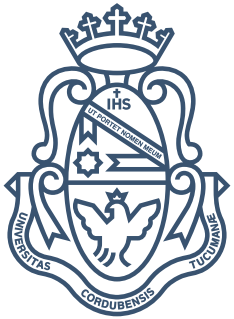


Universidad Nacional De Córdoba

Gracias Mateo

Antonio Mondejar, Pietro Palombini, Iván Renison

2024-12-08

Contest (1)

template.cpp — 18 lines

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

#define fst first
#define snd second
#define pb push_back
#define fore(i, a, b) for (ll i = a, gmat = b; i < gmat; i++)
#define ALL(x) x.begin(), x.end()
#define SZ(x) (ll)(x).size()
#define mset(a, v) memset((a), (v), sizeof(a))
typedef long long ll;
typedef pair<ll, ll> ii;
typedef vector<ll> vi;

int main() {
  cin.tie(0)->sync_with_stdio(0);

}
```

hash.sh — 3 lines

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

problemInteraction.sh — 3 lines

```
# For interactive problems
mkfifo fifo
(./solution < fifo) | (./interactor > fifo)
```

tester.py — 15 lines

```
from os import system

it = 100000
for t in range(it):
  system("./caseGen > in")
  system("./good < in > o")
  system("./bad < in > o2")
  x = open("o", "r").read().strip().split()
  y = open("o2", "r").read().strip().split()

  for i in range(len(x)):
    if (x[i] != y[i]):
      print("FAILED!!!!!", i + 1)
      exit(0)
  print("ok", t + 1)
```

troubleshoot.txt — 52 lines

```
Pre-submit:
Write a few simple test cases if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all data structures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a teammate.
Ask the teammate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a teammate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your teammates think about your algorithm?

Memory limit exceeded:
What is the max amount of memory your algorithm should need?
Are you clearing all data structures between test cases?
```

Mathematics (2)

2.1 Equations

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

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

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

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 Recurrences

If $a_n=c_1a_{n-1}+\cdots+c_ka_{n-k}$, and $r_1,\ldots,r_k$ are distinct roots of $x^k-c_1x^{k-1}-\cdots-c_k$, there are $d_1,\ldots,d_k$ s.t.

$$a_n=d_1r_1^n+\cdots+d_kr_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n=(d_1n+d_2)r^n$.

2.3 Trigonometry

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

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

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

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

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

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

2.4 Geometry

2.4.1 Pick's theorem

Given a simple polygon with vertices with integer coordinates, its area is $I+\frac{B}{2}-1$, where I is the number of integer points inside the polygon and B is the number of integer points on the boundary.

2.4.2 Triangles

Side lengths: a,b,c

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

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

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

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

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

Length of bisector (divides angles in two):

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

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

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

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

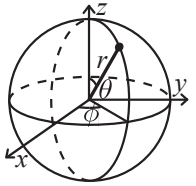
2.4.3 Quadrilaterals

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

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

2.4.4 Spherical coordinates

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



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

2.5 Derivatives/Integrals

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

Integration by parts:

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

2.6 Sums

$$c^a+c^{a+1}+\cdots+c^b=\frac{c^{b+1}-c^a}{c-1},c\neq1$$

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

2.7 Series

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

$$\ln(1+x)=x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}+\ldots,(-1<x\leq1)$$

$$\sqrt{1+x}=1+\frac{x}{2}-\frac{x^2}{8}+\frac{2x^3}{32}-\frac{5x^4}{128}+\ldots,(-1\leq x\leq1)$$

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

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

2.8 Probability theory

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

Expectation is linear:

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

For independent X and Y,

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

2.8.1 Discrete distributions

Binomial distribution

The number of successes in n independent yes/no experiments, each which yields success with probability p is $\text{Bin}(n, p)$, $n = 1, 2, \ldots$, $0 \le p \le 1$.

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\mu = np,\ \sigma^2 = np(1-p)$$

$\text{Bin}(n, p)$ is approximately $\text{Po}(np)$ for small p.

First success distribution

The number of trials needed to get the first success in independent yes/no experiments, each which yields success with probability p is $\text{Fs}(p)$, $0 \le p \le 1$.

$$p(k) = p(1-p)^{k-1},\ k = 1, 2, \ldots$$

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

Poisson distribution

The number of events occurring in a fixed period of time t if these events occur with a known average rate κ and independently of the time since the last event is $\text{Po}(\lambda)$, $\lambda = t\kappa$.

$$p(k) = e^{-\lambda}\frac{\lambda^k}{k!}, k = 0, 1, 2, \ldots$$

$$\mu = \lambda,\ \sigma^2 = \lambda$$

2.8.2 Continuous distributions

Uniform distribution

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

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

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

Exponential distribution

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

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

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

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

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

2.9 Markov chains

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

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

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

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

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

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

ac5104, 16 lines

```
#include "ext/pb_ds/assoc_container.hpp"
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<ll> t, t2; t.insert(8);
  auto it = t.insert(10).fst;
  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).

23cd06, 7 lines

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

rope.h

Description: Sequence with O(log(n)) random access, insert, erase at any position. Very high constant factors.
Usage:

```
s.pb(x);
s.insert(i,r) // insert rope r at position i
s.erase(i,k) // erase subsequence [i,i+k)
s.substr(i,k) // return a new rope
s[i] // access ith element (cannot modify)
s.mutable_reference_at(i)
s.begin() and s.end() are const iterators
s.mutable_begin(), s.mutable_end()
```

4cbb8d, 3 lines

```
#include <ext/rope>
using namespace __gnu_cxx;
rope<ll> s;
```

SegmentTree.h

Description: Zero-indexed max-tree. Bounds are inclusive to the left and exclusive to the right. Can be changed by modifying T, f and neut.
Time: $\mathcal{O}(\log N)$

921f55, 19 lines

```
struct Tree {
  typedef ll T;
  static constexpr T neut = LONG_LONG_MIN;
  T f(T a, T b) { return max(a, b); } // (any associative fn)
  vector<T> s; ll n;
  Tree(ll n = 0, T def = neut) : s(2*n, def), n(n) {}
  void upd(ll pos, T val) {
    for (s[pos += n] = val; pos /= 2;)
      s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
  }
  T query(ll b, ll e) { // query [b, e)
    T ra = neut, rb = neut;
    for (b += n, e += n; b < e; b /= 2, e /= 2) {
      if (b % 2) ra = f(ra, s[b++]);
      if (e % 2) rb = f(s[--e], rb);
    }
    return f(ra, rb);
  }
};
```

LazySegmentTree.h

Description: Segment tree with ability to add values of large intervals, and compute the sum of intervals. Ranges are [s, e). Can be changed to other things.
Usage: `Tree st(n);`
`st.init(x);`
`st.upd(s, e, v);`
`st.query(s, e);`
Time: $\mathcal{O}(\log N)$.

0fc20e, 53 lines

```
typedef ll T; typedef ll L; // T: data type, L: lazy type
const T tneut = 0; const L lneut = 0; // neutrals
T f(T a, T b) { return a + b; } // associative
T apply(T v, L l, ll s, ll e) { // new st according to lazy
  return v + l * (e - s); }
L comb(L a, L b) { return a + b; } // cumulative effect of lazy

struct Tree { // example: range sum with range addition
  ll n;
  vector<T> st;
  vector<L> lazy;
  Tree(ll n) : n(n), st(4*n, tneut), lazy(4*n, lneut) {}
  Tree(vector<T> &a) : n(SZ(a)), st(4*n), lazy(4*n, lneut) {
    init(1, 0, n, a);
  }
  void init(ll k, ll s, ll e, vector<T> &a) {
    lazy[k] = lneut;
    if (s + 1 == e) { st[k] = a[s]; return; }
    ll m = (s + e) / 2;
    init(2*k, s, m, a), init(2*k+1, m, e, a);
    st[k] = f(st[2*k], st[2*k+1]);
  }
  void push(ll k, ll s, ll e) {
    if (lazy[k] == lneut) return; // if neutral, nothing to do
    st[k] = apply(st[k], lazy[k], s, e);
    if (s + 1 < e) { // propagate to children
      lazy[2*k] = comb(lazy[2*k], lazy[k]);
      lazy[2*k+1] = comb(lazy[2*k+1], lazy[k]);
    }
    lazy[k] = lneut; // clear node lazy
  }
  void upd(ll k, ll s, ll e, ll a, ll b, L v) {
    push(k, s, e);
    if (s >= b || e <= a) return;
    if (s >= a && e <= b) {
      lazy[k] = comb(lazy[k], v); // accumulate lazy
      push(k, s, e);
      return;
    }
    ll m = (s + e) / 2;
    upd(2*k, s, m, a, b, v), upd(2*k+1, m, e, a, b, v);
    st[k] = f(st[2*k], st[2*k+1]);
  }
  T query(ll k, ll s, ll e, ll a, ll b) {
    if (s >= b || e <= a) return tneut;
    push(k, s, e);
    if (s >= a && e <= b) return st[k];
    ll m = (s + e) / 2;
    return f(query(2*k, s, m, a, b),query(2*k+1, m, e, a, b));
  }
  void upd(ll a, ll b, L v) { upd(1, 0, n, a, b, v); }
  T query(ll a, ll b) { return query(1, 0, n, a, b); }
};
```

MemoryLazySegmentTree.h

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

"../various/BumpAllocator.h" a87495, 53 lines

```
const ll inf = 1e18;
struct Node {
  typedef ll T; // data type
  struct L { ll toset, toadd; }; // lazy type
  const T tneut = -inf;     // neutral elements
  const L lneut = {inf, 0};
  T f (T a, T b) { return max(a, b); } // (any associative fn)
  T apply (T a, L b) {
    return b.toset != inf ? b.toset + b.toadd : a + b.toadd;
  } // Apply lazy
  L comb(L a, L b) {
    if (b.toset != inf) return b;
    return {a.toset, a.toadd + b.toadd};
  } // Combine lazy

  Node *l = 0, *r = 0;
  ll lo, hi; T val = tneut; L lazy = lneut;
  Node(ll lo,ll hi):lo(lo),hi(hi) {} //Large interval of tneut
  Node(vector<T>& v, ll lo, ll hi) : lo(lo), hi(hi) {
    if (lo + 1 < hi) {
      ll mid = lo + (hi - lo)/2;
      l = new Node(v, lo, mid); r = new Node(v, mid, hi);
      val = f(l->val, r->val);
    }
    else val = v[lo];
  }
  T query(ll L, ll R) {
    if (R <= lo || hi <= L) return tneut;
    if (L <= lo && hi <= R) return apply(val, lazy);
    push();
    return f(l->query(L, R), r->query(L, R));
  }
  void upd(ll Le, ll Ri, L x) {
    if (Ri <= lo || hi <= Le) return;
    if (Le <= lo && hi <= Ri) lazy = comb(lazy, x);
    else {
      push(), l->upd(Le, Ri, x), r->upd(Le, Ri, x);
      val = f(l->query(lo, hi), r->query(lo, hi));
    }
  }
  void set(ll L, ll R, ll x) { upd(L, R, {x, 0}); }
  void add(ll L, ll R, ll x) { upd(L, R, {inf, x}); }
  void push() {
    if (!l) {
      ll mid = lo + (hi - lo)/2;
      l = new Node(lo, mid), r = new Node(mid, hi);
    }
    l->lazy = comb(l->lazy, lazy);
    r->lazy = comb(r->lazy, lazy);
    lazy = lneut;
    val = f(l->query(lo, hi), r->query(lo, hi));
  }
};
```

