École Normale Supérieure, Paris

ENS Ulm 1

Justin Cahuzac, Sven Meyer, Yann Viegas

2024-10-17

Contest (1)

template.cpp — 38 lines

```
#include <bits/stdc++.h>
#define int long long
#define ll long long
#define sz(x) x.size()
#define all(x) x.begin(),x.end()
#define rep(i,a,b) for(int i=a;i<b;i++)
using namespace std;

string to_string(string s) { return s; }
template <typename T> string to_string(T v) {
  bool first = true;
  string res = "[";
  for (const auto &x : v) {
    if (!first) res += ", ";
    first = false;
    res += to_string(x);
  }
  res += "]";
  return res;
}

void dbg_out() { cout << endl; }
template <typename Head, typename... Tail> void dbg_out(Head H,
      Tail... T) {
  cout << ' ' << to_string(H);
  dbg_out(T...);
}

#ifdef DEBUG
#define dbg(...) cout << "(" << #__VA_ARGS__ << "):", dbg_out(
    __VA_ARGS__)
#else
#define dbg(...)
#endif

signed main(void) {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
}
```

comp.sh — 7 lines

```
#!/bin/bash
if g++ -DDEBUG -std=gnu++20 -Wall -Wextra -Wshadow -O0 -g -
     fsanitize=address,undefined $1; then
  for i in *.in; do
    echo $i
    ./a.out < $i
  done
fi
```

stress.sh — 7 lines

```
for ((i = 1; ; ++i)); do  # if they are same then will loop
     forever
    echo $i
    ./gen $i > int
    ./a.out < int > out1 || break
    ./brute < int > out2 || break
    diff -w out1 out2 || break
done
```

random.cpp
Description: randint(0, n) returns random integer between 0 and n (included)
Time: Constant
4 lines

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
     count());
int randint(int lb, int ub) {
  return uniform_int_distribution<int>(lb, ub)(rng);
}
```

Mathematics (2)

2.0.1 Game Theory

Sprague-Grundy theorem hypothesis: the game is impartial, each player have the same set of moves, a player that can't move loses. To calculate the SG number:

- The SG number of the sum two games with numbers SG_1 and SG_2 is $SG_1 \oplus SG_2$
- The SG number of a game is mex S where S is the set of SG numbers of the games you can move to, and mex $S = \min \mathbb{N} \setminus S$
- The first player has a winning strategy $\Leftrightarrow$ the SG number of the game is $\neq 0$

Classical Games (❶ last one wins (normal); ❷ last one loses (misère))
NIM:
n piles of objs. One can take any number of objs from any pile (i.e. set of possible moves for the i-th pile is $M = [pile_i]$, $[x] := \{1, 2, ..., \lfloor x \rfloor\}$).
$SG = \otimes_{i=1}^{n} pile_i$. Strategy: ❶ make the Nim-Sum 0 by *decreasing* a heap; ❷ the same, except when the normal move would only leave heaps of size 1. In that case, leave an odd number of 1's. The result of ❷ is the same as ❶, opposite if all piles are 1's. Many games are essentially NIM.
NIM (powers)
$M = \{a^m | m \geq 0\}$
If a odd: $SG_n = n\%2$
If a even: $SG_n = 2$, if $n \equiv a\%(a+1)$; $SG_n = n\%(a+1)\%2$, else.
NIM (half)
$M_① = [\frac{pile_i}{2}]$
$M_② = [\lceil \frac{pile_i}{2} \rceil, pile_i]$
①$SG_{2n} = n$, $SG_{2n+1} = SG_n$
②$SG_0 = 0$, $SG_n = [\log_2 n] + 1$
NIM (divisors)
$M_①$ = divisors of $pile_i$
$M_②$ = proper divisors of $pile_i$
①$SG_0 = 0$, $SG_n = SG_{②,n} + 1$
②$SG_1 = 0$, SG_n = number of 0's at the end of n_{binary}
Subtraction Game
$M_① = [k]$ $M_② = S$ (finite) $M_③ = S \cup \{pile_i\}$
$SG_{①,n} = n \mod (k+1)$. ❶lose if $SG = 0$; ❷lose if $SG = 1$.
$SG_{③,n} = SG_{②,n} + 1$
For any finite M, SG of one pile is eventually periodic.
Moore's NIM$_k$
One can take any number of objs from at most k piles.
❶Write $pile_i$ in base $k+1$, sum up each digit without carry. Losing if the result is 0.
❷ If all piles are 1's, losing iff $n \equiv 1\%(k+1)$. Otherwise the result is the same as ❶.
Staircase NIM
n piles in a line. One can take any number of objs from $pile_i$, $i > 0$ to $pile_{i-1}$
Losing if the NIM formed by the odd-indexed piles is losing(i.e. $\otimes_{i=0}^{(n-1)/2} pile_{2i+1} = 0$)
Lasker's NIM
Two possible moves: 1.take any number of objs; 2.split a pile into two (no obj removed)
$SG_n = n$, if $n \equiv 1, 2(\%4)$ $SG_n = n+1$, if $n \equiv 3(\%4)$
$SG_n = n-1$, if $n \equiv 0(\%4)$
Kayles
Two possible moves: 1.take 1 or 2 objs; 2.split a pile into two (after removing objs)
SG_n for small n can be computed recursively. SG_n for $n \in [72, 83]$: 4 1 2 8 1 4 7 2 1 8 2 7 SG_n becomes periodic from the 72-th item with period length 12.
Dawson's Chess
n boxes in a line. One can occupy a box if its neighbours are not occupied.

SG_n for $n \in [1, 18]$: 1 1 2 0 3 1 1 0 3 3 2 2 4 0 5 2 2 3 Period = 34 from the 52-th item.

Wythoff's Game

Two piles of objs. One can take any number of objs from either pile, or take the *same* number from *both* piles.
$n_k = \lfloor k\phi \rfloor = \lfloor m_k\phi \rfloor - m_k$ $m_k = \lfloor k\phi^2 \rfloor = \lceil n_k\phi \rceil = n_k + k$
$\phi := \frac{1+\sqrt{5}}{2}$. (n_k, m_k) is the k-th losing position.
n_k and m_k form a pair of complementary Beatty Sequences (since $\frac{1}{\phi} + \frac{1}{\phi^2} = 1$). Every $x > 0$ appears either in n_k or in m_k.

Mock Turtles

n coins in a line. One can turn over 1, 2 or 3 coins, with the rightmost from head to tail.
$SG_n = 2n$, if ones$(2n)$ odd; $SG_n = 2n + 1$, else. ones(x): the number of 1's in x_{binary}
SG_n for $n \in [0, 10]$ (leftmost position is 0): 1 2 4 7 8 11 13 14 16 19 21

Ruler

n coins in a line. One can turn over any *consecutive* coins, with the rightmost from head to tail.
SG_n = the largest power of 2 dividing n. This is implemented as $n\&-n$(lowbit)
SG_n for $n \in [1, 10]$: 1 2 1 4 1 2 1 8 1 2

Divisors

You have a number n. One can divide it at each turn.
SG_n = number of prime factors of n

2.1 Equations

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.

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

2.3 Geometry

2.3.1 Triangles

Side lengths: a, b, c
Semiperimeter: $p = \frac{a+b+c}{2}$
Area: $A = \sqrt{p(p-a)(p-b)(p-c)}$
Circumradius: $R = \frac{abc}{4A}$
Inradius: $r = \frac{A}{p}$
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: $\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c} = \frac{1}{2R}$
Law of cosines: $a^2 = b^2 + c^2 - 2bc\cos\alpha$
Law of tangents: $\frac{a+b}{a-b} = \frac{\tan\frac{\alpha+\beta}{2}}{\tan\frac{\alpha-\beta}{2}}$

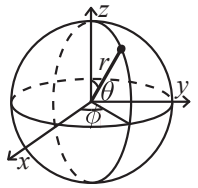
2.3.2 Quadrilaterals

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

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

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

2.3.3 Spherical coordinates



$$x = r\sin\theta\cos\phi \qquad r = \sqrt{x^2 + y^2 + z^2}$$
$$y = r\sin\theta\sin\phi \qquad \theta = \text{acos}(z/\sqrt{x^2 + y^2 + z^2})$$
$$z = r\cos\theta \qquad \phi = \text{atan2}(y, x)$$

2.3.4 Formulas

For a planar graph, $v - e + f = 2$
Pick Theorem : $S = I + B/2 - 1$. Where S is area, B number of points on boundary and I points strictly inside.

2.4 Derivatives/Integrals

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

Integration by parts:

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

2.5 Sums

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

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

2.6 Probability theory

2.6.1 Discrete distributions

2.6.2 Continuous distributions

Uniform distribution

If the probability density function is constant between a and b and 0 elsewhere it is $U(a, b)$, $a < b$.

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2},\ \sigma^2 = \frac{(b-a)^2}{12}$$

Exponential distribution

The time between events in a Poisson process is Exp(λ), $\lambda > 0$.

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \ge 0 \\ 0 & x < 0 \end{cases}$$

$$\mu = \frac{1}{\lambda},\ \sigma^2 = \frac{1}{\lambda^2}$$

Normal distribution

Most real random values with mean μ and variance σ^2 are well described by $\mathcal{N}(\mu, \sigma^2)$, $\sigma > 0$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

2.7 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 (3)

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

16 lines

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

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

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

HashMap.h

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

9 lines

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

using hash_table = __gnu_pbds::gp_hash_table<int, int, chash>;
using hash_set = __gnu_pbds::gp_hash_table<int, __gnu_pbds::
    null_type, chash>;
```

