \le x < \mathrm{lcm}(m,n)$. Assumes $mn < 2^{62}$.

Time: $\log(n)$

"euclid.h" 766e15, 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 $\Rightarrow a^{p-1} \equiv 1 \pmod p$ $\forall a$.

ddb727, 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: $\mathcal{O}(\log N)$

d01a3e, 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: $\mathcal{O}(\log(N))$

989e7d, 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: $\mathcal{O}(n)$
732749, 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 $\binom{k_1+\cdots+k_n}{k_1,k_2,\ldots,k_n} = \frac{(\sum k_i)!}{k_1!k_2!...k_n!}$.
50ddd2, 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. $\pi(j) \ge j$, 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

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.

"../../stress-tests/utilities/template.h" 5556bd, 67 lines

```
template<class F>
struct dinic {
  static constexpr db eps = 1e-6;
  struct Edge {
    int to; F flow, cap;
    Edge() = default;
    Edge(int to_, F flow_, F cap_) : to(to_), flow(flow_), cap(
         cap_) {}
  };

  int n; vvi gr; vector<Edge> edges;
  vector<F> dist; vector<int> first;
  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); }

  F res(int id) { return edges[id].cap - edges[id].flow; }
  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 = numeric_limits<F>::max()
      ) {
    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 max_flow(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(int s, int t) {
    max_flow(s, t);
    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: $\mathcal{O}(FLOW + m + n)$

74b971, 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: $\mathcal{O}(flow * mlogn + mn)$

3640cb, 47 lines

```
template <class F, class C = F> struct MCMF {
  struct Edge { int to; F flow, cap; C cost; };
  int n; vector<C> pot, dist;
  vector<int> previous_edge; vector<Edge> edges; vector<vector<
       int>> gr;

  MCMF(int n_) : n(n_) {
    pot.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();
    dist.assign(n, inf);

    using T = pair<C, int>; priority_queue<T, vector<T>,
        greater<T>> bfs;
    bfs.push({dist[s] = 0, s});
    while (!bfs.empty()) {
      auto [cur_dist, v] = bfs.top(); bfs.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 + pot[v] - pot[E.to]))
          previous_edge[E.to] = e, bfs.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]; if (E.cap) ckmin(pot[E.to], pot
          [edges[e ^ 1].to] + E.cost);  // Bellman–Ford
    }
    F totalFlow = 0; C totalCost = 0;
    while (path(s, t)) {
      for (int i = 0; i < n; ++i) pot[i] += (dist[i] ==
          numeric_limits<C>::max() ? 0 : 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 += (pot[t] - pot[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};
  }
};
```

MCMFPushRelabel.h

Description: Min-cost max-flow. Supports lower bounds and negative costs (and even cycles!)
Time: $\mathcal{O}\left(V^3 \log(VC)\right)$

"../../stress-tests/utilities/template.h" 2de8ec, 275 lines

```
template <class F> struct HLPP {
  struct Edge {
    int to, inv;
    F rem, cap;
  };
  vector<vector<Edge>> G;
  vector<F> excess;
  vector<int> hei, arc, prv, nxt, act, bot;
  queue<int> Q;
  int n, high, cut, work;
  HLPP(int k) : G(k) {}
  int addEdge(int u, int v, F cap, F rcap = 0) {
    assert(u != v);
    G[u].push_back({v, sz(G[v]), cap, cap});
    G[v].push_back({u, sz(G[u]) - 1, rcap, rcap});
    return sz(G[u]) - 1;
  }
  void raise(int v, int h) {
    prv[nxt[prv[v]] = nxt[v]] = prv[v];
    hei[v] = h;
    if (excess[v] > 0) {
      bot[v] = act[h];
      act[h] = v;
      high = max(high, h);
    }
    if (h < n)
      cut = max(cut, h + 1);
    nxt[v] = nxt[prv[v] = h += n];
    prv[nxt[nxt[h] = v]] = v;
  }
  void global(int s, int t) {
    hei.assign(n, n * 2);
    act.assign(n * 2, -1);
    iota(all(prv), 0);
    iota(all(nxt), 0);
    hei[t] = high = cut = work = 0;
    hei[s] = n;
    for (int x : {t, s})
      for (Q.push(x); !Q.empty(); Q.pop()) {
        int v = Q.front();
        for (auto &e : G[v])
          if (hei[e.to] == n * 2 && G[e.to][e.inv].rem)
            Q.push(e.to), raise(e.to, hei[v] + 1);
      }
  }
  void push(int v, Edge &e, bool z) {
    auto f = min(excess[v], e.rem);
    if (f > 0) {
      if (z && !excess[e.to]) {
        bot[e.to] = act[hei[e.to]];
        act[hei[e.to]] = e.to;
      }
      e.rem -= f;
      G[e.to][e.inv].rem += f;
      excess[v] -= f;
      excess[e.to] += f;
    }
  }
  void discharge(int v) {
    int h = n * 2, k = hei[v];
    for (int j = 0; j < sz(G[v]); j++) {
      auto &e = G[v][arc[v]];
      if (e.rem) {
        if (k == hei[e.to] + 1) {
          push(v, e, 1);
          if (excess[v] <= 0)
            return;
        } else
          h = min(h, hei[e.to] + 1);
      }
      if (++arc[v] >= sz(G[v]))
        arc[v] = 0;
    }
    if (k < n && nxt[k + n] == prv[k + n]) {
      for (int j = k; j < cut; j++)
        while (nxt[j + n] < n)
          raise(nxt[j + n], n);
      cut = k;
    } else
      raise(v, h), work++;
  }
  // Compute maximum flow from src to dst
  F flow(int src, int dst) {
    excess.assign(n = sz(G), 0);
    arc.assign(n, 0);
    prv.assign(n * 3, 0);
    nxt.assign(n * 3, 0);
    bot.assign(n, 0);
    for (auto &e : G[src])
      excess[src] = e.rem, push(src, e, 0);
    global(src, dst);
    for (; high; high--)
      while (act[high] != -1) {
        int v = act[high];
        act[high] = bot[v];
        if (v != src && hei[v] == high) {
          discharge(v);
          if (work > 4 * n)
            global(src, dst);
        }
      }
    global(src, dst);
    return excess[dst];
  }
  // Get flow through e-th edge of vertex v
  F getFlow(int v, int e) { return G[v][e].cap - G[v][e].rem; }
  // Get if v belongs to cut component with src
  bool cutSide(int v) { return hei[v] >= n; }
};