PersistentSegmentTree.h

Description: Max segment tree in which each update creates a new version of the tree and you can query and update on any version of the tree. Can be changed by modifying T, f and unit.
Usage: `Tree rmq(n);`
`ver = rmq.init(xs);`
`new_ver = rmq.upd(ver, i, x);`
`rmq.query(ver, l, u);`
Time: $\mathcal{O}(\log N)$

83b8e4, 43 lines

```
struct Tree {
  typedef ll T;
  static constexpr T neut = LONG_LONG_MIN;
  T f(T a, T b) { return max(a, b); } // (any associative fn)

  vector<T> st;
  vi L, R;
  ll n, rt;
  Tree(ll n) : st(1, neut), L(1), R(1), n(n), rt(0) {}
  ll new_node(T v, ll l, ll r) {
    st.pb(v), L.pb(l), R.pb(r);
    return SZ(st) - 1;
  }
  // not necessary in most cases
  ll init(ll s, ll e, vector<T>& a) {
    if (s + 1 == e) return new_node(a[s], 0, 0);
    ll m = (s + e) / 2, l = init(s, m, a), r = init(m, e, a);
    return new_node(f(st[l], st[r]), l, r);
  }
  ll upd(ll k, ll s, ll e, ll p, T v) {
    ll ks = new_node(st[k], L[k], R[k]);
    if (s + 1 == e) {
      st[ks] = v;
      return ks;
    }
    ll m = (s + e) / 2;
    if (p < m) L[ks] = upd(L[ks], s, m, p, v);
    else R[ks] = upd(R[ks], m, e, p, v);
    st[ks] = f(st[L[ks]], st[R[ks]]);
    return ks;
  }
  T query(ll k, ll s, ll e, ll a, ll b) {
    if (e <= a || b <= s) return neut;
    if (a <= s && e <= b) return st[k];
    ll m = (s + e) / 2;
    return f(query(L[k], s, m, a, b), query(R[k], m, e, a, b));
  }
  ll init(vector<T>& a) { return init(0, n, a); }
  ll upd(ll ver, ll p, T v) {return rt = upd(ver, 0, n, p, v);}
  // upd on last root
  ll upd(ll p, T v) { return upd(rt, p, v); }
  T query(ll ver, ll a, ll b) {return query(ver, 0, n, a, b);}
};
```

SegmentTree2d.h

Description: Query sum of area ans make point updates. Bounds are inclusive to the left and exclusive to the right. Can be changed by modifying T, f and unit.
Time: $\mathcal{O}(\log N)$

4f243e, 37 lines

```
struct Tree2 {
```

```
  typedef ll T;
  static constexpr T neut = 0;
  T f(T a, T b) { return a + b; } // associative & commutative

  ll n, m;
  vector<vector<T>> a, st;
  Tree2(ll n, ll m) : n(n), m(m), a(n, vector<T>(m)),
      st(2 * n, vector<T>(2 * m)) {
    fore(i, 0, n) fore(j, 0, m) st[i + n][j + m] = a[i][j];
    fore(i, 0, n) for (ll j = m; --j;)
      st[i + n][j] = f(st[i + n][2 * j], st[i + n][2 * j + 1]);
    for (ll i = n; --i;) fore(j, 0, 2 * m)
      st[i][j] = f(st[2 * i][j], st[2 * i + 1][j]);
  }
  void upd(ll x, ll y, T v) {
    st[x + n][y + m] = v;
    for (ll j = y + m; j > 1; j /= 2)
      st[x + n][j / 2] = f(st[x + n][j], st[x + n][j ^ 1]);
    for (x += n; x > 1; x /= 2) for (ll j = y + m; j; j /= 2)
      st[x / 2][j] = f(st[x][j], st[x ^ 1][j]);
  }
  T query(ll x0, ll x1, ll y0, ll y1) { // [x0, x1) * [y0, y1)
    T r = neut;
    for (x0 += n, x1 += n; x0 < x1; x0 /= 2, x1 /= 2) {
      ll t[4], q = 0;
      if (x0 & 1) t[q++] = x0++;
      if (x1 & 1) t[q++] = --x1;
      fore(k, 0, q)
        for (ll j0 = y0+m, j1 = y1+m; j0 < j1; j0/=2, j1/=2) {
          if (j0 & 1) r = f(r, st[t[k]][j0++]);
          if (j1 & 1) r = f(r, st[t[k]][--j1]);
        }
    }
    return r;
  }
};
```

UnionFind.h

Description: Disjoint-set data structure.
Time: $\mathcal{O}(\alpha(N))$

d49635, 14 lines

```
struct UF {
  vi e;
  UF(ll n) : e(n, -1) {}
  bool sameSet(ll a, ll b) { return find(a) == find(b); }
  ll size(ll x) { return -e[find(x)]; }
  ll find(ll x) { return e[x] < 0 ? x : e[x] = find(e[x]); }
  bool join(ll a, ll b) {
    a = find(a), b = find(b);
    if (a == b) return false;
    if (e[a] > e[b]) swap(a, b);
    e[a] += e[b], e[b] = a;
    return true;
  }
};
```

UnionFindRollback.h

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

41c6ec, 20 lines

```
struct RollbackUF {
  vi e; vector<ii> st;
  RollbackUF(ll n) : e(n, -1) {}
  ll size(ll x) { return -e[find(x)]; }
  ll find(ll x) { return e[x] < 0 ? x : find(e[x]); }
  ll time() { return SZ(st); }
  void rollback(ll t) {
    for (ll i = time(); i --> t;)
      e[st[i].fst] = st[i].snd;
    st.resize(t);
  }
  bool join(ll a, ll b) {
    a = find(a), b = find(b);
    if (a == b) return false;
    if (e[a] > e[b]) swap(a, b);
    st.pb({a, e[a]}), st.pb({b, e[b]});
    e[a] += e[b], e[b] = a;
    return true;
  }
};
```

UnionFindRollbackStore.h

Description: Disjoint-set data structure, with undo and support for storing additional data in each component and global data. Default operations and data are for sum in each component and count of components.
Time: $\mathcal{O}(\log(N))$

9207a8, 34 lines

```
typedef ll T; // Global data
typedef ll D; // Component data
struct RSUF {
  T ans; // Global data initial value, set in constructor
  void merge(D& large, const D& small) {
    large = large + small, ans--;
  }

  ll n;
  vi e; vector<D> d;
  vector<tuple<ll,ll,ll,D,T>> st;
  RSUF(ll n) : ans(n), n(n), e(n, -1), d(n) {}
  RSUF(vector<D>& d) : ans(SZ(d)), n(SZ(d)), e(n,-1), d(d) {}
  ll size(ll x) { return -e[find(x)]; }
  ll find(ll x) { return e[x] < 0 ? x : find(e[x]); }
  ll time() { return SZ(st); }
  D get(ll x) { return d[find(x)]; }
  void rollback(ll t) {
    while (SZ(st) > t) {
      auto [a, b, s, v, t] = st.back();
      st.pop_back();
      d[a] = v, e[a] -= e[b] = s, ans = t;
    }
  }
  bool join(ll a, ll b) {
    a = find(a), b = find(b);
    if (a == b) return false;
    if (e[a] > e[b]) swap(a, b);
    st.pb({a, b, e[b], d[a], ans});
    merge(d[a], d[b]);
    e[a] += e[b], e[b] = a;
    return true;
  }
};
```

SubMatrix.h

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

2bb68d, 13 lines

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

Matrix.h

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

ca79eb, 32 lines

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

LineContainer.h

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

8ec1c7, 31 lines

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

struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b);
  }
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, 0;
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
```

```
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

Treap.h

Description: A short self-balancing tree. It acts as a sequential container with log-time splits, joins, queries and updates. Can also support reversals with the commented REVERSE lines and getting the position of a node with the PARENT lines.

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

98a335, 138 lines

```
typedef ll T; typedef ll L; // T: data type, L: lazy type
const T tneut = 0; const L lneut = 0; // neutrals
T f(T a, T b) { return a + b; } // associative
T apply(T v, L l, ll len) { // new st according to lazy
  return v + l * len; }
L comb(L a, L b) { return a + b; } // cumulative effect of lazy

struct Node {
  Node *l = 0, *r = 0;
  // Node *p = 0; // PARENT
  T val, acc; L lazy = lneut;
  ll y, c = 1;
  // bool rev = false; // REVERSE
  Node(T val = tneut) : val(val), acc(val), y(rand()) {}
  void recalc() {
    c = 1, acc = tneut;
    if (l) l->push(), acc = f(acc, l->acc), c += l->c;
    acc = f(acc, val);
    if (r) r->push(), acc = f(acc, r->acc), c += r->c;
    // if (l) l->p = this; // PARENT
    // if (r) r->p = this;
  }
  void push() {
    // if (rev) { // REVERSE
    //  swap(l, r), rev = false;
    //  if (l) l->rev ^= 1; if (r) r->rev ^= 1;
    // }
    val = apply(val, lazy, 1), acc = apply(acc, lazy, c);
    if (l) l->lazy = comb(l->lazy, lazy);
    if (r) r->lazy = comb(r->lazy, lazy);
    lazy = lneut;
  }
  // void pullAll() { // PARENT
  //  if (p) p->pullAll();
  //  push();
  // }

  Node* split(ll k) {
    assert(k > 0);
    if (k >= c) return 0;
    push();
    ll cnt = l ? l->c : 0;
    if (k <= cnt) { // "k <= val" for lower_bound(k)
      Node* nl = l->split(k),* ret = l;
      l = nl;
      recalc();
      swap(*this, *ret);
      return ret;
    } else if (k == cnt + 1) { // k == val
      Node* ret = r;
      r = 0;
      recalc();
      return ret;
    } else {
      Node* ret = r->split(k - cnt - 1); // and just "k"
      recalc(), ret->recalc();
      return ret;
    }
  }
  void merge(Node* ri) {
    if (!ri) return;
    push(), ri->push();
    if (y > ri->y) {
      if (r) r->merge(ri);
      else r = ri;
    } else {
      merge(ri->l);
      ri->l = ri;
      swap(*this, *ri);
    }
    recalc();
  }
  // ll pos() { // In which position I am // PARENT
  //  pullAll();
  //  ll ans = l ? l->c : 0;
  //  if (!p) return ans;
  //  if (p->r == this) return ans + p->pos() + 1;
  //  else return p->pos() + 1 - (r ? r->c : 0);
  // }
  T query() { // Query full range
    push();
    return acc;
  }
  void upd(L v) { lazy = comb(lazy, v); } // Update full range
  // void reverse() { rev = !rev; } // REVERSE
};
<<<<<<< HEAD
=======

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 [l,r] = split(t, pos);
  return merge(merge(l, n), r);
}

// 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));
}
>>>>>>> kactl/main
```