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

30 lines

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

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

ConvexHullTrick.h

Description: Container where you can add lines of the form kx+b, and query maximum values at points x. Can also undo last add. If you don't need it, delete lines with (D) Lines must be added by increasing slope
Time: $\mathcal{O}(\log N)$

51 lines

```
struct cht {
  vector<double> from;
  vector<pair<int, int>> arr;
  vector<pair<int, int>> rem; // (D)
  vector<double> remf; // (D)
  vector<int> cnt; // (D)
  int fst = 0;

  // increasing k
  void add(int k, int b) {
    double x = -1e18;
    cnt.push_back(0); // (D)
    while (arr.size() && (x = (double)(b - arr.back().second) /
                              (arr.back().first - k)) < from.
                                  back()) {
      ++cnt.back(); // (D)
      rem.push_back(arr.back()); // (D)
      remf.push_back(from.back()); // (D)
      arr.pop_back();
      from.pop_back();
    }
    from.push_back(x);
    arr.emplace_back(k, b);
  }
  // maximum
  double get(double x) {
    if (!from.size())
      return 0;
    int ind = upper_bound(from.begin(), from.end(), x) - from.
        begin() - 1;
    return arr[ind].first * x + arr[ind].second;
  }

  // TO CHECK
  double getIncreasing(double x) {
    while (fst + 1 < (int)arr.size() and
           arr[fst].first * x + arr[fst].second <
               arr[fst + 1].first * x + arr[fst + 1].second)
      ++fst;
    return arr[fst].first * x + arr[fst].second;
  }

  // (D)
  void undo() {
    arr.pop_back();
    from.pop_back();
    for (int i = 0; i < cnt.back(); ++i) {
      arr.push_back(rem.back()), rem.pop_back();
      from.push_back(remf.back()), remf.pop_back();
    }
    cnt.pop_back();
  }
};
```

Treap.h

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

55 lines

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

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

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

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

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

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

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

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

FenwickTree.h

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

22 lines

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

FenwickTree2d.h

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

"FenwickTree.h" 26 lines

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

RMQ.h

Description: Range Minimum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time.
Usage: `RMQ rmq(values);`
`rmq.query(inclusive, exclusive);`
Time: $\mathcal{O}(|V| \log |V| + Q)$

14 lines

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

Numerical (4)

4.1 Polynomials and recurrences

PolyRoots.h

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

"Polynomial.h" 23 lines

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

PolyInterpolate.h

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

13 lines

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

BerlekampMassey.h

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

"../number-theory/ModPow.h" 20 lines

```
vector<ll> berlekampMassey(vector<ll> s) {
  int n = sz(s), L = 0, m = 0;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;

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

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

LinearRecurrence.h

Description: Generates a vector P of size n given linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$, such that if $S_0, ..., S_{n-1}$ are fixed, $S_k = \sum P_i S_i$ Faster than matrix multiplication. Can be faster by replacing combine by (a * b) Useful together with Berlekamp–Massey.
Usage: `linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number`
Time: $\mathcal{O}\left(n^2 \log k\right)$

23 lines

```
typedef vector<ll> Poly;
Poly linearRec(Poly tr, ll k) {
  int n = sz(tr);

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

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

  for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
  }
  return {pol.begin() + 1, pol.end()};
}
```

4.2 Optimization

GoldenSectionSearch.h

Description: Finds the argument minimizing the function f in the interval $[a, b]$ assuming f is unimodal on the interval, i.e. has only one local minimum. The maximum error in the result is eps. Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.
Usage: `double func(double x) { return 4+x+.3*x*x; }`
`double xmin = gss(-1000,1000,func);`
Time: $\mathcal{O}\left(\log((b-a)/\epsilon)\right)$

14 lines

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

HillClimbing.h

Description: Poor man's optimization for unimodal functions.

14 lines

```
typedef array<double, 2> P;

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

Integrate.h

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

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;
}
```

IntegrateAdaptive.h

Description: Fast integration using an adaptive Simpson's rule.
Usage: `double sphereVolume = quad(-1, 1, [](double x) {`
`return quad(-1, 1, [&](double y) {`
`return quad(-1, 1, [&](double z) {`
`return x*x + y*y + z*z < 1; });});});`

15 lines

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

template <class F>
d rec(F& f, d a, d b, d eps, d S) {
  d c = (a + b) / 2;
  d S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
  if (abs(T - S) <= 15 * eps || b - a < 1e-10)
    return T + (T - S) / 15;
  return rec(f, a, c, eps / 2, S1) + rec(f, c, b, eps / 2, S2);
}
template<class F>
d quad(d a, d b, F f, d eps = 1e-8) {
  return rec(f, a, b, eps, S(a, b));
}
```

Simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$. Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
Usage: `vvd A = {{1,-1}, {-1,1}, {-1,-2}};`
`vd b = {1,1,-4}, c = {-1,-1}, x;`
`T val = LPSolver(A, b, c).solve(x);`
Time: $\mathcal{O}\left(NM * \#pivots\right)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}\left(2^n\right)$ in the general case.

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;
  }
};
```

4.3 Matrices

Determinant.h

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

15 lines

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

```
    rep(j,i+1,n) {
      double v = a[j][i] / a[i][i];
      if (v != 0) rep(k,i+1,n) a[j][k] -= v * a[i][k];
    }
  }
  return res;
}
```

IntDeterminant.h

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

18 lines

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

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost. Time: $O(n^2m)$ Status: tested on kattis:equationsolver, and bruteforce-tested mod 3 and 5 for n,m <= 3

39 lines

```
typedef vector<double> vd;
const double eps = 1e-12;

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

  rep(i, 0, n) {
    double v, bv = 0;
    rep(r, i, n) rep(c, i, m) if ((v = fabs(A[r][c])) > bv)
     br = r, bc = c, bv = v;
    if (bv <= eps) {
      rep(j, i, n) if (fabs(b[j]) > eps) return -1;
      break;
    }
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    rep(j, 0, n) swap(A[j][i], A[j][bc]);
    bv = 1 / A[i][i];
    rep(j, i + 1, n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      rep(k, i + 1, m) A[j][k] -= fac * A[i][k];
    }
    rank++;
  }
  x.assign(m, 0);
  for (int i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    rep(j, 0, i) b[j] -= A[j][i] * b[i];
  }
  return rank; // (multiple solutions if rank < m)
}
```

SolveLinear2.h

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

"SolveLinear.h" 7 lines

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

SolveLinearBinary.h

Description: Solves $Ax = b$ over $\mathbb{F}_2$. If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys A and b. Time: $O(n^2m)$ Status: bruteforce-tested for n, m <= 4

39 lines

```
typedef bitset<1000> bs;

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

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

MatrixInverse.h

Description: Invert matrix A. Returns rank; result is stored in A unless singular (rank < n). Can easily be extended to prime moduli; for prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step. Time: $O(n^3)$ Status: Slightly tested

36 lines

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

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

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

  rep(i, 0, n) rep(j, 0, n) A[col[i]][col[j]] = tmp[i][j];
  return n;
}
```

MatrixInverse-mod.h

Description: Invert matrix A modulo a prime. Returns rank; result is stored in A unless singular (rank < n). For prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step. Time: $O(n^3)$ Status: Slightly tested

"../number-theory/ModPow.h" 41 lines

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

  rep(i, 0, n) {
    int r = i, c = i;
    rep(j, i, n) rep(k, i, n) if (A[j][k]) {
      r = j;
      c = k;
      goto found;
    }
    return i;
  found:
    A[i].swap(A[r]);
    tmp[i].swap(tmp[r]);
```

```
    rep(j, 0, n) swap(A[j][i], A[j][c]), swap(tmp[j][i], tmp[j
        ][c]);
    swap(col[i], col[c]);
    ll v = modpow(A[i][i], mod - 2);
    rep(j, i + 1, n) {
      ll f = A[j][i] * v % mod;
      A[j][i] = 0;
      rep(k, i + 1, n) A[j][k] = (A[j][k] - f * A[i][k]) % mod;
      rep(k, 0, n) tmp[j][k] = (tmp[j][k] - f * tmp[i][k]) %
          mod;
    }
    rep(j, i + 1, n) A[i][j] = A[i][j] * v % mod;
    rep(j, 0, n) tmp[i][j] = tmp[i][j] * v % mod;
    A[i][i] = 1;
  }

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

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

Tridiagonal.h

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

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

This is useful for solving problems on the type

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

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

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

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

26 lines

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

4.4 Fourier transforms

FastFourierTransform.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 $N = 2^{22}$)

35 lines

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

FastFourierTransformMod.h

Description: Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT)

"FastFourierTransform.h" 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;
}
```

NumberTheoreticTransform.h

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

"../number-theory/ModPow.h" 35 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};
}
```

InverseFPS.h

Description: Inverse of FPS $\pmod{X^n}$
Time: $\mathcal{O}(N \log N)$

17 lines