template <class T> struct Circulation {
  const T INF = numeric_limits<T>::max() / 2;
  T lowerBoundSum = 0;
  HLPP<T> mf;

  // Initialize for n vertices
  Circulation(int k) : mf(k + 2) {}
  void addEdge(int s, int e, T l, T r) {
    mf.addEdge(s + 2, e + 2, r - l);
    if (l > 0) {
      mf.addEdge(0, e + 2, l);
      mf.addEdge(s + 2, 1, l);
      lowerBoundSum += l;
    } else {
      mf.addEdge(0, s + 2, -l);
      mf.addEdge(e + 2, 1, -l);
      lowerBoundSum += -l;
    }
  }
  bool solve(int s, int e) {
    // mf.addEdge(e+2, s+2, INF); // to reduce as maxflow with
        lower bounds, in circulation problem skip this line
    return lowerBoundSum == mf.flow(0, 1);
    // to get maximum LR flow, run maxflow from s+2 to e+2
        again
  }
};

template <class T> struct MinCostCirculation {
  const int SCALE = 3; // scale by 1/(1 << SCALE)
  const T INF = numeric_limits<T>::max() / 2;
  struct EdgeStack {
    int s, e;
    T l, r, cost;
  };
  struct Edge {
    int pos, rev;
    T rem, cap, cost;
  };
  int n;
  vector<EdgeStack> estk;
  Circulation<T> circ;
  vector<vector<Edge>> gph;
  vector<T> p;
```