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

669a70, 22 lines

```
struct FT {
  vi s;
  FT(ll n) : s(n) {}
  void upd(ll pos, ll dif) { // a[pos] += dif
    for (; pos < SZ(s); pos |= pos + 1) s[pos] += dif;
  }
  ll query(ll pos) { // sum of values in [0, pos)
    ll res = 0;
    for (; pos > 0; pos &= pos - 1) res += s[pos-1];
    return res;
  }
  ll 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;
    ll pos = 0;
    for (ll 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 fakeUpd() before init()).

Time: $\mathcal{O}(\log^2 N)$. (Use persistent segment trees for $\mathcal{O}(\log N)$.)

"FenwickTree.h" 9abd20, 22 lines

```
struct FT2 {
  vector<vi> ys; vector<FT> ft;
  FT2(ll limx) : ys(limx) {}
  void fakeUpd(ll x, ll y) {
    for (; x < SZ(ys); x |= x + 1) ys[x].pb(y);
  }
  void init() {
    for (vi& v : ys) sort(ALL(v)), ft.pb(SZ(v));
  }
  ll ind(ll x, ll y) {
```

```
    return (ll)(lower_bound(ALL(ys[x]), y) - ys[x].begin()); }
  void upd(ll x, ll y, ll dif) {
    for (; x < SZ(ys); x |= x + 1)
      ft[x].upd(ind(x, y), dif);
  }
  ll query(ll x, ll 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: $\mathcal{O}(|V| \log |V| + Q)$ e0b6d1, 16 lines

```
template<class T>
struct RMQ {
  vector<vector<T>> jmp;
  RMQ(const vector<T>& V) : jmp(1, V) {
    for (ll pw = 1, k = 1; pw * 2 <= SZ(V); pw *= 2, ++k) {
      jmp.emplace_back(SZ(V) - pw * 2 + 1);
      fore(j,0,SZ(jmp[k]))
        jmp[k][j] = min(jmp[k - 1][j], jmp[k - 1][j + pw]);
    }
  }
  T query(ll a, ll b) {
    assert(a < b); // or return inf if a == b
    ll dep = 63 - __builtin_clzll(b - a);
    return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
  }
};
```

FastRMQ.h

Description: Like RMQ but with construction $O(|V|)$ and slower queries.
Time: $\mathcal{O}(|V| + Q)$ f4bfbf, 40 lines

```
template<class T>
struct RMQ {
  ll n;
  static constexpr T inf = 1e9; // change sign of inf for max
  vector<ll> mk;
  vector<T> bk, v;
  ll f(ll x) { return x >> 5; }
  RMQ(vector<T>& V) : n(SZ(V)), mk(n), bk(n, inf), v(V) {
    ll lst = 0;
    fore(i, 0, n) {
      bk[f(i)] = min(bk[f(i)], v[i]);
      while (lst && v[i-__builtin_ctzll(lst)]>v[i])// < for max
        lst ^= lst & -lst;
      mk[i] = ++lst, lst *= 2;
    }
    ll top = f(n);
    fore(k, 1, 64 - __builtin_clzll(top + 1)) {
      fore(i, 0, top - (1 << k) + 1)
        bk[top*k + i] =
          min(bk[top*(k-1) + i], bk[top*(k-1) + i + (1<<k-1)]);
    }
  }
  T get(ll st, ll en) {
    return v[en-64+__builtin_clzll(mk[en-1]&((1ll<<en-st)-1))];
  }
  T query(ll s, ll e) { // [s, e)
    ll b1 = f(s), b2 = f(e - 1);
    if (b1 == b2) return get(s, e);
    T ans = min(get(s, (b1 + 1)*32), get(b2*32, e));
    s = (b1 + 1)*32;
    e = b2*32;
    if (s < e) {
      ll k = 63 - __builtin_clzll(f(e - s));
      ll top = f(n) * k;
      ans = min(ans,
        min(bk[top + f(s)], bk[top + f(e - 1) - (1<<k) + 1]));
    }
    return ans;
  }
};
```

MoQueries.h

Description: Answer range queries offline using Hilbert order or SQRT decomposition. For Hilbert, change lines with `SQ` for commented lines. Define `add`, `rem`, `calc`, add necessary fields to `MoQueries`, instantiate, and call `solve`. Queries are given as pairs $[l, r)$. `end` is true if the element should be added/removed from the end of the range. `qid` is the query index. Starts at [0, 0).
Time: Fast $\mathcal{O}\left(TN\sqrt{Q} + CQ\right)$ where T is the cost of `add` and `rem` and C is the cost of `calc`. dad6b3, 35 lines

```
constexpr ll B = 447; // ~N/sqrt(Q)
struct MoQueries {
  typedef ll T;
  void add(ll pos, bool end, ll qid) {}
  void rem(ll pos, bool end, ll qid) {}
  T calc() { return 0; }
  ii k(ii &x) { return {x.fst/B, x.snd ^ -(x.fst/B&1)};} // SQ
  vector<T> solve(ll n, vector<ii> &qs) {
    ll l = 0, r = 0, q = SZ(qs); //, rx, ry, k, s;
    vi p(q); //, o(q);
    iota(ALL(p), 0);
  //fore(i, 0, q) {
  //  auto [x, y] = qs[i];
  //  for (k = s = bit_ceil((unsigned)n); s >>= 1;) {
  //    rx = (x&s)>0, ry = (y&s)>0, o[i] += s*s*((rx*3)^ry);
  //    if (!ry) {
  //      if (rx) x = k - 1 - x, y = k - 1 - y;
  //      swap(x, y);
  //    }
  //  }
  //}
  //sort(ALL(p), [&](ll i, ll j) { return o[i] < o[j]; });
    sort(ALL(p),[&](ll i,ll j){return k(qs[i])<k(qs[j]);});//SQ
    vector<T> res(q);
    for (ll i : p) {
      auto [ql, qr] = qs[i];
      while (l > ql) add(--l, 0, i);
      while (r < qr) add(r++, 1, i);
      while (l < ql) rem(l++, 0, i);
      while (r > qr) rem(--r, 1, i);
      res[i] = calc();
    }
    return res;
  }
};
```

MoTree.h

Description: Answer tree path queries by finding an approximate TSP through the queries, and moving from one query to the next by adding/removing points at the ends. If values are on tree edges, change `step` to add/remove the edge (a, c) and remove the initial `add` call (but keep `in`).
Time: $\mathcal{O}\left(N\sqrt{Q}\right)$ 437687, 31 lines

```
constexpr ll B = 447; // ~N/sqrt(Q)
struct MoTree {
  typedef ll T;
  void add(ll pos, ll end) {}
  void del(ll pos, ll end) {}
  T calc() { return 0; }
  vector<T> solve(vector<ii> Q, vector<vi>& g, ll root=0) {
    ll N = SZ(g), pos[2] = {};
    vi s(SZ(Q)), I(N), L(N), R(N), in(N), par(N);
    vector<T> res(SZ(Q));
    add(0, 0), in[0] = 1;
    auto dfs = [&](ll x, ll p, ll dep, auto&& f) -> void {
      par[x] = p, L[x] = dep ? I[x] = N++ : N;
      for (ll y : g[x]) if (y != p) f(y, x, !dep, f);
      R[x] = !dep ? I[x] = ++N : N;
    };
    dfs(root, -1, 0, dfs);
    #define K(x) ii(I[x.fst]/B, I[x.snd] ^ -(I[x.fst]/B & 1))
    iota(ALL(s), 0);
    sort(ALL(s), [&](ll s, ll t) {return K(Q[s]) < K(Q[t]);});
    for (ll qi : s) fore(e,0,2) {
      ll &a = pos[e], b = e ? Q[qi].fst : Q[qi].snd, i = 0;
#define step(c)in[c]?(del(a,e),in[a]=0):(add(c,e),in[c]=1),a=c;
      while (L[a] < L[b] || R[b] < R[a]) b = par[I[i++] = b];
      while (a != b) step(par[a]);
      while (i--) step(I[i]);
      if (e) res[qi] = calc();
    }
    return res;
  }
};
```

Numerical (4)

4.1 Polynomials and recurrences

Polynomial.h

Description: Some operations with polynomials. Everything except `div` works also with integers. For a faster version of `mul`, see FFT. 51f958, 48 lines

```
const double eps = 1e-9;
typedef vector<double> Poly;

double eval(const Poly& p, double x) { // O(n)
  double val = 0;
  for (ll i = SZ(p); i--;) (val *= x) += p[i];
  return val;
}
Poly derivate(const Poly& p) { // O(n)
  Poly res(SZ(p)-1);
  fore(i, 1, SZ(p)) res[i-1] = i*p[i];
  return res;
}
Poly add(const Poly& p, const Poly& q) { // O(n)
  Poly res(max(SZ(p), SZ(q)));
  fore(i,0,SZ(p)) res[i] += p[i];
  fore(i,0,SZ(q)) res[i] += q[i];
  while (!res.empty() && abs(res.back()) < eps) res.pop_back();
  return res;
}
Poly mul(const Poly& p, const Poly& q) { // O(n^2)
  if (p.empty() || q.empty()) return {};
  Poly res(SZ(p) + SZ(q) - 1);
  fore(i,0,SZ(p)) fore(j,0,SZ(q))
    res[i+j] += p[i] * q[j];
  return res;
}
pair<Poly, double> divSmall(const Poly& p, double x0) { // O(n)
  ll n = SZ(p)-1;     // Divide p by (x-x0), returns {res, rem}
  if (n < 0) return {{}, 0};
  if (n == 0) return {{}, p[0]};
```

```
  Poly res(n);
  res[n-1] = p[n];
  for (ll i = n - 1; i--;) res[i] = p[i+1] + x0 * res[i+1];
  return {res, p[0] + x0 * res[0]};
}
pair<Poly, Poly> div(Poly p, const Poly& q) { // O(n^2)
  if (SZ(p) < SZ(q)) return {{}, p};  // returns {res, rem}
  ll n = SZ(p) - SZ(q) + 1;
  Poly res(n);
  for (ll i = n; i--;) {
    res[i] = p.back() / q.back();
    fore(j, 0, SZ(q)) p[j + i] -= res[i] * q[j];
    p.pop_back();
  }
  while (!p.empty() && abs(p.back()) < eps) p.pop_back();
  return {res, p};
}
```