```
vector<int> inversePowerSeries(vector<int> p, int n) {
  vector<int> inv(1, modpow(p[0], mod - 2));
  for (int curSz = 2; curSz < 2 * n; curSz *= 2) {
    vector<int> cur(2 * curSz);
    for (int i = 0; i < curSz; ++i)
      cur[i] = i < (int)p.size() ? p[i] : 0;
    inv.resize(2 * curSz);
    ntt(inv), ntt(cur);
    int invSz = modpow(2 * curSz, mod - 2);
    for (int i = 0; i < 2 * curSz; ++i)
      inv[i] = inv[i] * (2 - cur[i] * inv[i] % mod + mod) % mod
          * invSz % mod;
    reverse(inv.begin() + 1, inv.begin() + 2 * curSz);
    ntt(inv);
    fill(inv.begin() + curSz, inv.begin() + 2 * curSz, 0);
  }
  return {inv.begin(), inv.begin() + n};
}
```

log-exp.h

Description: Exp/Log of FPS $\pmod{X^n}$
Time: $\mathcal{O}(N \log N)$

31 lines

```
// n should be power of 2
// a is modified
vector<int> logPowerSeries(vector<int> &a, int n) {
  a.resize(n);
  a[0] = 1;
  auto b = inversePowerSeries(a, n);
  for (int i = 0; i < n - 1; ++i)
    a[i] = (i + 1) * a[i + 1] % mod;
  a = conv(a, b);
  for (int i = n - 1; i >= 1; --i)
    a[i] = a[i - 1] * modpow(i, mod - 2) % mod;
  a[0] = 0;
  return a;
}

vector<int> expPowerSeries(vector<int> &a, int n) {
  vector<int> r(1, 1);
  a.resize(n);
  for (int m = 2; m <= n; m *= 2) {
    vector<int> x = r;
    x = logPowerSeries(x, m);
    for (int i = 0; i < m; ++i)
      x[i] = (a[i] - x[i] + mod) % mod;
    x[0] = (x[0] + 1) % mod;
    r.resize(m);
    x.resize(m);
    r = conv(r, x);
    r.resize(m);
  }
  return r;
}
```

EuclidFPS.h

Description: Euclid division of FPS
Time: $\mathcal{O}(N \log N)$

19 lines

```
pair<vector<int>, vector<int>> euclidFPS(vector<int> A, vector<
    int> B) {
  while (!B.empty() and !B.back()) B.pop_back();
  int n = A.size(), m = B.size();
  if (n < m) return pair(vector<int>{}, A);
  reverse(A.begin(), A.end()), reverse(B.begin(), B.end());
  auto invB = inversePowerSeries(B, n - m + 1);
  auto quotient = conv(A, invB);
  quotient.resize(n - m + 1);
  reverse(quotient.begin(), quotient.end());
  while (!quotient.empty() and quotient.back().x == 0) quotient
      .pop_back();
  reverse(A.begin(), A.end()), reverse(B.begin(), B.end());
  auto R = conv(B, quotient);
  for (int i = 0; i < (int)R.size(); ++i) {
    A[i] -= R[i];
    if (A[i] < 0) A[i] += MOD;
  }
  while (!A.empty() and !A.back()) A.pop_back();
  return pair(quotient, A);
}
```

MultipointEval.h

Description: Evaluates pol at pts[0], ..., pts[N-1]
Time: $\mathcal{O}(N \log^2 N)$

42 lines

```
vector<int> mulT(vector<int> &a, vector<int> b) {
  if (b.empty()) return {};
  int n = b.size();
  reverse(b.begin(), b.end());
  auto prd = conv(a, b);
  if ((int)prd.size() < n) return {};
  return {prd.begin() + n - 1, prd.end()};
}

vector<int> multiPoint(vector<int> pol, vector<int> pts) {
  if (pts.empty()) return {};
  int n = max(pol.size(), pts.size());
  vector<vector<Mint>> seg(4 * n);
  vector<int> sol(pts.size()); pts.resize(n);
  auto build = [&](auto rec, int node, int deb, int fin) ->
      void {
    if (deb + 1 == fin) {
      seg[node] = {1, -pts[deb]};
    } else {
      int mid = (deb + fin) / 2;
      rec(rec, 2 * node, deb, mid); rec(rec, 2 * node + 1, mid,
          fin);
      seg[node] = conv(seg[2 * node], seg[2 * node + 1]);
    }
  };
  build(build, 1, 0, n);
  auto trav = [&](auto rec, int node, int deb, int fin,
                  vector<Mint> cur) -> void {
    if (deb + 1 == fin) {
      if (deb < (int)sol.size())
        sol[deb] = cur[0];
      return;
    }
    int mid = (deb + fin) / 2;
    vector<int> lft = mulT(cur, seg[2 * node + 1]);
    lft.resize(min((int)lft.size(), mid - deb));
    rec(rec, 2 * node, deb, mid, move(lft));
    vector<int> rgt = mulT(cur, seg[2 * node]);
    rgt.resize(min((int)rgt.size(), fin - mid));
    rec(rec, 2 * node + 1, mid, fin, move(rgt));
  };
  trav(trav, 1, 0, n, mulT(pol, inversePowerSeries(seg[1], n)))
      ;
  return sol;
}
```

FastSubsetTransform.h

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

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 (5)

5.1 Modular arithmetic

ModInverse.h

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

3 lines

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

ModLog.h

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

11 lines

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

ModSum.h

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

16 lines

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

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

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

ModMulLL.h

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

11 lines

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

ModSqrt.h

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

"ModPow.h" 24 lines

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

Nimber.h

Description: Implements multiplication and division in Nimber field where addition = xor. Can be carried for any unsigned 64 bit integer
Time: decent constant factor

39 lines

```
using Nimber = uint64_t;
bool log_computed = false;
const int small = 1 << 16;
uint16_t pw[small], lg[small];

Nimber mul(Nimber a, Nimber b, int half = 32) {
  if (a <= 1 or b <= 1) return a * b;
  if (log_computed and a < small and b < small) {
    int t = (int)lg[a] + lg[b];
    return pw[t >= small - 1 ? t - (small - 1) : t];
  }
  auto mask = (1ULL << half) - 1;
  auto [a0, a1] = make_pair(a & mask, a >> half);
  auto [b0, b1] = make_pair(b & mask, b >> half);
  auto A = mul(a0, b0, half / 2);
  auto C = mul(a1, b1, half / 2);
  auto B = mul(a0 ^ a1, b0 ^ b1, half / 2) ^ A ^ C;
  B = (B << half);
  C = (C << half) ^ mul(C, 1ULL << (half - 1), half / 2);
  return A ^ B ^ C;
}

Nimber fastpow(Nimber a, int b) {
  Nimber ret = 1;
  for (; b; b /= 2, a = mul(a, a))
    if (b % 2) ret = mul(ret, a);
  return ret;
}

Nimber inv(Nimber x) { return fastpow(x, -2); }

void initNimber() { // CALL THIS
  pw[0] = 1;
  lg[1] = 0;
  uint16_t base = -1;
  for (int i = 1; i < small - 1; ++i)
    pw[i] = mul(pw[i - 1], base), lg[pw[i]] = i;
  log_computed = true;
}
```

5.2 Primality

FastEratosthenes.h

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

24 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" 12 lines

```
bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
  ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
      s = __builtin_ctzll(n-1), d = n >> s;
  for (ull a : A) {   // ^ count trailing zeroes
    ull p = modpow(a%n, d, n), i = s;
    while (p != 1 && p != n - 1 && a % n && i--)
      p = modmul(p, p, n);
    if (p != n-1 && i != s) return 0;
  }
  return 1;
}
```

Factor.h

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

"ModMulLL.h", "MillerRabin.h" 18 lines

```
ull pollard(ull n) {
  auto f = [n](ull x) { return modmul(x, x, n) + 1; };
  ull 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);
}
vector<ull> factor(ull n) {
  if (n == 1) return {};
  if (isPrime(n)) return {n};
  ull x = pollard(n);
  auto l = factor(x), r = factor(n / x);
  l.insert(l.end(), all(r));
  return l;
}
```

5.3 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 $a \pmod b$.

5 lines

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

CRT.h

Description: Chinese Remainder Theorem.
`crt(a, m, b, n)` computes x such that $x \equiv a \pmod m$, $x \equiv b \pmod n$. If $|a| < m$ and $|b| < n$, x will obey $0 \le x < \text{lcm}(m, n)$. Assumes $mn < 2^{62}$.
Time: $\log(n)$

"euclid.h" 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;
}
```

5.3.1 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

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

5.4 Fractions

ContinuedFractions.h

Description: Given N and a real number $x \geq 0$, finds the closest rational approximation p/q with $p, q \leq N$. It will obey $|p/q - x| \leq 1/qN$.
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: $\mathcal{O}(\log N)$

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 $p, q \leq N$. 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: $\mathcal{O}(\log(N))$