```
MinCostCirculation(int k) : circ(k), gph(k), p(k) { n = k; }
void addEdge(int s, int e, T l, T r, T cost){
  estk.push_back({s, e, l, r, cost});
}
pair<bool, T> solve(){
  for(auto &i : estk){
    if(i.s != i.e) circ.addEdge(i.s, i.e, i.l, i.r);
  }
  if(!circ.solve(-1, -1)){
    return make_pair(false, T(0));
  }
  vector<int> ptr(n);
  T eps = 0;
  for(auto &i : estk){
    T curFlow;
    if(i.s != i.e) curFlow = i.r - circ.mf.G[i.s + 2][ptr[i.s
        ]].rem;
    else curFlow = i.r;
    int srev = sz(gph[i.e]);
    int erev = sz(gph[i.s]);
    if(i.s == i.e) srev++;
    gph[i.s].push_back({i.e, srev, i.r - curFlow, i.r, i.cost
        * (n + 1)});
    gph[i.e].push_back({i.s, erev, -i.l + curFlow, -i.l, -i.
        cost * (n + 1)});
    eps = max(eps, abs(i.cost) * (n + 1));
    if(i.s != i.e){
      ptr[i.s] += 2;
      ptr[i.e] += 2;
    }
  }
  while(true){
    auto cost = [&](Edge &e, int s, int t){
      return e.cost + p[s] - p[t];
    };
    eps = 0;
    for(int i = 0; i < n; i++){
      for(auto &e : gph[i]){
        if(e.rem > 0) eps = max(eps, -cost(e, i, e.pos));
      }
    }
    if(eps <= T(1)) break;
    eps = max(T(1), eps >> SCALE);
    bool upd = 1;
    for(int it = 0; it < 5 && upd; it++){
      upd = false;
      for(int i = 0; i < n; i++){
        for(auto &e : gph[i]){
          if(e.rem > 0 && p[e.pos] > p[i] + e.cost + eps){
            p[e.pos] = p[i] + e.cost + eps;
            upd = true;
          }
        }
      }
      if(!upd) break;
    }
    if(!upd) continue;
    vector<T> excess(n);
    queue<int> que;
    auto push = [&](Edge &e, int src, T flow){
      e.rem -= flow;
      gph[e.pos][e.rev].rem += flow;
      excess[src] -= flow;
      excess[e.pos] += flow;
      if(excess[e.pos] <= flow && excess[e.pos] > 0){
        que.push(e.pos);
      }
    };
    vector<int> ptr(n);
      auto relabel = [&](int v){
        ptr[v] = 0;
        p[v] = -INF;
        for(auto &e : gph[v]){
          if(e.rem > 0){
            p[v] = max(p[v], p[e.pos] - e.cost - eps);
          }
        }
      };
      for(int i = 0; i < n; i++){
        for(auto &j : gph[i]){
          if(j.rem > 0 && cost(j, i, j.pos) < 0){
            push(j, i, j.rem);
          }
        }
      }
      while(sz(que)){
        int x = que.front();
        que.pop();
        while(excess[x] > 0){
          for(; ptr[x] < sz(gph[x]); ptr[x]++){
            Edge &e = gph[x][ptr[x]];
            if(e.rem > 0 && cost(e, x, e.pos) < 0){
              push(e, x, min(e.rem, excess[x]));
              if(excess[x] == 0) break;
            }
          }
          if(excess[x] == 0) break;
          relabel(x);
        }
      }
    }
    T ans = 0;
    for(int i = 0; i < n; i++){
      for(auto &j : gph[i]){
        j.cost /= (n + 1);
        ans += j.cost * (j.cap - j.rem);
      }
    }
    return make_pair(true, ans / 2);
  }
  void bellmanFord(){
    fill(all(p), T(0));
    bool upd = 1;
    while(upd){
      upd = 0;
      for(int i = 0; i < n; i++){
        for(auto &j : gph[i]){
          if(j.rem > 0 && p[j.pos] > p[i] + j.cost){
            p[j.pos] = p[i] + j.cost;
            upd = 1;
          }
        }
      }
    }
  }
};
```

Hungarian.h

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

b2e770, 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: $\mathcal{O}(V^3)$

319bef, 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

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

dd7412, 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: $\mathcal{O}(N^3)$

"../../stress-tests/utilities/template.h" 966510, 129 lines

```
struct blossom {
    int n, m;
    vi mate;   vvi b;
    vi p, d, bl; vvi 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, vi(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;
    }

    vi trace(int x) {
        vi 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, vi &vx, vi &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];
                }
            }
        }
    }

    vi lift(vi &vx) {
        vi 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) {
                            vi vx = trace(x);
                            vi 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);
                                vi A = lift(vx);
                                vi 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: $\mathcal{O}(E+V)$

"../../stress-tests/utilities/template.h", "graphs_structures.h" 719d65, 43 lines

```
template <typename T> vi find_scc(const digraph<T> &g, int &cnt
    ) {
  digraph<T> g_rev = g.reverse();
  vi order;
  vector<bool> was(g.n, false);
  function<void(int)> dfs1 = [&](int v) {
    was[v] = true;
    for (int id : g.g[v]) {
      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]) {
      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: $\mathcal{O}(E+V)$

afe742, 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.

"../../stress-tests/utilities/template.h", "graphs_structures.h", "SCC.h" 2ea981, 23 lines

```
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));
        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); } }
  vi solve() { int cnt; vi c = find_scc(g, cnt); vi res(n);
    for (int i = 0; i < n; i++) {
      if (c[i << 1] == c[i << 1 ^ 1]) return vi();
      res[i] = (c[i << 1] < c[i << 1 ^ 1]);
    }
    return res; }
};
```

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: $\mathcal{O}(V+E)$

d9d811, 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()};
}
```

Johnson.h

Description: calculates all-pairs shortest paths in a graph that might have negative edge weights.
Time: $\mathcal{O}\left(N^3\right)$