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" c5dc54, 20 lines

```
vector<double> polyRoots(Poly& p, double xmin, double xmax) {
  if (SZ(p) == 2) return {-p[0]/p[1]};
  vector<double> ret;
  Poly der = derivate(p), dr = polyRoots(der, xmin, xmax);
  dr.pb(xmin-1), dr.pb(xmax+1);
  sort(ALL(dr));
  fore(i,0,SZ(dr)-1) {
    double l = dr[i], h = dr[i+1];
    bool sign = eval(p, l) > 0;
    if (sign ^ (eval(p, h) > 0)) {
      fore(it, 0, 60) { // while (h - l > 1e-8)
        double m = (l + h) / 2, f = eval(p, m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.pb((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 $x[k] = c * \cos(k/(n-1) * \pi), k = 0 \dots n-1$.
Time: $\mathcal{O}\left(n^2\right)$

d72300, 13 lines

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

PolyInterpolatePuntual.h

Description: Given for $i \in \{0, 1, ..., n-1\}$ points $(i, y[i])$ computes the value of the n-1-degree polynomial that passes through them at point x.
Time: $\mathcal{O}\left(n \log mod\right)$

"../number-theory/ModularArithmetic.h" 0bd5d5, 15 lines

```
Mod interpolate(vector<Mod>& y, Mod x) {
  ll n = SZ(y);
  static vector<Mod> fi {1}; // Inverses of factorials
  while (SZ(fi) < n) fi.pb(fi.back() / SZ(fi));
  if (x.x < n) return y[x.x];
  Mod p = 1;
  fore(i, 0, n) p = p * (x - i);
  Mod ans = 0;
  fore(i, 0, n) {
    Mod t = y[i] * p / (x-i) * fi[i] * fi[n-1-i];
    if ((n-i) % 2 == 0) t = t * (mod - 1);
    ans = ans + t;
  }
  return ans;
}
```

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" 6620a1, 20 lines

```
vi berlekampMassey(vi s) {
  ll n = SZ(s), L = 0, m = 0;
  vi C(n), B(n), T;
  C[0] = B[0] = 1;

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

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

LinearRecurrence.h

Description: Generates the k'th term of an n-order linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$, given $S[0 \dots \geq n-1]$ and $tr[0 \dots n-1]$. Faster than matrix multiplication. Useful together with Berlekamp–Massey.
Usage: `linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number`
Time: $\mathcal{O}\left(n^2 \log k\right)$

a61838, 26 lines

```
typedef vi Poly;
ll linearRec(Poly S, Poly tr, ll k) {
  ll n = SZ(tr);

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

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

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

4.2 Optimization

GoldenSectionSearch.h

Description: Finds the argument minimizing the function f in the interval $[a, b]$ assuming f is unimodal on the interval, i.e. has only one local minimum and no local maximum. The maximum error in the result is eps. Works equally well for maximization with a small change in the code. See 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: $\mathcal{O}\left(\log((b-a)/\epsilon)\right)$

badd5e, 10 lines

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

HillClimbing.h

Description: Poor man's optimization for unimodal functions.

e848c6, 13 lines

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

pair<double, P> hillClimb(P start, auto&& f) {
  pair<double, P> cur(f(start), start);
  for (double jmp = 1e9; jmp > 1e-20; jmp /= 2) {
    fore(j,0,100) fore(dx,-1,2) fore(dy,-1,2) {
      P p = cur.snd;
      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.

1c4ce6, 6 lines

```
double quad(double a, double b, auto f, const ll n = 1000) {
  double h = (b - a) / 2 / n, v = f(a) + f(b);
  fore(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; });});});
```
592cae, 13 lines

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

d rec(auto& 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);
}
d quad(d a, d b, auto 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: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case.
76f7d9, 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 {
  ll 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)) {
    fore(i,0,m) fore(j,0,n) D[i][j] = A[i][j];
    fore(i,0,m) {B[i]=n+i; D[i][n]=-1; D[i][n+1]=b[i];}
    fore(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1, D[m+1][n] = 1;
  }

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

  bool simplex(ll phase) {
    ll x = m + phase - 1;
    for (;;) {
      ll s = -1;
      fore(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return true;
      ll r = -1;
      fore(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) {
    ll r = 0;
    fore(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;
      fore(i,0,m) if (B[i] == -1) {
        ll s = 0;
        fore(j,1,n+1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    fore(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
  }
};
```

4.3 Matrices

Determinant.h

Description: Calculates determinant of a matrix. Destroys the matrix.
Time: $\mathcal{O}(N^3)$
144e26, 15 lines

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

IntDeterminant.h

Description: Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.
Time: $\mathcal{O}(N^3)$
d39cf6, 18 lines

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

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.
Time: $\mathcal{O}(n^2 m)$
9863aa, 38 lines

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

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

  fore(i,0,n) {
    double v, bv = 0;
    fore(r,i,n) fore(c,i,m)
      if ((v = fabs(A[r][c])) > bv)
        br = r, bc = c, bv = v;
    if (bv <= eps) {
      fore(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]);
    fore(j,0,n) swap(A[j][i], A[j][bc]);
    bv = 1/A[i][i];
    fore(j,i+1,n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      fore(k,i+1,m) A[j][k] -= fac*A[i][k];
    }
    rank++;
  }

  x.assign(m, 0);
  for (ll i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    fore(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" 11ffdd, 7 lines

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

SolveLinearBinary.h

Description: Solves $Ax = b$ over $\mathbb{F}_2$. If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys A and b.
Time: $\mathcal{O}(n^2 m)$
54a024, 34 lines

```
typedef bitset<1000> bs;

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

  x = bs();
  for (ll i = rank; i--;) {
    if (!b[i]) continue;
    x[col[i]] = 1;
    fore(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 $A^{-1} = A^{-1}(2I - AA^{-1})$ (mod p^k) where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}(n^3)$

8a7b27, 35 lines

```
ll matInv(vector<vector<double>>& A) {
  ll n = SZ(A); vi col(n);
  vector<vector<double>> tmp(n, vector<double>(n));
  fore(i,0,n) tmp[i][i] = 1, col[i] = i;

  fore(i,0,n) {
    ll r = i, c = i;
    fore(j,i,n) fore(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]);
    fore(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];
    fore(j,i+1,n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      fore(k,i+1,n) A[j][k] -= f*A[i][k];
      fore(k,0,n) tmp[j][k] -= f*tmp[i][k];
    }
    fore(j,i+1,n) A[i][j] /= v;
    fore(j,0,n) tmp[i][j] /= v;
    A[i][i] = 1;
  }
  for (ll i = n-1; i > 0; --i) fore(j,0,i) {
    double v = A[j][i];
    fore(k,0,n) tmp[j][k] -= v*tmp[i][k];
  }

  fore(i,0,n) fore(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 $A^{-1} = A^{-1}(2I - AA^{-1})$ (mod p^k) where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}(n^3)$

"../number-theory/ModPow.h" a019e9, 37 lines

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

  fore(i,0,n) {
    ll r = i, c = i;
    fore(j,i,n) fore(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]);
    fore(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);
    fore(j,i+1,n) {
      ll f = A[j][i] * v % mod;
      A[j][i] = 0;
      fore(k,i+1,n) A[j][k] = (A[j][k] - f*A[i][k]) % mod;
      fore(k,0,n) tmp[j][k] = (tmp[j][k] - f*tmp[i][k]) % mod;
    }
    fore(j,i+1,n) A[i][j] = A[i][j] * v % mod;
    fore(j,0,n) tmp[i][j] = tmp[i][j] * v % mod;
    A[i][i] = 1;
  }

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

  fore(i,0,n) fore(j,0,n)
    A[col[i]][col[j]] = tmp[i][j] % mod + (tmp[i][j] < 0)*mod;
  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

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

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: $\mathcal{O}(N)$

3c76ca, 26 lines

```
typedef double T;
vector<T> tridiagonal(vector<T> diag, const vector<T>& super,
    const vector<T>& sub, vector<T> b) {
  ll n = SZ(b); vi tr(n);
  fore(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 (ll 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 $N = 2^{22}$)

71e979, 35 lines

```
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
  ll n = SZ(a), L = 63 - __builtin_clzll(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster if double)
  for (static ll k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    fore(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
  }
  vi rev(n);
  fore(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  fore(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (ll k = 1; k < n; k *= 2)
    for (ll i = 0; i < n; i += 2 * k) fore(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);
  ll L = 64 - __builtin_clzll(SZ(res)), n = 1 << L;
  vector<C> in(n), out(n);
  copy(ALL(a), begin(in));
  fore(i,0,SZ(b)) in[i].imag(b[i]);
  fft(in);
  for (C& x : in) x *= x;
  fore(i,0,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  fore(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" 8121b2, 21 lines

```
template<ll M> vi convMod(const vi &a, const vi &b) {
  if (a.empty() || b.empty()) return {};
  vi res(SZ(a) + SZ(b) - 1);
  ll B=64-__builtin_clzll(SZ(res)), n=1<<B, cut=ll(sqrt(M));
  vector<C> L(n), R(n), outs(n), outl(n);
  fore(i,0,SZ(a)) L[i] = C((ll)a[i] / cut, (ll)a[i] % cut);
  fore(i,0,SZ(b)) R[i] = C((ll)b[i] / cut, (ll)b[i] % cut);
  fft(L), fft(R);
  fore(i,0,n) {
    ll 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);
  fore(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" a249ae, 34 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.
void ntt(vi &a) {
  ll n = SZ(a), L = 63 - __builtin_clzll(n);
  static vi rt(2, 1);
  for (static ll k = 2, s = 2; k < n; k *= 2, s++) {
    rt.resize(n);
    ll z[] = {1, modpow(root, mod >> s)};
    fore(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1] % mod;
  }
  vi rev(n);
  fore(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  fore(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (ll k = 1; k < n; k *= 2)
    for (ll i = 0; i < n; i += 2 * k) fore(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);
    }
}
vi conv(const vi &a, const vi &b) {
  if (a.empty() || b.empty()) return {};
  ll s = SZ(a) + SZ(b) - 1, B = 64 - __builtin_clzll(s),
     n = 1 << B;
  ll inv = modpow(n, mod - 2);
  vi L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  fore(i,0,n)
    out[-i & (n - 1)] = (ll)L[i] * R[i] % mod * inv % mod;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
```

FastSubsetTransform.h

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

265ad3, 16 lines

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

NTT-operations.h

Description: Some operations on polynomials made fast with NTT. The may also work with doubles and FFT, but it's numerically unstable. `inv`, `log` ans `exp` return truncated infinite series. `Poly` elements should not have trailing zeros. The zero polynomial is {}.