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;
}
```

5.5 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.

5.6 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 $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

5.7 Estimates

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

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$. Highly composite numbers:
8, 96, 840, 9240, 98280, 997920, 8648640, 99459360, 994593600, 9777287520, 97772875200, 963761198400, 9958865716800, 97821761637600, 978217616376000, 9651747148243200, 98930408269492800, 994651672331116800

5.8 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)

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

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

5.9 Irreducible polynomials over $\mathbb{F}_2$

	2,1	3,1	4,1	5,2	6,1	7,1	8,4,3,1	9,1	10,3
11,2	12,3	13,4,3,1	14,5	15,1	16,5,3,1	17,3	18,3	19,5,2,1	20,3
21,2	22,1	23,5	24,4,3,1	25,3	26,4,3,1	27,5,2,1	28,1	29,2	30,1
31,3	32,7,3,2	33,10	34,7	35,2	36,9	37,6,4,1	38,6,5,1	39,4	40,5,4,3
41,3	42,7	43,6,4,3	44,5	45,4,3,1	46,1	47,5	48,5,3,2	49,9	50,4,3,2
51,6,3,1	52,3	53,6,2,1	54,9	55,7	56,7,4,2	57,4	58,19	59,7,4,2	60,1
61,5,2,1	62,29	63,1	64,4,3,1	65,18	66,3	67,5,2,1	68,9	69,6,5,2	70,5,3,1
71,6	72,10,9,3	73,25	74,35	75,6,3,1	76,21	77,6,5,2	78,6,5,3	79,9	80,9,4,2
81,4	82,8,3,1	83,7,4,2	84,5	85,8,2,1	86,21	87,13	88,7,6,2	89,38	90,27
91,8,5,1	92,21	93,2	94,21	95,11	96,10,9,6	97,6	98,11	99,6,3,1	100,15
101,7,6,1	102,29	103,9	104,4,3,1	105,4	106,15	107,9,7,4	108,17	109,5,4,2	110,33
111,10	112,5,4,3	113,9	114,5,3,2	115,8,7,5	116,4,2,1	117,5,2,1	118,33	119,8	120,4,3,1
121,18	122,6,2,1	123,2	124,19	125,7,6,5	126,21	127,1	128,7,2,1	129,5	130,3

Combinatorial (6)

6.1 Permutations

6.1.1 Matroids

MatroidIntersect.h

Description: Computes a set of maximum size which is independent in both graphic and colorful matroids, aka a spanning forest where no two edges are of the same color. In general, construct the exchange graph and find a shortest path. Can apply similar concept to partition matroid. Usage = MatroidIsect<Gmat,Cmat> M(sz(ed),Gmat(ed),Cmat(col))
Time: $\mathcal{O}\left(GI^{1.5}\right)$ calls to oracles, where G is size of ground set and I is size of independent set.

70 lines

```
struct Gmat { // graphic matroid
  int V = 0;
  vpi ed;
  DSU D;
  Gmat(vpi _ed) : ed(_ed) {
    map<int, int> m;
    each(t, ed) m[t.f] = m[t.s] = 0;
    each(t, m) t.s = V++;
    each(t, ed) t.f = m[t.f], t.s = m[t.s];
  }
  void clear() { D.init(V); }
  void ins(int i) { assert(D.unite(ed[i].f, ed[i].s)); }
  bool indep(int i) { return !D.sameSet(ed[i].f, ed[i].s); }
};
struct Cmat { // colorful matroid
  int C = 0;
  vi col;
  V<bool> used;
  Cmat(vi col) : col(col) { each(t, col) ckmax(C, t + 1); }
  void clear() { used.assign(C, 0); }
  void ins(int i) { used[col[i]] = 1; }
  bool indep(int i) { return !used[col[i]]; }
};
template <class M1, class M2> struct MatroidIsect {
  int n;
  V<bool> iset;
  M1 m1;
  M2 m2;
  bool augment() {
    vi pre(n + 1, -1);
    queue<int> q({n});
    while (sz(q)) {
      int x = q.ft;
      q.pop();
      if (iset[x]) {
        m1.clear();
        FOR(i, n) if (iset[i] && i != x) m1.ins(i);
        FOR(i, n)
        if (!iset[i] && pre[i] == -1 && m1.indep(i))
          pre[i] = x, q.push(i);
      } else {
        auto backE = [&]() { // back edge
          m2.clear();
          FOR(c, 2) FOR(i, n) if ((x == i || iset[i]) && (pre[i
              ] == -1) == c) {
            if (!m2.indep(i))
              return c ? pre[i] = x, q.push(i), i : -1;
            m2.ins(i);
          }
          return n;
        };
        for (int y; (y = backE()) != -1;)
          if (y == n) {
            for (; x != n; x = pre[x])
              iset[x] = !iset[x];
            return 1;
          }
      }
    }
    return 0;
```

```
  }
  MatroidIsect(int n, M1 m1, M2 m2) : n(n), m1(m1), m2(m2) {
    iset.assign(n + 1, 0);
    iset[n] = 1;
    m1.clear();
    m2.clear(); // greedily add to basis
    ROF(i, n) if (m1.indep(i) && m2.indep(i)) iset[i] = 1, m1.
        ins(i), m2.ins(i);
    while (augment())
      ;
  }
};
```

6.1.2 Factorial

IntPerm.h

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

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

7 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;
}
```

6.1.3 Derangements

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

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

6.1.4 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

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

6.2 Partitions and subsets

6.2.1 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

6.2.2 Binomials

multinomial.h

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

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;
}
```

6.3 General purpose numbers

6.3.1 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:

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

6.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

6.3.3 Eulerian numbers

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

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

6.3.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

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

6.3.5 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,

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

6.3.6 Labeled unrooted trees

on n vertices: n^{n-2}
on k existing trees of size n_i: $n_1n_2\cdots n_kn^{k-2}$
with degrees d_i: $(n-2)!/((d_1-1)!\cdots(d_n-1)!)$

6.3.7 Catalan numbers

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

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

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

- 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.

6.3.8 Pentagonal numbers

$$\prod_{n=1}^{+\infty}(1-x)^n = \sum_{k=-\infty}^{+\infty}(-1)^k x^{\frac{k*(3k-1)}{2}}$$

Graph (7)

7.1 Network flow

NetworkSimplex.h

Description: Solves min cost circulation in network, gives both flow values and dual. To get flow of edge look at capacity of back edge. To solve Min Cost Max Flow, add edge from tap to source of infinite capacity and cost -INF where $INF = 1+\sum_e abs(weight_e)$ To recover max flow, look at flow on this back edge and don't forget to update price To solve flow with demands on vertices, add an extra node and use INF cost again to enforce circulation. Dual values obtained are the same as if you didn't do transformation

Time: Should be fast $\mathcal{O}(M^2)$ worst case, add random shuffle of edges (permute indices) if time might be an issue. Should be faster than MCMF. DANGER : dual variables can cause problems if not in same connected component.

91 lines

```
template <class Flow, class Cost> struct NetworkSimplex {
  struct Edge {
    int nxt, to;
    Flow cap;
```

```
  Cost cost;
};
vector<Edge> edges;
vector<int> head, fa, fe, mark, cyc;
vector<Cost> dual;
int ti;

NetworkSimplex(int n)
    : head(n, 0), fa(n), fe(n), mark(n), cyc(n + 1), dual(n),
        ti(0) {
  edges.push_back({0, 0, 0, 0});
  edges.push_back({0, 0, 0, 0});
}

int addEdge(int u, int v, Flow cap, Cost cost) {
  assert(edges.size() % 2 == 0);
  int e = edges.size();
  edges.push_back({head[u], v, cap, cost});
  head[u] = e;
  edges.push_back({head[v], u, 0, -cost});
  head[v] = e + 1;
  return e;
}

void initTree(int x) {
  mark[x] = 1;
  for (int i = head[x]; i; i = edges[i].nxt) {
    int v = edges[i].to;
    if (!mark[v] and edges[i].cap) {
      fa[v] = x, fe[v] = i;
      initTree(v);
    }
  }
}

int phi(int x) {
  if (mark[x] == ti) return dual[x];
  return mark[x] = ti, dual[x] = phi(fa[x]) - edges[fe[x]].
      cost;
}

void pushFlow(int e, Cost &cost) {
  int pen = edges[e ^ 1].to, lca = edges[e].to;
  ti++;
  while (pen) mark[pen] = ti, pen = fa[pen];
  while (mark[lca] != ti) mark[lca] = ti, lca = fa[lca];

  int e2 = 0, f = edges[e].cap, path = 2, clen = 0;
  for (int i = edges[e ^ 1].to; i != lca; i = fa[i]) {
    cyc[++clen] = fe[i];
    if (edges[fe[i]].cap < f)
      f = edges[fe[e2 = i] ^ (path = 0)].cap;
  }
  for (int i = edges[e].to; i != lca; i = fa[i]) {
    cyc[++clen] = fe[i] ^ 1;
    if (edges[fe[i] ^ 1].cap <= f)
      f = edges[fe[e2 = i] ^ (path = 1)].cap;
  }
  cyc[++clen] = e;

  for (int i = 1; i <= clen; ++i) {
    edges[cyc[i]].cap -= f, edges[cyc[i] ^ 1].cap += f;
    cost += edges[cyc[i]].cost * f;
  }
  if (path == 2)  return;

  int laste = e ^ path, last = edges[laste].to, cur = edges[
      laste ^ 1].to;
  while (last != e2) {
    mark[cur]--;
    laste ^= 1;
    swap(laste, fe[cur]);
    swap(last, fa[cur]); swap(last, cur);
  }
}