"../../stress-tests/utilities/template.h", "../../content/graph/BellmanFord.h", "../../content/graph/dijkstra.h" d752ab, 18 lines

```
vector<vector<ll>> johnson(graph<ll> &g) {
  int n = g.n;

  ++g.n; g.g.resize(g.n);
  for (int i = 0; i < n; ++i) g.add(n, i, 0);
  vector<ll> d = fordbellman(g, n);
  g.edges.erase(g.edges.end() - n, g.edges.end());
  for (auto& adj : g.g) while (!adj.empty() && adj.back() >= g.
      edges.size()) { adj.pop_back(); }
  --g.n; g.g.resize(g.n);

  for(auto& e : g.edges) e.cost += d[e.from] - d[e.to];
  vector<vector<ll>> ans(n, vector<ll>(n, numeric_limits<ll>::
      max()));
  for (int v = 0; v < n; ++v){
    ans[v] = dijkstra(g, v);
    for (int u = 0; u < n; ++u) if (ans[v][u] != numeric_limits
        <ll>::max()) ans[v][u] -= d[v] - d[u];
  }
  return ans;
}
```

6.4 Coloring

DCPOffline.h

Description: DCP offline algorithm. Actually could be generalized to any offline queries.
Time: $\mathcal{O}\left(V\log^2(V)\right)$

"../../stress-tests/utilities/template.h" 4ed61a, 87 lines

```
struct dsu_save {
    int v, rnkv, u, rnku;
    dsu_save() = default;
    dsu_save(int _v, int _rnkv, int _u, int _rnku) : v(_v),
         rnkv(_rnkv), u(_u), rnku(_rnku) {}
};

struct dsu_with_rollbacks {
    vector<int> p, rnk; int comps;
    stack<dsu_save> op;
    dsu_with_rollbacks() = default;
    dsu_with_rollbacks(int n) {
        p.resize(n); rnk.resize(n);
        iota(p.begin(), p.end(), 0); rnk.assign(n, 0); comps =
             n;
    }

    int find_set(int v) {
        return (v == p[v]) ? v : find_set(p[v]);
    }

    bool unite(int v, int u) {
        v = find_set(v); u = find_set(u);
        if (v == u) return false;
        comps--;
        if (rnk[v] > rnk[u]) swap(v, u);
        op.push(dsu_save(v, rnk[v], u, rnk[u])); p[v] = u;
        if (rnk[u] == rnk[v]) rnk[u]++;
        return true;
    }

    void rollback() {
        if (op.empty()) return;
        dsu_save x = op.top(); op.pop(); comps++;
        p[x.v] = x.v; rnk[x.v] = x.rnkv;
        p[x.u] = x.u; rnk[x.u] = x.rnku;
    }
};

struct query {
    int v, u;
    bool united = true;
    query(int _v, int _u) : v(_v), u(_u) {}
};

struct QueryTree {
    vector<vector<query>> t;
    dsu_with_rollbacks dsu;
    int T;

    QueryTree() = default;

    QueryTree(int _T, int n) : T(_T) {
        dsu = dsu_with_rollbacks(n);
        t.resize(4 * T + 4);
    }

    void add_to_tree(int v, int l, int r, int ul, int ur, query
         & q) {
        if (ul > ur) return;
        if (l == ul && r == ur) {
            t[v].push_back(q);
            return;
        }
        int mid = (l + r) / 2;
        add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
        add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur
             , q);
    }

    void add_query(query q, int l, int r) { // edge (q.u, q.v)
         lives on segment [l, r]
        add_to_tree(1, 0, T - 1, l, r, q);
    }

    void dfs(int v, int l, int r, vector<int>& ans) {
        for (query& q : t[v]) {
            q.united = dsu.unite(q.v, q.u);
        }
        if (l == r) ans[l] = dsu.comps; // here you can
             customize answers on queries
        else { int mid = (l + r) / 2; dfs(2 * v, l, mid, ans);
             dfs(2 * v + 1, mid + 1, r, ans);}
        for (auto q : t[v]) {
            if (q.united) dsu.rollback();
        }
    }

    vector<int> solve() {
        vector<int> ans(T);
        dfs(1, 0, T - 1, ans);
        return ans;
    }
};
```