"NumberTheoreticTransform.h", "../number-theory/FastInverse.h" b315c7, 102 lines

```
typedef vi Poly;

Poly add(const Poly& p, const Poly& q) { // O(n)
  Poly res(max(SZ(p), SZ(q)));
  fore(i, 0, SZ(p)) res[i] += p[i];
  fore(i, 0, SZ(q)) res[i] += q[i];
  for (ll& x : res) x %= mod;
  while (!res.empty() && !res.back()) res.pop_back();
  return res;
}
Poly derivate(const Poly& p) { // O(n)
  Poly res(max(0ll, SZ(p)-1));
  fore(i, 1, SZ(p)) res[i-1] = (i * p[i]) % mod;
  return res;
}
Poly integrate(const Poly& p) { // O(n)
  Poly ans(SZ(p) + 1);
  fore(i, 0, SZ(p)) ans[i+1] = (p[i] * inv(i+1)) % mod;
  return ans;
}
```

```
Poly takeMod(Poly p, ll n) { // O(n)
  p.resize(min(SZ(p), n));   // p % (x^n)
  while (!p.empty() && !p.back()) p.pop_back();
  return p;
}

Poly inv(const Poly& p, ll d) { // O(n log(n))
  Poly res = {inv(p[0])};       // first d terms of 1/p
  ll sz = 1;
  while (sz < d) {
    sz *= 2;
    Poly pre(p.begin(), p.begin() + min(SZ(p), sz));
    Poly cur = conv(res, pre);
    fore(i, 0, SZ(cur)) if (cur[i]) cur[i] = mod - cur[i];
    cur[0] = cur[0] + 2;
    res = takeMod(conv(res, cur), sz);
  }
  res.resize(d);
  return res;
}
Poly log(const Poly& p, ll d) { // O(n log(n))
  Poly cur = takeMod(p, d);     // first d terms of log(p)
  Poly res = integrate(
    takeMod(conv(inv(cur, d), derivate(cur)), d - 1));
  res.resize(d);
  return res;
}
Poly exp(const Poly& p, ll d) { // O(n log(n)^2)
  Poly res = {1};               // first d terms of exp(p)
  for (ll sz = 1; sz < d; ) {
    sz *= 2;
    Poly lg = log(res, sz), cur(sz);
    fore(i, 0, sz) cur[i] = (mod + (i<SZ(p) ? p[i] : 0)
      - (i<SZ(lg) ? lg[i] : 0)) % mod;
    cur[0] = (cur[0] + 1) % mod;
    res = takeMod(conv(res, cur), sz);
  }
  res.resize(d);
  return res;
}

pair<Poly,Poly> div(const Poly& a, const Poly& b) {
  ll m = SZ(a), n = SZ(b);   // O(n log(n)), returns {res, rem}
  if (m < n) return {{}, a}; // if min(m-n,n) < 750 it may be
  Poly ap = a, bp = b;       // faster to use cuadratic version
  reverse(ALL(ap)), reverse(ALL(bp));
  Poly q = conv(ap, inv(bp, m - n + 1));
  q.resize(SZ(q) + m - n - SZ(q) + 1, 0), reverse(ALL(q));
  Poly bq = conv(b, q);
  fore(i, 0, SZ(bq)) if (bq[i]) bq[i] = mod - bq[i];
  return {q, add(a, bq)};
}

vector<Poly> filltree(vi& x) {
  ll k = SZ(x);
  vector<Poly> tr(2*k);
  fore(i, k, 2*k) tr[i] = {(mod - x[i - k]) % mod, 1};
  for (ll i = k; --i;) tr[i] = conv(tr[2*i], tr[2*i+1]);
  return tr;
}
vi evaluate(Poly& a, vi& x) { // O(n log(n)^2)
  ll k = SZ(x);               // Evaluate a in all points of x
  if (!SZ(a)) return vi(k);
  vector<Poly> tr = filltree(x), ans(2*k);
  ans[1] = div(a, tr[1]).snd;
  fore(i, 2, 2*k) ans[i] = div(ans[i/2], tr[i]).snd;
  vi r(k);
  fore(i, 0, k) if (SZ(ans[i+k])) r[i] = ans[i+k][0];
```

```
  return r;
}
Poly interpolate(vi& x, vi& y) { // O(n log(n)^2)
  vector<Poly> tr = filltree(x);
  Poly p = derivate(tr[1]);
  ll k = SZ(y);
  vi d = evaluate(p, x); // pass tr here for a speed up
  vector<Poly> intr(2*k);
  fore(i, k, 2*k) intr[i] = {(y[i-k] * inv(d[i-k])) % mod};
  for (ll i = k; --i;) intr[i] = add(
    conv(tr[2*i], intr[2*i+1]), conv(tr[2*i+1], intr[2*i]));
  return intr[1];
}
```

Number theory (5)

5.1 Modular arithmetic

ModularArithmetic.h

Description: Operators for modular arithmetic. Update mod. Use commented code in invert if mod is not prime.

"euclid.h" 433a25, 20 lines

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

PrecomputeInverses.h

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

7f666d, 6 lines

```
constexpr ll mod = 1e9+7, LIM = 2e5;
array<ll, LIM> inv;
void initInv() {
  inv[1] = 1;
  fore(i,2,LIM) inv[i] = mod - mod / i * inv[mod % i] % mod;
}
```

FastInverse.h

Description: Fast modular inverse for a constant modulus.
Time: $\mathcal{O}(logn)$, $\approx$ 2.7x faster than euclid in CF.

4fccf0, 11 lines

```
constexpr ll mod = 1e9 + 7;
constexpr ll k = bit_width((unsigned long long)(mod - 2));
ll inv(ll a) {
  ll r = 1;
#pragma GCC unroll(k)
  fore(l, 0, k) {
    if ((mod - 2) >> l & 1) r = r * a % mod;
    a = a * a % mod;
  }
  return r;
}
```

ModPow.h

c6ee78, 8 lines

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

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

ModLog.h

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

0e2062, 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))
    fore(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) $= \sum_{i=0}^{to-1} (ki + c)\%m$. divsum is similar but for floored division.
Time: log(m), with a large constant.

5c5bc5, 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 $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.
Time: $\mathcal{O}(1)$ for modmul, $\mathcal{O}(\log b)$ for modpow

bbbd8f, 11 lines

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

ModSqrt.h

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

"ModPow.h" b167b9, 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;
  ll 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;
  }
}
```

5.2 Primality

Eratosthenes.h

Description: s[i] = smallest prime factor of i (except for i = 0, 1). sieve returns sorted primes less than L. fact returns sorted prime, exponent pairs of the factorization of n.

55dd05, 18 lines

```
const ll L = 1e6;
array<ll, L> s;
vi sieve() {
  vi p;
  for (ll i = 4; i < L; i += 2) s[i] = 2;
  for (ll i = 3; i * i < L; i += 2) if (!s[i])
    for (ll j=i*i; j < L; j += 2*i) if (!s[j]) s[j] = i;
  fore(i,2,L) if (!s[i]) p.pb(i), s[i] = i;
  return p;
}
vector<ii> fact(ll n) {
  vector<ii> res;
  for (; n > 1; n /= s[n]) {
    if (!SZ(res) || res.back().fst!=s[n]) res.pb({s[n],0});
    res.back().snd++;
  }
  return res;
}
```

FastEratosthenes.h

Description: Prime sieve for generating all primes smaller than LIM.
Time: $\mathcal{O}(n \log \log n)$; LIM=1e9 $\approx$ 1.5s

9d96d8, 20 lines

```
const ll LIM = 1e6;
bitset<LIM> isPrime;
vi eratosthenes() {
  const ll S = (ll)round(sqrt(LIM)), R = LIM / 2;
  vi pr = {2}, sieve(S+1); pr.reserve(ll(LIM/log(LIM)*1.1));
  vector<ii> cp;
  for (ll i = 3; i <= S; i += 2) if (!sieve[i]) {
    cp.pb({i, i * i / 2});
    for (ll j = i * i; j <= S; j += 2 * i) sieve[j] = 1;
  }
```

```
  for (ll L = 1; L <= R; L += S) {
    array<bool, S> block{};
    for (auto &[p, idx] : cp)
      for (ll i=idx; i < S+L; idx = (i+=p)) block[i-L] = 1;
    fore(i,0,min(S, R - L))
      if (!block[i]) pr.pb((L + i) * 2 + 1);
  }
  for (ll 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" 1d5cc3, 11 lines

```
bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
  ll s = __builtin_ctzll(n-1), d = n >> s;
  for (ull a : {2,325,9375,28178,450775,9780504,1795265022}) {
    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" fd4221, 18 lines

```
ull pollard(ull n) {
  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  auto f = [&](ull x) { return modmul(x, x, n) + i; };
  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;
}
```

FastDivisors.h

Description: Given the prime factorization of a number, returns all its divisors.
Time: $\mathcal{O}(d)$, where d is the number of divisors.

3c6074, 8 lines

```
vi divisors(vector<ii>& f) {
  vi res = {1};
  for (auto& [p, k] : f) {
    ll sz = SZ(res);
    fore(i,0,sz) for(ll j=0,x=p;j<k;j++,x*=p) res.pb(res[i]*x);
  }
  return res;
}
```

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 $a \pmod b$.

33ba8f, 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;
}
```

Diophantine.h

Description: Returns (x_0, y_0, dx, dy) such that all integer solutions (x, y) to $ax + by = r$ are $(x_0 + k \cdot dx, y_0 + k \cdot dy)$ for integer k.
Time: $\mathcal{O}(\log(\min(a, b)))$

"./euclid.h" 12347a, 6 lines

```
array<ll, 4> diophantine(ll a, ll b, ll r) {
  ll x, y, g = euclid(a, b, x, y);
  a /= g, b /= g, r /= g, x *= r, y *= r;
  assert(a * x + b * y == r); // otherwise no solution
  return {x, y, -b, a};
}
```

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" 04d93a, 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

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

phiFunction.h

Description: *Euler's* ϕ function is defined as $\phi(n) :=$ # of positive integers $\le n$ that are coprime with n. $\phi(1) = 1$, p prime $\Rightarrow \phi(p^k) = (p-1)p^{k-1}$, m, n coprime $\Rightarrow \phi(mn) = \phi(m)\phi(n)$. If $n = p_1^{k_1} p_2^{k_2} ... p_r^{k_r}$ then $\phi(n) = (p_1-1)p_1^{k_1-1}...(p_r-1)p_r^{k_r-1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$.
$\sum_{d|n} \phi(d) = n$, $\sum_{1 \le k \le n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$
Euler's: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod n, a^m \equiv a^{m \bmod \phi(n)} \pmod n$.
Generalization: $m \ge \log_2(n) \Rightarrow x^m \equiv x^{\phi(n)+(m \bmod \phi(n))} \pmod n$.
Fermat's little thm: p prime $\Rightarrow \forall a : a^{p-1} \equiv 1 \pmod p$.