Cost compute() {
  Cost cost = 0;
  initTree(0);
  mark[0] = ti = 2, fa[0] = cost = 0;
  int ncnt = edges.size() - 1;
  for (int i = 2, pre = ncnt; i != pre; i = i == ncnt ? 2 : i
      + 1)
    if (edges[i].cap and edges[i].cost < phi(edges[i ^ 1].to)
        - phi(edges[i].to))
      pushFlow(pre = i, cost);
  ++ti;
  for (int u = 0; u < (int)dual.size(); ++u)
    phi(u);
  return cost;
}
};
```

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.
Status: Tested on SPOJ FASTFLOW and SPOJ MATCHING, stress-tested
47 lines

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

GlobalMinCut.h

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

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

GomoryHu.h

Description: Given a list of edges representing an undirected flow graph, returns edges of the Gomory-Hu tree. The max flow between any pair of vertices is given by minimum edge weight along the Gomory-Hu tree path.
Time: $\mathcal{O}(V)$ Flow Computations
"PushRelabel.h" 13 lines

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

7.2 Matching

hopcroftKarp.h

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

```
struct HopcroftKarp {
  vector<int> g, l, r;
```

```
  int ans;
  HopcroftKarp(int n, int m, const vector<pair<int, int>> &e)
      : g(e.size()), l(n, -1), r(m, -1), ans(0) {
    std::vector<int> deg(n + 1);
    for (auto &[x, y] : e) deg[x]++;
    for (int i = 1; i <= n; i++) deg[i] += deg[i - 1];
    for (auto &[x, y] : e) g[--deg[x]] = y;

    std::vector<int> a, p, q(n);
    for (;;) {
      a.assign(n, -1), p.assign(n, -1);
      int t = 0;
      for (int i = 0; i < n; i++)
        if (l[i] == -1) q[t++] = a[i] = p[i] = i;

      bool match = false;
      for (int i = 0; i < t; i++) {
        int x = q[i];
        if (~l[a[x]]) continue;
        for (int j = deg[x]; j < deg[x + 1]; j++) {
          int y = g[j];
          if (r[y] == -1) {
            while (~y) r[y] = x, swap(l[x], y), x = p[x];
            match = true, ans++;
            break;
          }

          if (p[r[y]] == -1)
            q[t++] = y = r[y], p[y] = x, a[y] = a[x];
        }
      }

      if (!match) break;
    }
  }
};
```

MinimumVertexCover.h

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

`"DFSMatching.h"` 20 lines

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

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.

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

31 lines

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

GeneralMatching.h

Description: Matching for general graphs. Fails with probability N/mod.

Time: $\mathcal{O}\left(N^3\right)$

`"../numerical/MatrixInverse-mod.h"` 46 lines

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

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

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

  vi has(M, 1);
  vector<pii> ret;
  rep(it, 0, M / 2) {
    rep(i, 0, M) if (has[i]) rep(j, i + 1, M) if (A[i][j] &&
        mat[i][j]) {
      fi = i;
      fj = j;
      goto done;
    }
    assert(0);
  done:
    if (fj < N)
      ret.emplace_back(fi, fj);
    has[fi] = has[fj] = 0;
    rep(sw, 0, 2) {
      ll a = modpow(A[fi][fj], mod - 2);
      rep(i, 0, M) if (has[i] && A[i][fj]) {
        ll b = A[i][fj] * a % mod;
        rep(j, 0, M) A[i][j] = (A[i][j] - A[fi][j] * b) % mod;
      }
      swap(fi, fj);
    }
  }
  return ret;
}
```

7.3 DFS algorithms

SCC.h

Description: Finds strongly connected components in a directed graph. If vertices u, v belong to the same component, we can reach u from v and vice versa.

Usage: `scc(graph, [&](vi& v) { ... })` visits all components in reverse topological order. `comp[i]` holds the component index of a node (a component only has edges to components with lower index). `ncomps` will contain the number of components.

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

24 lines

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

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

BiconnectedComponents.h

Description: Finds all biconnected components in an undirected graph, and runs a callback for the edges in each. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle.

Usage: `int eid = 0; ed.resize(N);`
`for each edge (a,b) {`
`ed[a].emplace_back(b, eid);`
`ed[b].emplace_back(a, eid++); }`
`bicomps([&](const vi& edgelist) {...});`
Time: $\mathcal{O}(E+V)$

33 lines

```
vi num, st;
vector<vector<pii>> ed;
int Time;
template <class F> int dfs(int at, int par, F &f) {
  int me = num[at] = ++Time, e, y, top = me;
  for (auto pa : ed[at])
    if (pa.second != par) {
      tie(y, e) = pa;
      if (num[y]) {
        top = min(top, num[y]);
        if (num[y] < me)
          st.push_back(e);
      } else {
        int si = sz(st);
        int up = dfs(y, e, f);
        top = min(top, up);
        if (up == me) {
          st.push_back(e);
          f(vi(st.begin() + si, st.end()));
          st.resize(si);
        } else if (up < me)
          st.push_back(e);
        else { /* e is a bridge */
        }
      }
    }
  return top;
}

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

2sat.h

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

63 lines

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

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

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

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

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

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

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

EulerWalk.h

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

23 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()};
}
```

DominatorTree.h

Description: Builds dominator tree of digraph rooted at source dom[u] = children of u in dominator u, where u is ancestor of v iff all paths from source to v contain u
Time: $\mathcal{O}(E \log V)$

67 lines

```
struct DominatorTree {
  int n, sz = 0;
  vi dfn, pre, pt, semi, dsu, idom, best;
  vvi dom;

  DominatorTree(const vvi &g, ll source)
      : n(g.size()), dfn(n, -1), pre(n), pt(n), semi(n), dsu(n)
          , idom(n),
        best(n), dom(n) {
    vvi ginv(n);
    rep(i, 0, n) {
      for (ll j : g[i])
        ginv[j].push_back(i);
      dsu[i] = best[i] = semi[i] = i;
    }
    dfs(source, g);
    vvi mydom(n);
    tarjan(ginv, mydom);
    rep(i, 0, sz) for (ll d : mydom[i]) dom[pt[i]].push_back(pt
        [d]);
  }

  ll get(ll x) {
    if (x == dsu[x])
      return x;
    ll y = get(dsu[x]);
    if (semi[best[x]] > semi[best[dsu[x]]])
      best[x] = best[dsu[x]];
    return dsu[x] = y;
  }

  void dfs(ll u, const vvi &succ) {
    dfn[u] = sz;
    pt[sz++] = u;
    for (ll v : succ[u])
      if (dfn[v] < 0) {
        dfs(v, succ);
        pre[dfn[v]] = dfn[u];
      }
  }

  void tarjan(const vvi &pred, vvi &dom) {
    for (int j = sz - 1; j >= 1; --j) {
      ll u = pt[j];
      for (ll tv : pred[u])
        if (dfn[tv] >= 0) {
          ll v = dfn[tv];
          get(v);
          if (semi[best[v]] < semi[j])
            semi[j] = semi[best[v]];
        }
      dom[semi[j]].push_back(j);
```

```
    ll x = dsu[j] = pre[j];
    for (ll z : dom[x]) {
      get(z);
      if (semi[best[z]] < x)
        idom[z] = best[z];
      else
        idom[z] = x;
    }
    dom[x].clear();
  }
  rep(i, 1, sz) {
    if (semi[i] != idom[i])
      idom[i] = idom[idom[i]];
    dom[idom[i]].push_back(i);
  }
 }
};
```

7.4 Coloring

EdgeColoring.h

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

31 lines

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

7.5 Heuristics

MaximalCliques.h

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

12 lines

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

MaximumClique.h

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

49 lines

```
typedef vector<bitset<200>> vb;
struct Maxclique {
  double limit=0.025, pk=0;
  struct Vertex { int i, d=0; };
  typedef vector<Vertex> vv;
  vb e;
  vv V;
  vector<vi> C;
  vi qmax, q, S, old;
  void init(vv& r) {
    for (auto& v : r) v.d = 0;
    for (auto& v : r) for (auto j : r) v.d += e[v.i][j.i];
    sort(all(r), [](auto a, auto b) { return a.d > b.d; });
    int mxD = r[0].d;
    rep(i,0,sz(r)) r[i].d = min(i, mxD) + 1;
  }
  void expand(vv& R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (sz(R)) {
      if (sz(q) + R.back().d <= sz(qmax)) return;
      q.push_back(R.back().i);
      vv T;
      for(auto v:R) if (e[R.back().i][v.i]) T.push_back({v.i});
      if (sz(T)) {
        if (S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1, mnk = max(sz(qmax) - sz(q) + 1, 1);
        C[1].clear(), C[2].clear();
        for (auto v : T) {
          int k = 1;
          auto f = [&](int i) { return e[v.i][i]; };
          while (any_of(all(C[k]), f)) k++;
          if (k > mxk) mxk = k, C[mxk + 1].clear();
          if (k < mnk) T[j++].i = v.i;
          C[k].push_back(v.i);
        }
        if (j > 0) T[j - 1].d = 0;
        rep(k,mnk,mxk + 1) for (int i : C[k])
          T[j].i = i, T[j++].d = k;
        expand(T, lev + 1);
      } else if (sz(q) > sz(qmax)) qmax = q;
      q.pop_back(), R.pop_back();
    }
  }
  vi maxClique() { init(V), expand(V); return qmax; }
  Maxclique(vb conn) : e(conn), C(sz(e)+1), S(sz(C)), old(S) {
    rep(i,0,sz(e)) V.push_back({i});
  }
};
```