6.5 Trees

Centroids.h

Description: Finds centroids decomposition
Time: $\mathcal{O}(n\log)$

980ca7, 89 lines

```
template <typename T>
vector<int> centroid_decomposition(const forest<T>& g) {
  int n = g.n;
  vector<bool> alive(n, true);
  vector<int> res; res.reserve(n);

  vector<int> sz(n);
  function<void(int, int)> Dfs = [&](int v, int pr) {
    sz[v] = 1;
    for (int eid : g.g[v]) {
      auto& e = g.edges[eid];
      int u = e.from ^ e.to ^ v;
      if (u != pr && alive[u]) {
        Dfs(u, v);
        sz[v] += sz[u];
      }
    }
  };
  function<void(int)> Build = [&](int v) -> void {
    Dfs(v, -1);
    int c = v;
    int pr = -1;
    while (true) {
      int nxt = -1;
      for (int eid : g.g[c]) {
        auto& e = g.edges[eid];
        int u = e.from ^ e.to ^ c;
        if (u != pr && alive[u] && 2 * sz[u] > sz[v]) {
          nxt = u;
          break;
        }
      }
      if (nxt == -1) {
        break;
      }
      pr = c;
      c = nxt;
    }
    res.pb(c);
    alive[c] = false;
    for (int eid : g.g[c]) {
      auto& e = g.edges[eid];
      int u = e.from ^ e.to ^ c;
      if (alive[u]) {
        Build(u);
      }
    }
  };
  for (int i = 0; i < n; i++) {
    if (alive[i]) {
      Build(i);
    }
  }
  return res;
}

/*
  auto centers = centroid_decomposition(g);
  constexpr int LOGN = 17;
  vector<vector<array<int, 2>>> parents(LOGN, vector<array<int,
        2>>(n));
  vector<int> alive(n, 1);
  vector<int> pointers(n, 0);
  {
```

```
    vector<array<int, 2>> bfs(n);
    int head = 0, tail = 0;
    for (auto c : centers) {
      alive[c] = false;
      head = tail = 0;
      bfs[tail++] = {c, -1};
      while(head < tail) {
        auto [v, par] = bfs[head++];
        if (par != -1) {
          parents[pointers[v]][v] = parents[pointers[v]][par];
              parents[pointers[v]][v][1]++;  ++pointers[v];
        } else {
          parents[pointers[v]++][v] = {v, 0};
        }

        for (auto eid : g.g[v]) {
          auto& e = g.edges[eid];
          auto to = e.from ^ e.to ^ v;
          if (to == par || !alive[to]) {
            continue;
          }
          bfs[tail++] = {to, v};
        }
      }
    }
  }
*/
```