151aa0, 8 lines

```
const ll LIM = 5000000;
array<ll, LIM> phi;

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

PointsUnderLine.h

Description: Given $a, b > 0$, `f` returns the number of lattice points (x, y) such that $ax + by \le c$ and $x, y > 0$, and `g` returns the number of lattice points (x, y) such that $ax + by \le c$, $0 < x \le X$ and $0 < y \le Y$

388aa4, 11 lines

```
ll f(ll a, ll b, ll c) {
  if (c <= 0) return 0;
  if (a < b) swap(a, b);
  ll m = c / a, k = (a - 1) / b, h = (c - a * m) / b;
  if (a == b) return m * (m - 1) / 2;
  return f(b, a - b*k, c - b*(k*m + h)) + k*m*(m - 1)/2 + m*h;
}
ll g(ll a, ll b, ll c, ll X, ll Y) {
  if (a * X + b * Y <= c) return X * Y;
  return f(a,b,c)-f(a,b,c-a*X)-f(a,b,c-b*Y)+f(a,b,c-a*X-b*Y);
}
```

5.4 Fractions

ContinuedFractions.h

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

5f8089, 21 lines

```
// double is safe for N ~ 1e7, use long double for N ~ 1e9
typedef long double ld;
ii approximate(ld x, ll N) {
  ll LP = 0, LQ = 1, P = 1, Q = 0, inf = LLONG_MAX; ld 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 - (ld)NP/(ld)NQ) < abs(x - (ld)P/(ld)Q)) ?
        ii{NP, NQ} : ii{P, Q};
    }
    if (abs(y = 1/(y - (ld)a)) > 3*N) {
      return {NP, NQ};
    }
    LP = P, P = NP, LQ = Q, Q = NQ;
  }
}
```

FracBinarySearch.h

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

79308c, 20 lines

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

Frac fracBS(auto&& 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 (ll 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 Highly composite numbers

The number of divisors of n is $O(\log(\log(n)))$. Max number of divisors up to 10^n:

n	0	1	2	3	4	5	6	7
divisors	1	4	12	32	64	128	240	448
number	1	6	60	840	7560	83160	720720	8648640

8	9	10	11	12
768	1344	2304	4032	6720
73513440	735134400	6983776800	97772875200	963761198400

13	14	15
10752	17280	26880
9316358251200	97821761637600	866421317361600

16	17	18
41472	64512	103680
8086598962041600	74801040398884800	897612484786617600

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

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.h

Description: Computes the Mobius function $\mu(n)$ for all $n < L$.
Time: $\mathcal{O}(L \log L)$ 50cb20, 6 lines

```
const ll L = 1e6;
array<int8_t, L> mu;
void calculateMu() {
  mu[1] = 1;
  fore(i,1,L) if(mu[i]) for(ll j=2*i; j<L; j+=i) mu[j]-=mu[i];
}
```

Mobius Inversion:

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

Other useful formulas/forms:

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

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

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

Combinatorial (6)

6.1 Permutations

6.1.1 Factorial

Legendre's formula: the exponent of p in the factorization of $n!$ is

$$\nu_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor = \frac{n - s_p(n)}{p - 1}$$

where $s_p(n)$ is the sum of the digits of n in base p.

Wilson's: $n > 1$ is prime iff $(n-1)! \equiv -1 \pmod{n}$.

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

n	1	2	3	4	5	6	7	8	9	10
$n!$	1	2	6	24	120	720	5040	40320	362880	3628800

n	11	12	13	14	15	16	17
$n!$	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14

n	20	25	30	40	50	100	150	171
$n!$	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.
Time: $\mathcal{O}(n)$ d7d731, 7 lines

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

6.1.2 Cycles

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

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

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

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

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 Lucas' Theorem

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

6.2.3 Binomials

Kummer's theorem: the exponent of p in $\binom{n}{k}$ is the number of carries when adding k and $n-k$ in base p, or, equivalently

$$\nu_p\left(\binom{n}{k}\right) = \frac{s_p(k)+s_p(n-k)-s_p(n)}{p-1}$$

where $s_p(n)$ is the sum of the digits of n in base p. Therefore $\binom{n}{k}$ is odd iff k is a submask of n.

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} = \prod_{i=1}^{k}\frac{n+1-i}{i} = \binom{n-1}{k-1}+\binom{n-1}{k}$$

$$\sum_{i=0}^{k}\binom{n+i}{i} = \binom{n+k+1}{k}, \quad \sum_{i=0}^{n}\binom{i}{k} = \binom{n+1}{k+1}$$

$$\sum_{k=0}^{n}\binom{n}{k}^2 = \binom{2n}{n}, \quad \sum_{k=0}^{n}k\binom{n}{k} = n2^{n-1}$$

multinomial.h

Description: Computes $\binom{k_1+\cdots+k_n}{k_1,k_2,\ldots,k_n} = \frac{(\sum k_i)!}{k_1!k_2!\ldots k_n!}$. 91eef8, 5 lines

```
ll multinomial(vi& v) {
  ll c = 1, m = v.empty() ? 1 : v[0];
  fore(i,1,SZ(v)) fore(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 Fibonacci numbers

$$F_{2n+1} = F_{n+1}^2 + F_n^2, \quad F_{2n} = F_{n+1}^2 - F_{n-1}^2, \quad \sum_{i=1}^{n} F_i = F_{n+2} - 1$$

$$F_{n+i}F_{n+j} - F_nF_{n+i+j} = (-1)^nF_iF_j$$

6.3.7 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: $(n-2)!/((d_1-1)!\cdots(d_n-1)!)$

6.3.8 Catalan numbers

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$
$$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2}C_n,\ C_{n+1} = \sum C_iC_{n-i}$$
$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

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

- 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.

6.4 Game theory

Misere nim: Second player wins iff (some heap size is greater than 1 and xor is 0) or (all heap sizes are 1 and n is odd).

Graph (7)

7.1 Fundamentals

BellmanFord.h

Description: Calculates shortest paths from s in a graph that might have negative edge weights. Stores the answer in nodes. Unreachable nodes get dist = inf; nodes reachable through negative-weight cycles get dist = -inf. Assumes $V^2 \max|w_i| < \sim 2^{63}$.
Time: $\mathcal{O}(VE)$ 387d3f, 17 lines

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

void bellmanFord(vector<Node>& nodes, vector<Ed>& eds, ll s) {
  nodes[s].dist = 0;
  sort(ALL(eds), [](Ed a, Ed b) { return a.s() < b.s(); });
  ll lim = SZ(nodes) / 2 + 2; // /3+100 with shuffled vertices
  fore(i,0,lim) for (Ed ed : eds) {
    Node cur = nodes[ed.a], &dest = nodes[ed.b];
    if (abs(cur.dist) == inf) continue;
    ll d = cur.dist + ed.w;
    if (d < dest.dist) dest = {i < lim-1 ? d : -inf, ed.a};
  }
  fore(i,0,lim) for (Ed e : eds)
    if (nodes[e.a].dist == -inf) nodes[e.b].dist = -inf;
}
```

FloydWarshall.h

Description: Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix m, where $m[i][j] =$ inf if i and j are not adjacent. As output, $m[i][j]$ is set to the shortest distance between i and j, inf if no path, or -inf if the path goes through a negative-weight cycle.
Time: $\mathcal{O}(N^3)$ 7eb90d, 12 lines

```
const ll inf = 1LL << 62;
void floydWarshall(vector<vi>& m) {
  ll n = SZ(m);
  fore(i,0,n) m[i][i] = min(m[i][i], 0LL);
  fore(k,0,n) fore(i,0,n) fore(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) {
      auto newDist = max(m[i][k] + m[k][j], -inf);
      m[i][j] = min(m[i][j], newDist);
    }
  fore(k,0,n) if (m[k][k] < 0) fore(i,0,n) fore(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

TopoSort.h

Description: Topological sorting. Given is an oriented graph. Output is an ordering of vertices, such that there are edges only from left to right. If there are cycles, the returned list will have size smaller than n – nodes reachable from cycles will not be returned.
Time: $\mathcal{O}(|V| + |E|)$

d543aa, 8 lines

```
vi topoSort(const vector<vi>& g) {
  vi d(SZ(g)), q;
  for (auto& li : g) for (ll x : li) d[x]++;
  fore(i,0,SZ(g)) if (!d[i]) q.pb(i);
  for (ll j = 0; j < SZ(q); j++) for (ll x : g[q[j]])
    if (!--d[x]) q.pb(x);
  return q;
}
```

DynamicConnectivity.h

Description: Offline disjoint-set data structure with remove of arbitrary edges. Uses UnionFindRollbackStore so it also supports queries of global value of RSUF. First use add, remove and query to make operations and then call process to get the answers of the queries in the variable ans. Does not support multiple edges.
Time: $\mathcal{O}\left(Q \log^2 N\right)$

"../data-structures/UnionFindRollbackStore.h" c8407a, 51 lines

```
enum { ADD, DEL, QUERY };
struct Query { ll type, x, y; }; // You can add stuff for QUERY
struct DynCon {
  vector<Query> q;
  RSUF uf;
  vi mt;
  map<ii, ll> last;
  vector<T> ans;
  DynCon(ll n) : uf(n) {}
  DynCon(vector<D>& d) : uf(d) {}
  void add(ll x, ll y) {
    if (x > y) swap(x, y);
    mt.pb(-1);
    last[{x, y}] = SZ(q);
    q.pb({ADD, x, y});
  }
  void remove(ll x, ll y) {
    if (x > y) swap(x, y);
    ll pr = last[{x, y}];
    mt[pr] = SZ(q);
    mt.pb(pr);
    q.pb({DEL, x, y});
  }
  void query() { // Add parameters if needed
    q.pb({QUERY, -1, -1});
    mt.pb(-1);
  }
  void process() { // Answers all queries in order
    if (q.empty()) return;
    fore(i, 0, SZ(q))
      if (q[i].type == ADD && mt[i] < 0) mt[i] = SZ(q);
    go(0, SZ(q));
  }
  void go(ll s, ll e) {
    if (s + 1 == e) {
      if (q[s].type == QUERY) { // Answer query using DSU
        ans.pb(uf.ans); // Maybe you want to use uf.get(x)
      }                 // for some x stored in Query
      return;
    }
    ll k = uf.time(), m = (s + e) / 2;
    for (ll i = e; --i >= m;)
      if (0 <= mt[i] && mt[i] < s) uf.join(q[i].x, q[i].y);
    go(s, m);
    uf.rollback(k);
    for (ll i = m; --i >= s;)
      if (mt[i] >= e) uf.join(q[i].x, q[i].y);
    go(m, e);
    uf.rollback(k);
  }
};
```