MaximumIndependentSet.h

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

7.6 Trees

LCA.h

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

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

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

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

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

CompressTree.h

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

"LCA.h" 21 lines

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

LinkCutTree.h

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

125 lines

```
struct SplayTree {
  struct Node {
    int ch[2] = {0, 0}, p = 0;
    int self = 0, path = 0;
    int sub = 0, vir = 0;
    bool flip = 0;
  };
  vector<Node> T;

  SplayTree(int n) : T(n + 1) {}

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

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

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

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

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

  int access(int x) {
    int u = x, v = 0;
    for (; u; v = u, u = T[u].p) {
      splay(u);
      int &ov = T[u].ch[1];
      T[u].vir += T[ov].sub;
      T[u].vir -= T[v].sub;
      ov = v; pull(u);
    }
    splay(x);
    return v;
  }

  void reroot(int x) {
    access(x);
    T[x].flip ^= 1;
    push(x);
  }

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

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

  int lca(int u, int v) {
    ++u, ++v;
    if (u == v) return u;
    access(u);
    int ret = access(v);
    return T[u].p ? ret : 0;
  }

  // subtree aggregate for u when footed at v
  int subTree(int u, int v) {
    ++u, ++v;
    reroot(v); access(u);
    return T[u].vir + T[u].self;
  }

  // puts v at root and path to u is in left child
  // returns path aggregate
  int path(int u, int v)
  {
    ++u, ++v;
    reroot(u);
    access(v);
    return T[v].path;
  }

  void update(int u, int v) {
    ++u;
    access(u);
    T[u].self = v;
    pull(u);
  }
};
```

DirectedMST.h

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

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

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

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

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

7.7 Math

7.7.1 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).

7.7.2 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 (8)

8.1 Geometric primitives

Point.h

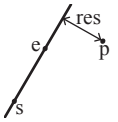
Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

28 lines

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

lineDistance.h

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

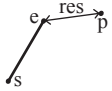


"Point.h" 4 lines

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

SegmentDistance.h

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



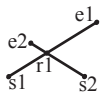
Usage: `Point<double> a, b(2,2), p(1,1);`
`bool onSegment = segDist(a,b,p) < 1e-10;`

"Point.h" 6 lines

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

SegmentIntersection.h

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



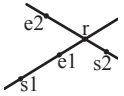
Usage: `vector<P> inter = segInter(s1,e1,s2,e2);`
`if (sz(inter)==1)`
`cout << "segments intersect at " << inter[0] << endl;`

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

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

lineIntersection.h

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



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

"Point.h" 8 lines

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

sideOf.h

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

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

"Point.h" 9 lines

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

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

OnSegment.h

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

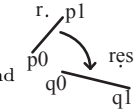
"Point.h" 3 lines

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

linearTransformation.h

Description:

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



"Point.h" 6 lines

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

LineProjectionReflection.h

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

"Point.h" 5 lines

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

Angle.h

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

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

35 lines

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

// Given two points, this calculates the smallest angle between
// them, i.e., the angle that covers the defined line segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
          make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b < a)};
}
```

8.2 Circles

CircleIntersection.h

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

"Point.h" 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" 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;
}
```

CircleLine.h

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

"Point.h" 9 lines

```
template<class P>
vector<P> circleLine(P c, double r, P a, P b) {
  P ab = b - a, p = a + ab * (c-a).dot(ab) / ab.dist2();
  double s = a.cross(b, c), h2 = r*r - s*s / ab.dist2();
  if (h2 < 0) return {};
  if (h2 == 0) return {p};
  P h = ab.unit() * sqrt(h2);
  return {p - h, p + h};
}
```

CirclePolygonIntersection.h

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

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

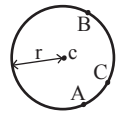
"../../content/geometry/Point.h" 19 lines

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

circumcircle.h

Description:

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



"Point.h" 9 lines

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

MinimumEnclosingCircle.h

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

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

"circumcircle.h" 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};
}
```

8.3 Polygons

InsidePolygon.h

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

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

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

"OnSegment.h", "Point.h", "SegmentDistance.h" 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;
}
```

PolygonArea.h

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

"Point.h" 6 lines

```
template<class T>
```

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

PolygonCenter.h

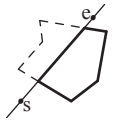
Description: Returns the center of mass for a polygon.
Time: $\mathcal{O}(n)$

"Point.h" 9 lines

```
typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (int i = 0, j = sz(v) - 1; i < sz(v); j = i++) {
    res = res + (v[i] + v[j]) * v[j].cross(v[i]);
    A += v[j].cross(v[i]);
  }
  return res / A / 3;
}
```

PolygonCut.h

Description:
Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.
Usage: vector<P> p = ...;
p = polygonCut(p, P(0,0), P(1,0));



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

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

PolygonUnion.h

Description: Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)
Time: $\mathcal{O}\left(N^2\right)$, where N is the total number of points

"Point.h", "sideOf.h" 33 lines

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

ConvexHull.h

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

"Point.h" 13 lines

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

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).

"Point.h" 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;
}
```

PointInsideHull.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" 14 lines

```
typedef Point<ll> P;

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

LineHullIntersection.h

Description: Line-convex polygon intersection. The polygon must be ccw and have no collinear points. lineHull(line, poly) returns a pair describing the intersection of a line with the polygon: $\bullet$ $(-1,-1)$ if no collision, $\bullet$ $(i,-1)$ if touching the corner i, $\bullet$ (i,i) if along side $(i,i+1)$, $\bullet$ (i,j) if crossing sides $(i,i+1)$ and $(j,j+1)$. In the last case, if a corner i is crossed, this is treated as happening on side $(i,i+1)$. The points are returned in the same order as the line hits the polygon. `extrVertex` returns the point of a hull with the max projection onto a line.
Time: $\mathcal{O}(\log n)$

"Point.h" 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;
}
```

8.4 Misc. Point Set Problems

ClosestPair.h

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

"Point.h" 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;
}
```

ManhattanMST.h

Description: Given N points, returns up to 4*N edges, which are guaranteed to contain a minimum spanning tree for the graph with edge weights w(p, q) = —p.x - q.x— + —p.y - q.y—. Edges are in the form (distance, src, dst). Use a standard MST algorithm on the result to find the final MST.
Time: $\mathcal{O}(N \log N)$

"Point.h" 23 lines

```
typedef Point<int> P;
vector<array<int, 3>> manhattanMST(vector<P> ps) {
  vi id(sz(ps));
  iota(all(id), 0);
  vector<array<int, 3>> edges;
  rep(k,0,4) {
    sort(all(id), [&](int i, int j) {
         return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;});
    map<int, int> sweep;
    for (int i : id) {
      for (auto it = sweep.lower_bound(-ps[i].y);
                it != sweep.end(); sweep.erase(it++)) {
        int j = it->second;
        P d = ps[i] - ps[j];
        if (d.y > d.x) break;
        edges.push_back({d.y + d.x, i, j});
      }
      sweep[-ps[i].y] = i;
    }
    for (P& p : ps) if (k & 1) p.x = -p.x; else swap(p.x, p.y);
  }
  return edges;
}
```