TouristHLD.h

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

f7c2c5, 403 lines

```
template <typename T>
  class digraph : public graph<T> {
public:
    using graph<T>::edges;
    using graph<T>::g;
    using graph<T>::n;

    digraph(int _n) : graph<T>(_n) {}

    int add(int from, int to, T cost = 1) override {
      assert(0 <= from && from < n && 0 <= to && to < n);
      int id = (int)edges.size();
      g[from].push_back(id);
      edges.push_back({from, to, cost});
      return id;
    }

    digraph<T> reverse() const {
      digraph<T> rev(n);
      for (auto &e : edges) {
        rev.add(e.to, e.from, e.cost);
      }
      return rev;
    }
  };

  template <typename T>
  class dfs_digraph : public digraph<T> {
public:
    using digraph<T>::edges;
    using digraph<T>::g;
    using digraph<T>::n;

    vector<int> pv;
    vector<int> pe;
    vector<int> order;
    vector<int> pos;
    vector<int> end;
    vector<int> sz;
    vector<int> root;
    vector<int> depth;
    vector<T> dist;

    dfs_digraph(int _n) : digraph<T>(_n) {}

    void clear() {
      pv.clear();
      pe.clear();
      order.clear();
      pos.clear();
      end.clear();
      sz.clear();
      root.clear();
      depth.clear();
      dist.clear();
    }

    void init() {
      pv = vector<int>(n, -1);
      pe = vector<int>(n, -1);
      order.clear();
      pos = vector<int>(n, -1);
      end = vector<int>(n, -1);
      sz = vector<int>(n, 0);
      root = vector<int>(n, -1);
      depth = vector<int>(n, -1);
      dist = vector<T>(n);
    }

private:
    void do_dfs(int v) {
      pos[v] = (int)order.size();
      order.push_back(v);
      sz[v] = 1;
      for (int id : g[v]) {
        if (id == pe[v]) {
          continue;
        }
        auto &e = edges[id];
        int to = e.from ^ e.to ^ v;
        // well, this is controversial...
        if (depth[to] != -1) {
          continue;
        }
        depth[to] = depth[v] + 1;
        dist[to] = dist[v] + e.cost;
        pv[to] = v;
        pe[to] = id;
        root[to] = (root[v] != -1 ? root[v] : to);
        do_dfs(to);
        sz[v] += sz[to];
      }
      end[v] = (int)order.size() - 1;
    }

    void do_dfs_from(int v) {
      depth[v] = 0;
      dist[v] = T{};
      root[v] = v;
      pv[v] = pe[v] = -1;
      do_dfs(v);
    }

public:
    int dfs_one_unsafe(int v) {
      // run init() before this
      // then run this with the required v's
      do_dfs_from(v);
      return v;
    }

    int dfs(int v) {
      init();
      do_dfs_from(v);
      //    assert((int) order.size() == n);
      return v;
    }

    void dfs_many(const vector<int> &roots) {
      init();
      for (int v : roots) {
        if (depth[v] == -1) {
          do_dfs_from(v);
        }
      }
      //    assert((int) order.size() == n);
    }

    vector<int> dfs_all() {
      init();
      vector<int> roots;
      for (int v = 0; v < n; v++) {
        if (depth[v] == -1) {
          roots.push_back(v);
          do_dfs_from(v);
        }
      }
      assert((int)order.size() == n);
      return roots;
    }
  };

  template <typename T>
  class forest : public graph<T> {
public:
    using graph<T>::edges;
    using graph<T>::g;
    using graph<T>::n;

    forest(int _n) : graph<T>(_n) {
    }

    int add(int from, int to, T cost = 1) {
      assert(0 <= from && from < n && 0 <= to && to < n);
      int id = (int) edges.size();
      assert(id < n - 1);
      g[from].push_back(id);
      g[to].push_back(id);
      edges.push_back({from, to, cost});
      return id;
    }
  };

  template <typename T>
  class dfs_forest : public forest<T> {
public:
    using forest<T>::edges;
    using forest<T>::g;
    using forest<T>::n;

    vector<int> pv;
    vector<int> pe;
    vector<int> order;
    vector<int> pos;
    vector<int> end;
```

```
  vector<int> sz;
  vector<int> root;
  vector<int> depth;
  vector<T> dist;

  dfs_forest(int _n) : forest<T>(_n) {
  }

  void init() {
    pv = vector<int>(n, -1);
    pe = vector<int>(n, -1);
    order.clear();
    pos = vector<int>(n, -1);
    end = vector<int>(n, -1);
    sz = vector<int>(n, 0);
    root = vector<int>(n, -1);
    depth = vector<int>(n, -1);
    dist = vector<T>(n);
  }

  void clear() {
    pv.clear();
    pe.clear();
    order.clear();
    pos.clear();
    end.clear();
    sz.clear();
    root.clear();
    depth.clear();
    dist.clear();
  }

private:
  void do_dfs(int v) {
    pos[v] = (int) order.size();
    order.push_back(v);
    sz[v] = 1;
    for (int id : g[v]) {
      if (id == pe[v]) {
        continue;
      }
      auto [e_from, e_to, cost] = edges[id];
      int to = e_from ^ e_to ^ v;
      depth[to] = depth[v] + 1;
      dist[to] = dist[v] + cost;
      pv[to] = v;
      pe[to] = id;
      root[to] = (root[v] != -1 ? root[v] : to);
      do_dfs(to);
      sz[v] += sz[to];
    }
    end[v] = (int) order.size() - 1;
  }

  void do_dfs_from(int v) {
    depth[v] = 0;
    dist[v] = T{};
    root[v] = v;
    pv[v] = pe[v] = -1;
    do_dfs(v);
  }

public:
  void dfs(int v, bool clear_order = true) {
    if (pv.empty()) {
      init();
    } else {
      if (clear_order) {
        order.clear();
      }
    }
    do_dfs_from(v);
  }

  void dfs_all() {
    init();
    for (int v = 0; v < n; v++) {
      if (depth[v] == -1) {
        do_dfs_from(v);
      }
    }
    assert((int) order.size() == n);
  }
};

template <typename T>
class hld_forest : public dfs_forest<T> {
public:
  using dfs_forest<T>::edges;
  using dfs_forest<T>::g;
  using dfs_forest<T>::n;
  using dfs_forest<T>::pv;
  using dfs_forest<T>::sz;
  using dfs_forest<T>::root;
  using dfs_forest<T>::pos;
  using dfs_forest<T>::end;
  using dfs_forest<T>::order;
  using dfs_forest<T>::depth;
  using dfs_forest<T>::dfs;
  using dfs_forest<T>::dfs_all;

  vector<int> head;
  vector<int> visited;

  hld_forest(int _n) : dfs_forest<T>(_n) {
    visited.resize(n);
  }

  void build_hld(const vector<int> &roots) {
    for (int tries = 0; tries < 2; tries++) {
      if (roots.empty()) {
        dfs_all();
      } else {
        order.clear();
        for (int root : roots) {
          dfs(root, false);
        }
        assert((int) order.size() == n);
      }
      if (tries == 1) {
        break;
      }
      for (int i = 0; i < n; ++i) {
        if (g[i].empty()) {
          continue;
        }
        int best = -1, bid = 0;
        for (int j = 0; j < (int) g[i].size(); ++j) {
          int id = g[i][j];
          auto [from, to, cost] = edges[id];
          int v = from ^ to ^ i;
          if (pv[v] != i) {
            continue;
          }
          if (sz[v] > best) {
            best = sz[v];
            bid = j;
          }
        }
        swap(g[i][0], g[i][bid]);
      }
    }
    head.resize(n);
    iota(head.begin(), head.end(), 0);
    for (int i = 0; i + 1 < n; ++i) {
      int x = order[i];
      int y = order[i + 1];
      if (pv[y] == x) {
        head[y] = head[x];
      }
    }
  }

  void build_hld(int v) {
    build_hld(vector<int>{v});
  }

  void build_hld_all() {
    build_hld(vector<int>());
  }

  bool apply_on_path(int x, int y, bool with_lca, function<
      void(int,int,bool)> f) {
    // f(x, y, up): up -- whether this part of the path goes
        up
    assert(!head.empty());
    int z = lca(x, y);
    if (z == -1) {
      return false;
    }
    {
      int v = x;
      while (v != z) {
        if (depth[head[v]] <= depth[z]) {
          f(pos[z] + 1, pos[v], true);
          break;
        }
        f(pos[head[v]], pos[v], true);
        v = pv[head[v]];
      }
    }
    if (with_lca) {
      f(pos[z], pos[z], false);
    }
    {
      int v = y;
      int cnt_visited = 0;
      while (v != z) {
        if (depth[head[v]] <= depth[z]) {
          f(pos[z] + 1, pos[v], false);
          break;
        }
        visited[cnt_visited++] = v;
        v = pv[head[v]];
      }
      for (int at = cnt_visited - 1; at >= 0; at--) {
        v = visited[at];
        f(pos[head[v]], pos[v], false);
      }
    }
    return true;
  }

  bool anc(int x, int y) {
    return (pos[x] <= pos[y] && end[y] <= end[x]);
  }
```

```
  int go_up(int x, int up) {
    int target = depth[x] - up;
    if (target < 0) {
      return -1;
    }
    while (depth[head[x]] > target) {
      x = pv[head[x]];
    }
    return order[pos[x] - depth[x] + target];
  }

  int lca(int x, int y) {
    if (root[x] != root[y]) {
      return -1;
    }
    for (; head[x] != head[y]; y = pv[head[y]]) {
      if (depth[head[x]] > depth[head[y]]) {
        swap(x, y);
      }
    }
    return depth[x] < depth[y] ? x : y;
  }
};
```