7.2 Network flow

PushRelabel.h

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

da4f8b, 48 lines

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

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

  void addFlow(Edge& e, ll f) {
    Edge &back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].pb(e.dest);
    e.f += f; e.c -= f; ec[e.dest] += f;
    back.f -= f; back.c += f; ec[back.dest] -= f;
  }
  ll calc(ll s, ll t) {
    ll v = SZ(g); H[s] = v; ec[t] = 1;
    vi co(2*v); co[0] = v-1;
    fore(i,0,v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);

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

MinCostMaxFlow.h

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

735f9d, 80 lines

```
#include "ext/pb_ds/priority_queue.hpp"

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

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

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

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

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

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

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

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

      totflow += fl;
      for (edge* x = par[t]; x; x = par[x->from]) {
        x->flow += fl;
        ed[x->to][x->rev].flow -= fl;
      }
    }
    fore(i,0,N) for (edge& e : ed[i])
      totcost += e.cost * e.flow;
    return {totflow, totcost/2};
  }
```

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

EdmondsKarp.h

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

1e8088, 36 lines

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

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

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

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

Dinic.h

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

d44dbc, 42 lines

```
struct Dinic {
  struct Edge {
    ll 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(ll n) : lvl(n), ptr(n), q(n), adj(n) {}
  void addEdge(ll a, ll b, ll c, ll rcap = 0) {
    adj[a].pb({b, SZ(adj[b]), c, c});
    adj[b].pb({a, SZ(adj[a]) - 1, rcap, rcap});
  }
  ll dfs(ll v, ll t, ll f) {
    if (v == t || !f) return f;
    for (ll& 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(ll s, ll t) {
    ll flow = 0; q[0] = s;
    fore(L,0,31) do { // 'll L=30' maybe faster for random data
      lvl = ptr = vi(SZ(q));
      ll qi = 0, qe = lvl[s] = 1;
      while (qi < qe && !lvl[t]) {
        ll 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(ll a) { return lvl[a] != 0; }
};
```

MinCut.h

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

GlobalMinCut.h

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

16cb60, 21 lines

```
pair<ll, vi> globalMinCut(vector<vi> mat) {
  pair<ll, vi> best = {LLONG_MAX, {}};
  ll n = SZ(mat);
  vector<vi> co(n);
  fore(i,0,n) co[i] = {i};
  fore(ph,1,n) {
    vi w = mat[0];
    size_t s = 0, t = 0;
    fore(it,0,n-ph) { // O(V^2) -> O(E log V) with prio. queue
      w[t] = LLONG_MIN;
      s = t, t = max_element(ALL(w)) - w.begin();
      fore(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]));
    fore(i,0,n) mat[s][i] += mat[t][i];
    fore(i,0,n) mat[i][s] = mat[s][i];
    mat[0][t] = LLONG_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" a93d73, 13 lines

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

7.3 Matching

hopcroftKarp.h

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

2bbb99, 42 lines

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

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

```
      res += dfs(a, 0, g, btoa, A, B);
  }
}
```

DFSMatching.h

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

6a75ec, 22 lines

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

MinimumVertexCover.h

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

"DFSMatching.h" cd3f06, 20 lines

```
vi cover(vector<vi>& g, ll n, ll m) {
  vi match(m, -1);
  ll res = dfsMatching(g, match);
  vector<bool> lfound(n, true), seen(m);
  for (ll it : match) if (it != -1) lfound[it] = false;
  vi q, cover;
  fore(i,0,n) if (lfound[i]) q.pb(i);
  while (!q.empty()) {
    ll i = q.back(); q.pop_back();
    lfound[i] = 1;
    for (ll e : g[i]) if (!seen[e] && match[e] != -1) {
      seen[e] = true;
      q.pb(match[e]);
    }
  }
  fore(i,0,n) if (!lfound[i]) cover.pb(i);
  fore(i,0,m) if (seen[i]) cover.pb(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. Requires $N \leq M$.
Time: $\mathcal{O}\left(N^2M\right)$

178f97, 31 lines

```
pair<ll, vi> hungarian(const vector<vi> &a) {
  if (a.empty()) return {0, {}};
  ll n = SZ(a) + 1, m = SZ(a[0]) + 1;
  vi u(n), v(m), p(m), ans(n - 1);
  fore(i,1,n) {
    p[0] = i;
    ll j0 = 0; // add "dummy" worker 0
    vi dist(m, LLONG_MAX), pre(m, -1);
    vector<bool> done(m + 1);
    do { // dijkstra
      done[j0] = true;
      ll i0 = p[j0], j1, delta = LLONG_MAX;
      fore(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;
      }
      fore(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
      ll j1 = pre[j0];
      p[j0] = p[j1], j0 = j1;
    }
  }
  fore(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" d15d4f, 40 lines

```
vector<ii> generalMatching(ll N, vector<ii>& ed) {
  vector<vi> mat(N, vi(N)), A;
  for (auto [a, b] : ed) {
    ll r = rand() % mod;
    mat[a][b] = r, mat[b][a] = (mod - r) % mod;
  }

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

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

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

7.4 DFS algorithms

SCC.h

Description: Finds strongly connected components in a directed graph. If vertices u, v belong to the same component, we can reach u from v and vice versa. Returns the number of components and the component of each vertex. Natural order of components is reverse topological order (a component only has edges to components with lower index).
Time: $\mathcal{O}(E+V)$

d2c984, 17 lines

```
pair<ll, vi> scc(vector<vi>& g) {
  ll n = SZ(g), Time = 0, ncomps = 0;
  vi val(n), comp(n, -1), z;
  function<ll(ll)> dfs = [&](ll j) {
    ll low = val[j] = ++Time, x; z.pb(j);
    for (ll e : g[j]) if (comp[e] < 0)
      low = min(low, val[e] ?: dfs(e));
    if (low == val[j]) {
      do comp[x = z.back()] = ncomps, z.pop_back();
      while (x != j);
      ncomps++;
    }
    return val[j] = low;
  };
  fore(i, 0, n) if (comp[i] < 0) dfs(i);
  return {ncomps, comp};
}
```

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

6def45, 43 lines

```
auto BCC(ll n, const vector<ii>& edges) {
  ll Time = 0, eid = 0;
  vi num(n), st;
  vector<vector<ii>> adj(n);
  for (auto [a, b] : edges)
    adj[a].pb({b, eid}), adj[b].pb({a, eid++});

  ll nComps = 0; // number of biconnected components
  vi edgesComp(eid, -1); // comp of each edge or -1 if bridge
  vector<set<ll>> nodesComp(n); // comps of each node

  function<ll(ll, ll)> dfs = [&](ll at, ll par) {
    ll me = num[at] = ++Time, top = me;
    for (auto [y, e] : adj[at]) if (e != par) {
      if (y == at) { // self loop
        nodesComp[at].insert(edgesComp[e] = nComps++);
      } else if (num[y]) {
        top = min(top, num[y]);
        if (num[y] < me) st.pb(e);
      } else {
        ll si = SZ(st), up = dfs(y, e);
        top = min(top, up);
        if (up == me) {
          st.pb(e); // from si to SZ(st) we have a comp
          fore(i, si, SZ(st)) {
```

```
          edgesComp[st[i]] = nComps;
          auto [u, v] = edges[st[i]];
          nodesComp[u].insert(nComps);
          nodesComp[v].insert(nComps);
        }
        nComps++, st.resize(si);
      } else if (up < me) st.pb(e); // else e is bridge
    }
  }
  return top;
};

fore(u, 0, n) if (!num[u]) dfs(u, -1);
fore(u, 0, n) if (nodesComp[u].empty())
  nodesComp[u].insert(nComps++);

return tuple(nComps, edgesComp, nodesComp);
};
```

2sat.h

Description: Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type $(a||b)\&\&(!a||c)\&\&(d||!b)\&\&\ldots$ becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions (~x).

Usage: `TwoSat ts(number of boolean variables);`
`ts.either(0, ~3); // Var 0 is true or var 3 is false`
`ts.setValue(2); // Var 2 is true`
`ts.atMostOne({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true`
`ts.solve(); // Returns true iff it is solvable`
`ts.values[0..N-1] holds the assigned values to the vars`

Time: $\mathcal{O}(N+E)$, where N is the number of boolean variables, and E is the number of clauses.

0c5d9b, 50 lines

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

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

  ll addVar() { // (optional)
    gr.pb({}), gr.pb({});
    return N++;
  }

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

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

  vi val, comp, z; ll time = 0;
  ll dfs(ll i) {
    ll low = val[i] = ++time, x; z.pb(i);
    for (ll e : gr[i]) if (!comp[e])
      low = min(low, val[e] ?: dfs(e));
    if (low == val[i]) do {
      comp[x = z.back()] = low, z.pop_back();
      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;
    fore(i, 0, 2*N) if (!comp[i]) dfs(i);
    fore(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 .snd to s and ret.

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

092ee4, 13 lines

```
vi eulerWalk(vector<vector<ii>>& gr, ll nedges, ll src=0) {
  ll 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()) {
    ll x = s.back(), &it = its[x], end = SZ(gr[x]);
    if (it == end) { ret.pb(x), s.pop_back(); continue; }
    auto [y, e] = gr[x][it++];
    if (!eu[e]) D[x]--, D[y]++, eu[e] = 1, s.pb(y);
  }
  for (ll x : D) if (x < 0 || SZ(ret) != nedges+1) return {};
  return {ret.rbegin(), ret.rend()};
}
```

7.5 Coloring

EdgeColoring.h

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

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

6f96a9, 26 lines

```
vi edgeColoring(ll N, vector<ii> eds) {
  vi cc(N + 1), ret, fan(N), free(N), loc;
  for (auto [u, v] : eds) ++cc[u], ++cc[v];
  ll ncols = *max_element(ALL(cc)) + 1;
  vector<vi> adj(N, vi(ncols, -1));
  for (auto [u, v] : eds) {
    fan[0] = v, loc.assign(ncols, 0);
    ll 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 (ll 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) {
      ll 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 (ll y : {fan[0], u, end})
      for (ll& z = free[y] = 0; adj[y][z] != -1; z++);
  }
  for (auto [u, v] : eds)
    ret.pb(find(ALL(adj[u]), v) - adj[u].begin());
  return ret;
}
```

7.6 Heuristics

MaximalCliques.h

Description: Runs a callback for all maximal cliques in a graph (given as a symmetric bitset matrix; self-edges not allowed). Callback is given a bitset representing the maximal clique.