kdTree.h

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

"Point.h" 63 lines

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

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

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

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

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

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

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

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

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

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

FastDelaunay.h

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

"Point.h" 128 lines

```
struct quad_edge {
  int o = -1; // origin of the arc
  quad_edge *onext, *rot;
  bool mark = false;
  quad_edge() {}
  quad_edge(int o) : o(o) {}
  int d() { return sym()->o; } // destination of the arc
  quad_edge *sym() { return rot->rot; }
  quad_edge *oprev() { return rot->onext->rot; }
  quad_edge *lnext() { return sym()->oprev(); }
  static quad_edge *make_sphere(int a, int b) {
    array<quad_edge *, 4> q{
        {new quad_edge{a}, new quad_edge{}, new quad_edge{b},
            new quad_edge{}}};
    for (auto i = 0; i < 4; ++i)
      q[i]->onext = q[-i & 3], q[i]->rot = q[i + 1 & 3];
    return q[0];
  }
  static void splice(quad_edge *a, quad_edge *b) {
    swap(a->onext->rot->onext, b->onext->rot->onext);
    swap(a->onext, b->onext);
  }
  static quad_edge *connect(quad_edge *a, quad_edge *b) {
    quad_edge *q = make_sphere(a->d(), b->o);
    splice(q, a->lnext()), splice(q->sym(), b);
    return q;
  }
};

template <class T, class T_large, class F1, class F2>
bool delaunay_triangulation(const vector<Point<T>> &a, F1
     process_outer_face,
                                F2 process_triangles) {
  vector<int> ind(a.size());
  iota(ind.begin(), ind.end(), 0);
  sort(ind.begin(), ind.end(), [&](int i, int j) { return a[i]
       < a[j]; });
  ind.erase(unique(ind.begin(), ind.end(),
                   [&](int i, int j) { return a[i] == a[j]; }),
            ind.end());
  int n = (int)ind.size();
  if (n < 2)
    return {};
  auto circular = [&](Point<T> p, Point<T> a, Point<T> b, Point
       <T> c) {
    a = a - p, b = b - p, c = c - p;
    return ((T_large)a.dist2() * b.cross(c) + (T_large)b.dist2
         () * c.cross(a) +
            (T_large)c.dist2() * a.cross(b)) *
              (a.cross(b, c) > 0 ? 1 : -1) >
          0;
  };
  auto recurse = [&](auto self, int l, int r) -> array<
       quad_edge *, 2> {
    if (r - l <= 3) {
      quad_edge *p = quad_edge::make_sphere(ind[l], ind[l + 1])
          ;
      if (r - l == 2)
        return {p, p->sym()};
      quad_edge *q = quad_edge::make_sphere(ind[l + 1], ind[l +
           2]);
      quad_edge::splice(p->sym(), q);
      auto side = a[ind[l]].cross(a[ind[l + 1]], a[ind[l + 2]])
          ;
      quad_edge *c = side ? quad_edge::connect(q, p) : NULL;
      return {side < 0 ? c->sym() : p, side < 0 ? c : q->sym()}
          ;
    }
    int m = l + (r - l >> 1);
    auto [ra, A] = self(self, l, m);
    auto [B, rb] = self(self, m, r);
    while (a[B->o].cross(a[A->d()], a[A->o]) < 0 && (A = A->
         lnext()) ||
           a[A->o].cross(a[B->d()], a[B->o]) > 0 && (B = B->sym
                ()->onext))
      ;
    quad_edge *base = quad_edge::connect(B->sym(), A);
    if (A->o == ra->o)
      ra = base->sym();
    if (B->o == rb->o)
      rb = base;
#define valid(e) (a[e->d()].cross(a[base->d()], a[base->o]) >
     0)
#define DEL(e, init, dir) \
  quad_edge *e = init->dir; \
  if (valid(e)) \
    while (circular(a[e->dir->d()], a[base->d()], a[base->o], a
         [e->d()])) { \
```

```
      quad_edge *t = e->dir; \
      quad_edge::splice(e, e->oprev());\
      quad_edge::splice(e->sym(), e->sym()->oprev()); \
      delete e->rot->rot->rot; \
      delete e->rot->rot; \
      delete e->rot; \
      delete e; \
      e = t; \
    }
    while (true) {
      DEL(LC, base->sym(), onext);
      DEL(RC, base, oprev());
      if (!valid(LC) && !valid(RC))
        break;
      if (!valid(LC) ||
          valid(RC) && circular(a[RC->d()], a[RC->o], a[LC->d()
              ], a[LC->o]))
        base = quad_edge::connect(RC, base->sym());
      else
        base = quad_edge::connect(base->sym(), LC->sym());
    }
    return {ra, rb};
  };
  auto e = recurse(recurse, 0, n)[0];
  vector<quad_edge *> q = {e}, rem;
  while (a[e->onext->d()].cross(a[e->d()], a[e->o]) < 0)
    e = e->onext;
  vector<int> face;
  face.reserve(n);
  bool colinear = false;
#define ADD \
  { \
    quad_edge *c = e; \
    face.clear(); \
    do {  \
      c->mark = true; \
      face.push_back(c->o);  \
      q.push_back(c->sym()); \
      rem.push_back(c); \
      c = c->lnext();  \
    } while (c != e); \
  }
  ADD;
  process_outer_face(face);
  for (auto qi = 0; qi < (int)q.size(); ++qi) {
    if (!(e = q[qi])->mark) {
      ADD;
      colinear = false;
      process_triangles({face[0], face[1], face[2]});
    }
  }
  for (auto e : rem)
    delete e->rot, delete e;
  return !colinear;
}
```

HalfPlaneIntersect.h

Description: Computes convex polygon which represents intersection of half planes (each half plane is to the left of the segment) Careful with precision !

40 lines

```
const double EPS = 1e-9;
const double DINF = 1e100;
template <typename T> struct HalfPlane {
  Point<T> p, pq;
  T angle;
  HalfPlane() {}
  HalfPlane(Point<T> _p, Point<T> _q): p(_p), pq(_q - _p),
      angle(atan2(pq.y, pq.x)) {}
  bool operator<(HalfPlane &b) const { return angle < b.angle;
      }
  bool out(Point<T> q) { return cross(pq, q - p) < EPS; }
  Point<T> intersect(HalfPlane<T> l) {
    if (abs(cross(pq, l.pq)) < EPS) return Point<T>(DINF, DINF)
        ;
    return l.p + l.pq * (cross(p - l.p, pq) / cross(l.pq, pq));
  }
};
// Halfplane to the left of line

template <typename T> vector<Point<T>> intersect(vector<
    HalfPlane<T>> b) {
  vector<Point<T>> bx = {{DINF, DINF}, {-DINF, DINF}, {-DINF, -
      DINF}, {DINF, -DINF}};
  for (int i = 0; i < 4; ++i) b.emplace_back(bx[i], bx[(i + 1)
      % 4]);
  sort(b.begin(), b.end());
  int n = b.size(), q = 1, h = 0;
  vector<HalfPlane<T>> c(b.size() + 10);
  for (int i = 0; i < n; ++i) {
    while (q < h and b[i].out(c[h].intersect(c[h - 1]))) h--;
    while (q < h and b[i].out(c[q].intersect(c[q + 1]))) q++;
    c[++h] = b[i];
    if (q < h and abs(cross(c[h].pq, c[h - 1].pq)) < EPS) {
      if (dot(c[h].pq, c[h - 1].pq) <= 0) return {};
      h--;
      if (b[i].out(c[h].p))c[h] = b[i];
    }
  }
  while (q < h - 1 and c[q].out(c[h].intersect(c[h - 1]))) h--;
  while (q < h - 1 and c[h].out(c[q].intersect(c[q + 1]))) q++;
  if (h - q <= 1) return {};
  c[h + 1] = c[q];
  vector<Point<T>> s;
  for (int i = q; i < h + 1; ++i) s.push_back(c[i].intersect(c[
      i + 1]));
  return s;
}
```

8.5 3D

PolyhedronVolume.h

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

6 lines

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

Point3D.h

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

32 lines

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z); }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z); }
  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P operator/(T d) const { return P(x/d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)dist2()); }
  //Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in interval [0, pi]
  double theta() const { return atan2(sqrt(x*x+y*y),z); }
  P unit() const { return *this/(T)dist(); } //makes dist()=1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit(); }
  //returns point rotated 'angle' radians ccw around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit();
    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
  }
};
```

3dHull.h

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

Time: $\mathcal{O}\left(n^2\right)$

"Point3D.h" 49 lines

```
typedef Point3D<double> P3;

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

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

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

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

```
    }
  }
  for (F& it : FS) if ((A[it.b] - A[it.a]).cross(
    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
  return FS;
};
```

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

8 lines

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

Strings (9)

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: $\mathcal{O}(n)$

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: $\mathcal{O}(n)$

12 lines

```
vi Z(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: $\mathcal{O}(N)$

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: $\mathcal{O}(N)$

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

23 lines

```
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++);
  }
};
```

SuffixArrayLinear.h

Description: Suffix Array en temps lineaire. À intégrer dans structure non linéaire pour obtenir LCP. Il suffit d'insérer un s.size() au début du vecteur de retour

Time: Linear

51 lines

```
vector<int> sa_is(const vector<int> &s, int lim = 256) {
  int n = s.size(); if (!n) return {};
  vector<int> sa(n); vector<bool> ls(n);
  for (int i = n - 2; i >= 0; --i)
    ls[i] = s[i] == s[i + 1] ? ls[i + 1] : s[i] < s[i + 1];
  vector<int> sum_l(lim), sum_s(lim);
  for (int i = 0; i < n; ++i)
    (ls[i] ? sum_l[s[i] + 1] : sum_s[s[i]])++;
  for (int i = 0; i < lim; ++i) {
    if (i) sum_l[i] += sum_s[i - 1];
    sum_s[i] += sum_l[i];
  }
  auto induce = [&](const vector<int> &lms) {
    fill(sa.begin(), sa.end(), -1);
    vector<int> buf = sum_s;
    for (int d : lms) if (d != n) sa[buf[s[d]]++] = d;
    buf = sum_l;
    sa[buf[s[n - 1]]++] = n - 1;
    for (int i = 0; i < n; ++i) {
      int v = sa[i] - 1;
      if (v >= 0 && !ls[v]) sa[buf[s[v]]++] = v;
    }
    buf = sum_l;
    for (int i = n - 1; i >= 0; --i) {
      int v = sa[i] - 1; if (v >= 0 && ls[v]) sa[--buf[s[v] +
          1]] = v;
    }
  };
  vector<int> lms_map(n + 1, -1), lms; int m = 0;
  for (int i = 1; i < n; ++i)  if (!ls[i - 1] && ls[i])
      lms_map[i] = m++, lms.push_back(i);
  induce(lms);
  vector<int> sorted_lms;
  for (auto &v : sa)
    if (lms_map[v] != -1)sorted_lms.push_back(v);
  vector<int> rec_s(m); int rec_upper = 0;
  for (int i = 1; i < m; ++i) {
    int l = sorted_lms[i - 1], r = sorted_lms[i];
    int end_l = lms_map[l] + 1 < m ? lms[lms_map[l] + 1] : n;
    int end_r = lms_map[r] + 1 < m ? lms[lms_map[r] + 1] : n;
    bool same = 0;
    if (end_l - l == end_r - r) {
      for (; l < end_l && s[l] == s[r]; ++l, ++r);
      if (l != n && s[l] == s[r]) same = 1;
    }
    rec_s[lms_map[sorted_lms[i]]] = (rec_upper += !same);
  }
  vector<int> rec_sa = sa_is(rec_s, rec_upper + 1);
  for (int i = 0; i < m; ++i) sorted_lms[i] = lms[rec_sa[i]];
  induce(sorted_lms);
  return sa;
}
```

SuffixTree.h

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).

Time: $\mathcal{O}(26N)$

50 lines