6.5.1 Tree hashes

$h(v) = \sum_{sorted\_by\_hash(ch)} h(ch)^2 + h(ch)p^i + 239$

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.)

67be69, 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 << ")"; }
};
```

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" 400eab, 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" 7b7463, 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;
}
```

MinEnclosCirc.h

Description: Computes the minimum circle that encloses a set of points.
Time: expected $\mathcal{O}(n)$

"circumcircle.h" e48a3d, 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

extremeVertex.cpp

Description: Given convex polygon p ordered ccw and point z, finds vertex of polygon w, such that dot(w, z) is maximum. top - upper right vertex. Needs any adequate implementation of PT structure

8c5324, 24 lines

```
inline int dot(PT a, PT b) { return a.x * b.x + a.y * b.y; }

int extreme_vertex(vector<PT> &p, const PT &z, const int top) {
    int n = p.size();
    if (n == 1) return 0;
    int ans = dot(p[0], z); int id = 0;
    if (dot(p[top], z) > ans) ans = dot(p[top], z), id = top;
    int l = 1, r = top - 1;
    while (l < r) {
        int mid = l + r >> 1;
        if (dot(p[mid + 1], z) >= dot(p[mid], z)) l = mid + 1;
        else r = mid;
    }
    if (dot(p[l], z) > ans) ans = dot(p[l], z), id = l;
    l = top + 1, r = n - 1;
    while (l < r) {
        int mid = l + r >> 1;
        if (dot(p[(mid + 1) % n], z) >= dot(p[mid], z)) l = mid
             + 1;
        else r = mid;
    }
    l %= n;
    if (dot(p[l], z) > ans) ans = dot(p[l], z), id = l;
    return id;
}
```

pointPolyDist.cpp

Description: Given convex polygon p ordered ccw and point z, finds distance from z to p. Assumes that p strictly outside. Requires some trivial geometry functions

a9d38e, 58 lines

```
inline int orientation(PT a, PT b, PT c) { return sign(cross(b
     - a, c - a)); }

pair<PT, int> point_poly_tangent(vector<PT> &p, PT Q, int dir,
     int l, int r) {
    while (r - l > 1) {
        int mid = (l + r) >> 1;
        bool pvs = orientation(Q, p[mid], p[mid - 1]) != -dir;
        bool nxt = orientation(Q, p[mid], p[mid + 1]) != -dir;
        if (pvs && nxt) return {p[mid], mid};
        if (!(pvs || nxt)) {
            auto p1 = point_poly_tangent(p, Q, dir, mid + 1, r)
                 ;
            auto p2 = point_poly_tangent(p, Q, dir, l, mid - 1)
                 ;
            return orientation(Q, p1.first, p2.first) == dir ?
                 p1 : p2;
        }
```

```
        if (!pvs) {
            if (orientation(Q, p[mid], p[l]) == dir)  r = mid -
                 1;
            else if (orientation(Q, p[l], p[r]) == dir) r = mid
                 - 1;
            else l = mid + 1;
        }
        if (!nxt) {
            if (orientation(Q, p[mid], p[l]) == dir)  l = mid +
                 1;
            else if (orientation(Q, p[l], p[r]) == dir) r = mid
                 - 1;
            else l = mid + 1;
        }
    }
    pair<PT, int> ret = {p[l], l};
    for (int i = l + 1; i <= r; i++) ret = orientation(Q, ret.
        first, p[i]) != dir ? make_pair(p[i], i) : ret;
    return ret;
}

pair<int, int> tangents_from_point_to_polygon(vector<PT> &p, PT
     Q){
    int ccw = point_poly_tangent(p, Q, 1, 0, (int)p.size() - 1)
        .second;
    int cw = point_poly_tangent(p, Q, -1, 0, (int)p.size() - 1)
        .second;
    return make_pair(ccw, cw);
}

// minimum distance from a point to a convex polygon
// it assumes point lie strictly outside the polygon
double dist_from_point_to_polygon(vector<PT> &p, PT z) {
    double ans = inf;
    int n = p.size();
    if (n <= 3) {
        for(int i = 0; i < n; i++) ans = min(ans,
            dist_from_point_to_seg(p[i], p[(i + 1) % n], z));
        return ans;
    }
    auto [r, l] = tangents_from_point_to_polygon(p, z);
    if(l > r) r += n;
    while (l < r) {
        int mid = (l + r) >> 1;
        double left = dist2(p[mid % n], z), right= dist2(p[(mid
             + 1) % n], z);
        ans = min({ans, left, right});
        if(left < right) r = mid;
        else l = mid + 1;
    }
    ans = sqrt(ans);
    ans = min(ans, dist_from_point_to_seg(p[l % n], p[(l + 1) %
        n], z));
    ans = min(ans, dist_from_point_to_seg(p[l % n], p[(l - 1 +
        n) % n], z));
    return ans;
}
```

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" 78720e, 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;
}
```

HalfplaneInt.h

Description: find halfplane intersection
Time: $\mathcal{O}(NlogN)$

18c3ef, 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" 7e8d01, 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" 57d900, 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: • $(-1,-1)$ if no collision, • $(i,-1)$ if touching the corner i, • (i,i) if along side $(i,i+1)$, • (i,j) if crossing sides $(i,i+1)$ and $(j,j+1)$. In the last case, if a corner i is crossed, this is treated as happening on side $(i,i+1)$. The points are returned in the same order as the line hits the polygon. `extrVertex` returns the point of a hull with the max projection onto a line.
Time: $\mathcal{O}(\log n)$

"Point.h" d3ea5a, 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" 335c6e, 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: $\mathcal{O}(n)$

d25715, 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: $\mathcal{O}(n)$

27498b, 9 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: $\mathcal{O}(N)$

ea8b7a, 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: $\mathcal{O}(N)$

e7ec17, 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: $\mathcal{O}(n \log n)$

"../../stress-tests/utilities/template.h" 6d073c, 211 lines

```
#define Size(x) (int)(x).size()

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];
```

```cpp
        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};
```

```cpp
    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];
}
```

```cpp
};

//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: $\mathcal{O}(N)$

"../../stress-tests/utilities/template.h" 841711, 76 lines

```cpp
constexpr int HASH_MOD = MOD; constexpr 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]; }

  // if C++ 20 is availiable use auto operator<=> and bool
       operator== instead
  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 (exchange(used, true)) { return;
       }
  for (auto &u: base.vals) { u = BDIST(rng); }
  pows.emplace_back(make_hash(1));
}

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]; }
};

struct custom_int_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15; x = (x ^ (x >> 30)) * 0
         xbf58476d1ce4e5b9; x = (x ^ (x >> 27)) * 0
         x94d049bb133111eb;
    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 $\mathcal{O}(NM)$. f95ee7, 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 Misc. algorithms

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: $\mathcal{O}(N \max(w_i))$ 1ad58f, 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;
}
```

9.2 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: $\mathcal{O}((N + (hi - lo)) \log N)$ 96ef35, 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: $\mathcal{O}\left(N * 2^N\right)$ b2d048, 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: $\mathcal{O}\left(n^2 C logn/64\right)$ and $\mathcal{O}\left(nC/64\right)$ 2ffd3d, 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
```