```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
```

```
  string a; // v = cur node, q = cur position
  int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;

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

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

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

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 $O(26N)$, where N = sum of length of patterns. find(x) is $O(N)$, where N = length of x. findAll is $O(NM)$.

70 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;
  }
};
```

PalindromicTree.h

Description: You add characters one by one with function add. Builds a tree of all distinct palindromes, each node represents a palindrome and if node represents X, transition a goes to aXa. Backpointers for max suffix palindrome

Time: Linear in number of added characters, $\mathcal{O}(alphabet * length)$ in memory

65 lines

```
const int MAX_TRA = 26; // number of transitions
int normalize(char c) { // normalize `c` in [0, ..., MAX_TRA-1]
  return c - 'a';
}

struct PalindromicTree {
  // len = length of the palindrome, sufCount = number of
      suffix palindrome
  // link = node of the maximum suffix palindrome, occAsMax =
      used in computeOcc
  struct Node {
    int len, link, sufCount, occAsMax, next[MAX_TRA];
    Node() : occAsMax(0) { FOR(i, MAX_TRA) next[i] = -1; }
  };

  string s;
  vector<Node> t; // tree. node 0: root with len -1, node 1:
      root with len 0
  int suff;       // node of the current maximum suffix
      palindrome

  PalindromicTree() {
    suff = 1;
    t.resize(2);
    t[0].len = -1;
    t[0].link = 0;
    t[0].sufCount = 0;
    t[1].len = 0;
    t[1].link = 0;
    t[1].sufCount = 0;
  }

  int suffix(int x, int i) {
    while (i - t[x].len - 1 < 0 || s[i - t[x].len - 1] != s[i])
      x = t[x].link;
    return x;
  }

  // returns true if `c` created a new distinct palindrome
  bool add(char c) {
    int let = normalize(c);
    int pos = s.size();
    s.push_back(c);
    suff = suffix(suff, pos);

    bool newNode = t[suff].next[let] == -1;
    if (newNode) {
      int x = t.size();
      t[suff].next[let] = x;
      t.emplace_back();
      t[x].len = t[suff].len + 2;
      t[x].link = suff == 0 ? 1 : t[suffix(t[suff].link, pos)].
          next[let];
      t[x].sufCount = 1 + t[t[x].link].sufCount;
    }
    int x = t[suff].next[let];
    ++t[x].occAsMax;
    suff = x;
    return newNode;

  }

  // occ[i] = number of occurences of the node i in the string
  vi computeOcc() {
    vi occ(t.size());
    FOR(i, t.size()) occ[i] = t[i].occAsMax;
```

```
    FORD(i, 0, occ.size()) occ[t[i].link] += occ[i];
    return occ;
  }
};
```

SuffixAutomaton.h

Description: Suffix automaton, automaton which accepts all suffixes of a given string. aut[state][char] gives transition and isFinal[state] says if state is final. len[state] is the length of string corresponding to that state.

Time: Linear

59 lines

```
const int MAX_TRA = 26; // number of transitions
int normalize(char c) { // normalize 'c' in [0, ..., MAX_TRA-1]
  return c - 'a';
}

struct SuffixAutomaton {
  array<int, MAX_TRA> defNode;
  vector<array<int, MAX_TRA>> aut;
  vector<int> len, link;
  vb isFinal;
  int last;

  int insert(int le, int li, int cp = -1) {
    len.pb(le);
    link.pb(li);
    aut.pb(cp == -1 ? defNode : aut[cp]);
    return aut.size() - 1;
  }

  SuffixAutomaton() {
    FOR(i, MAX_TRA) defNode[i] = -1;
    last = insert(0, -1);
  }

  void reserve(int n) {
    aut.reserve(2 * n);
    link.reserve(2 * n);
    len.reserve(2 * n);
  }

  void add(char cc) {
    int c = normalize(cc);
    int p, cur = insert(len[last] + 1, -2);
    for (p = last; p != -1 && aut[p][c] == -1; p = link[p]) {
      aut[p][c] = cur;
    }
    if (p != -1) {
      int q = aut[p][c];
      if (len[q] == len[p] + 1) {
        link[cur] = q;
      } else {
        int qq = insert(len[p] + 1, link[q], q);
        link[q] = link[cur] = qq;
        for (; p != -1 && aut[p][c] == q; p = link[p]) {
          aut[p][c] = qq;
        }
      }
    } else {
      link[cur] = 0;
    }
    last = cur;
  }

  void computeFinals() {
    isFinal.assign(aut.size(), false);
    for (int cur = last; cur >= 0; cur = link[cur])
      isFinal[cur] = true;
  }
};
```

Various (10)

10.1 Misc. algorithms

TernarySearch.h

Description: Find the smallest i in $[a, b]$ that maximizes $f(i)$, assuming that $f(a) < \ldots < f(i) \geq \cdots \geq f(b)$. To reverse which of the sides allows non-strict inequalities, change the $<$ marked with (A) to $<=$, and reverse the loop at (B). To minimize f, change it to $>$, also at (B).

Usage: `int ind = ternSearch(0,n-1,[&](int i){return a[i];});`

Time: $\mathcal{O}(\log(b-a))$

11 lines

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

10.2 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 $f(b,c) \leq f(a,d)$ and $f(a,c) + f(b,d) \leq f(a,d) + f(b,c)$ for all $a \leq b \leq c \leq d$. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.

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

FastMod.h

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

9 lines

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

FastInput.h

Description: Read an integer from stdin. Usage requires your program to pipe in input from file.

Usage: `./a.out < input.txt`

Time: About 5x as fast as cin/scanf.

17 lines

```
inline char gc() { // like getchar()
  static char buf[1 << 16];
  static size_t bc, be;
  if (bc >= be) {
    buf[0] = 0, bc = 0;
    be = fread(buf, 1, sizeof(buf), stdin);
  }
  return buf[bc++]; // returns 0 on EOF
}

int readInt() {
  int a, c;
  while ((a = gc()) < 40);
  if (a == '-') return -readInt();
  while ((c = gc()) >= 48) a = a * 10 + c - 480;
  return a - 48;
}
```

10.3 Techniques

min-weight vertex cover in a bipartite graph: partition into A and B. add edges $s \to A$ with capacities $w(A)$ and edges $B \to t$ with capacities $w(B)$. add edges of capacity ∞ from A to B where there are edges in the graph. answer is maxflow. the vertex cover is the set of nodes that are adjacent to cut edges, or alternatively, the left-side nodes NOT reachable from the source and the right-side edges reachable from the source (in the residual network).

Bipartite matching with weights on the left-hand nodes, minimizing the matched weight sum: sort left-hand nodes ascending by weight, then just use the normal bipartite matching algorithm (Kuhn's)

Poset width / partition into maximum number of chains: Duplicate each element in $\{0, \ldots, n-1\}$, add edge $(u, n+v)$ for $u < v$. Edges in maximum matching in the resulting bipartite graph correspond to chain edges. Width is n - max matching. For weighted nodes, duplicate elements so they form an antichain.

Erdős–Gallai theorem: A sequence of non-negative integers $d_1 \geq \cdots \geq d_n$ can be represented as the degree sequence of a finite simple graph on n vertices if and only if $d_1 + \cdots + d_n$ is even and $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k)\ \forall\ 1 \leq k \leq n$

Capacity constraints without sink:feasible flow in a network with both upper and lower capacity constraints, no source or sink : capacity are changed to upperbound-lowerbound. Add a new source and a sink. let M[v] = (sum of lowerbounds of ingoing edges to v) - (sum of lowserbounds of outgoing edges from v). For all v, if M[v]¿0 then add edge (S,v) with capacity M, otherwise add (v,T) with capacity -M. If all outgoing edges from S are full, then a feasible flow exists, it is the flow plus the original lowerbounds.

Capacity constraints with sink: feasible flow in a network with both upper and lower capacity constraints, with source s and sink t: add edge (t,s) with capacity infinity. Binary search for the lower bound, check whether a feasible exists for a network WITHOUT source or sink (B).