Time: $\mathcal{O}\left(3^{n/3}\right)$, much faster for sparse graphs

307f7f, 9 lines

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

89fbc5, 49 lines

```
typedef vector<bitset<200>> vb;
struct Maxclique {
  double limit=0.025, pk=0;
  struct Vertex { ll 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; });
    ll mxD = r[0].d;
    fore(i,0,SZ(r)) r[i].d = min(i, mxD) + 1;
  }
  void expand(vv& R, ll 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.pb(R.back().i);
      vv T;
      for (auto v : R) if (e[R.back().i][v.i]) T.pb({v.i});
      if (SZ(T)) {
        if (S[lev]++ / ++pk < limit) init(T);
        ll j = 0, mxk = 1, mnk = max(SZ(qmax) - SZ(q) + 1,1ll);
        C[1].clear(), C[2].clear();
        for (auto v : T) {
          ll k = 1;
          auto f = [&](ll 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].pb(v.i);
        }
        if (j > 0) T[j - 1].d = 0;
        fore(k,mnk,mxk + 1) for (ll 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) {
    fore(i,0,SZ(e)) V.pb({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.

7.7 Trees

BinaryLifting.h

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

90fce2, 22 lines

```
vector<vi> treeJump(vi& P){
  ll d = bit_width((unsigned)SZ(P));
  vector<vi> jmp(d, P);
  fore(i,1,d)fore(j,0,SZ(P)) jmp[i][j] = jmp[i-1][jmp[i-1][j]];
  return jmp;
}

ll jmp(vector<vi>& tbl, ll nod, ll steps) {
  fore(i,0,SZ(tbl)) if (steps & (1<<i)) nod = tbl[i][nod];
  return nod;
}

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

LCA.h

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

"../data-structures/RMQ.h" b66e8f, 21 lines

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

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

  ll lca(ll a, ll 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)];}
};
```

TreePathQueries.h

Description: Data structure for computing queries on paths in a tree. Works for nodes having values but can be changed to work with edges having values.
Time: construction $\mathcal{O}(N \log N)$, queries $\mathcal{O}(\log N)$

aeb7e6, 56 lines

```
struct PathQueries {
  typedef ll T;
  constexpr static T neut = LONG_LONG_MIN;
  T f(const T& a, const T& b) {
    return max(a, b);
  } // (any associative and commutative fn)

  ll n, K;
  vector<vi> anc;
  vector<vector<T>> part;
  vi depth;
  PathQueries(const vector<vi>& g, vector<T>& vals) // NODES
    : n(SZ(g)), K(64 - __builtin_clzll(n)), anc(K, vi(n, -1)),
      part(K, vector<T>(n, neut)), depth(n) {
    part[0] = vals;
//PathQueries(vector<vector<pair<ll, T>>> &g_) // EDGES
//  : n(SZ(g_)), K(64 - __builtin_clzll(n)), anc(K, vi(n, -1)),
//    part(K, vector<T>(n, neut)), depth(n) {
//  vector<vi> g(n);
//  fore(u, 0, n) for (auto [v, data] : g_[u]) {
//    g[u].pb(v);
//  }
    vi s = {0};
    while (!s.empty()) {
      ll u = s.back();
      s.pop_back();
      for (ll v : g[u]) if (v != anc[0][u]) {
        anc[0][v] = u, depth[v] = depth[u] + 1, s.pb(v);
      }
    }
//  fore(u, 0, n) for (auto [v, data] : g_[u]) { // EDGES
//    part[0][depth[u] > depth[v] ? u : v] = data;
//  }
    fore(k, 0, K - 1) fore(v, 0, n) {
      if (anc[k][v] != -1) {
        anc[k + 1][v] = anc[k][anc[k][v]];
        part[k + 1][v] = f(part[k][v], part[k][anc[k][v]]);
      }
    }
  }
  T query(ll u, ll v) {
    if (depth[u] < depth[v]) swap(u, v);
    T ans = neut;
    fore(k, 0, K) if ((depth[u] - depth[v]) & (1 << k))
      ans = f(ans, part[k][u]), u = anc[k][u];
    if (u == v) return f(ans, part[0][u]); // NODES
//  if (u == v) return ans; // EDGES
    for (ll k = K; k--;) if (anc[k][u] != anc[k][v]) {
      ans = f(ans, f(part[k][u], part[k][v]));
      u = anc[k][u], v = anc[k][v];
    }
    ans = f(ans, f(part[0][u], part[0][v]));
    return f(ans, part[0][anc[0][u]]); // NODES
//  return ans; // EDGES
  }
};
```

CompressTree.h

Description: Given a tree T rooted at 0, and a subset of nodes X returns tree with vertex set $\{\text{lca}(x, y) : x \in X, y \in X\}$ and edges between every pair of vertices in which one is an ancestor of the other in T. Size is at most $2|X| - 1$, including X. Returns a list of (par, orig_index) representing a tree rooted at 0. The root points to itself.
Time: $\mathcal{O}(|X| \log |X|)$

"LCA.h" d8a7c9, 12 lines

```
vector<ii> compressTree(LCA& lca, vi X) {
  static vi rev; rev.resize(SZ(lca.time));
  auto cmp = [&](ll a,ll b){ return lca.time[a]<lca.time[b]; };
  sort(ALL(X), cmp);
  fore(i,0,SZ(X)-1) X.pb(lca.lca(X[i], X[i+1]));
  sort(ALL(X), cmp), X.erase(unique(ALL(X)), X.end());
  fore(i,0,SZ(X)) rev[X[i]] = i;
  vector<ii> ret = {{{0, X[0]}}};
  fore(i,0,SZ(X)-1)
    ret.pb({rev[lca.lca(X[i], X[i+1])], X[i+1]});
  return ret;
}
```

HLD.h

Description: Decomposes a tree into vertex disjoint heavy paths and light edges such that the path from any leaf to the root contains at most log(n) light edges. Code does additive modifications and sum queries, but can support commutative segtree modifications/queries on paths and subtrees. For a non-commutative operation see LinkCutTree. Takes as input the full adjacency list. VALS_ED being true means that values are stored in the edges, as opposed to the nodes, and updates have to be done on child nodes. All values initialized to the segtree default. Root must be 0. If you only have point updates you can use normal segment tree instead of lazy.
Time: $\mathcal{O}((\log N)^2)$

"../data-structures/LazySegmentTree.h" 9855fc, 49 lines

```
template <bool VALS_ED> struct HLD {
  ll N, tim = 0;
  vector<vi> adj;
  vi par, siz, rt, pos;
  Tree t;
  HLD(vector<vi> adj_)
    : N(SZ(adj_)), adj(adj_), par(N, -1), siz(N, 1),
      rt(N), pos(N), t(N) { dfsSz(0), dfsHld(0); }
  void dfsSz(ll v) {
    if (par[v] != -1) adj[v].erase(find(ALL(adj[v]), par[v]));
    for (ll& u : adj[v]) {
      par[u] = v;
      dfsSz(u);
      siz[v] += siz[u];
      if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
    }
  }
  void dfsHld(ll v) {
    pos[v] = tim++;
    for (ll u : adj[v]) {
      rt[u] = (u == adj[v][0] ? rt[v] : u);
      dfsHld(u);
    }
  }
  void process(ll u, ll v, auto op) {
    for (; rt[u] != rt[v]; v = par[rt[v]]) {
      if (pos[rt[u]] > pos[rt[v]]) swap(u, v);
      op(pos[rt[v]], pos[v] + 1);
    }
    if (pos[u] > pos[v]) swap(u, v);
    op(pos[u] + VALS_ED, pos[v] + 1);
  }
  void updPath(ll u, ll v, L val) {
    process(u, v, [&](ll l, ll r) { t.upd(l, r, val); });
  }
```

```
  T queryPath(ll u, ll v) {
    T res = tneut;
    process(u, v, [&](ll l, ll r) {
        res = f(res, t.query(l, r));
    });
    return res;
  }
  T querySubtree(ll v) { // updSubtree is similar
    return t.query(pos[v] + VALS_ED, pos[v] + siz[v]);
  }
  // void updPoint(ll v, T val) { // For normal segment tree
  //  t.upd(pos[v], val); // queryPoint is similar
  // }
};
```

Kruskal.h

Description: Finds MST of weighted graph. Returns only cost by default.
Time: $\mathcal{O}(m)$

"../data-structures/UnionFind.h" eed941, 13 lines

```
ll kruskal(vector<pair<ll, ii>>& es, ll n) {
  sort(ALL(es));
  UF uf(n);
  ll r = 0;
  for (auto [w, p] : es) {
    auto [u, v] = p;
    if (uf.join(u, v)) {
      r += w;
      // {w, {u, v}} is in MST
    }
  }
  return r;
}
```

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" 6825cd, 60 lines

```
struct Edge { ll 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(ll n, ll 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<ll, ll, vector<Edge>>> cycs;
  fore(s,0,n) {
    ll 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;
        ll 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]}});
      }
    }
    fore(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;
  }
  fore(i,0,n) par[i] = in[i].a;
  return {res, par};
}
```

CentroidTree.h

Description: Calculate the centroid tree of a tree. Returns vector of fathers (-1 for root).
Time: $\mathcal{O}(N)$

f96917, 24 lines

```
vi centroidTree(vector<vi>& g) {
  ll n = SZ(g);
  vector<bool> vis(n, false);
  vi fat(n), szt(n);
  function<ll(ll, ll)> calcsz = [&](ll x, ll f) {
    szt[x] = 1;
    for (ll y : g[x]) if (y != f && !vis[y])
      szt[x] += calcsz(y, x);
    return szt[x];
  };
  function<void(ll, ll, ll)> dfs = [&](ll x, ll f, ll sz) {
    if (sz < 0) sz = calcsz(x, -1);
    for (ll y : g[x]) if (!vis[y] && szt[y] * 2 >= sz) {
      szt[x] = 0;
      dfs(y, f, sz);
      return;
    }
    vis[x] = true;
    fat[x] = f;
    for (ll y : g[x]) if (!vis[y]) dfs(y, x, -1);
  };
  dfs(0, -1, -1);
  return fat;
}
```

LinkCutTree.h

Description: Represents a forest of rooted trees with nodes **indexed from one**. You can add and remove edges (as long as the result is still a forest), make path queries, subtree queries, point updates and select a node as root of its subtree. If you don't have subtree queries you can remove the parts that say SUBTREE.
Time: All operations take amortized $\mathcal{O}(\log N)$.

f82ea6, 123 lines

```
typedef ll T;
const T neut = 0;
T f(T a, T b) { return a + b; } // associative
T neg(T a, T b) { return a - b; } // inverse of f for SUBTREE
struct SplayTree {
  struct Node {
    array<ll, 2> c = {0, 0};
    ll p = 0;
    array<T, 2> path = {neut, neut};
    T self = neut, sub = neut, vir = neut;
    bool reverse = false;
  };
  vector<Node> nods;

  SplayTree(ll n) : nods(n + 1) {}

  ll getDir(ll x) {
    ll p = nods[x].p;
    if (!p) return -1;
    if (nods[p].c[0] == x) return 0;
    return nods[p].c[1] == x ? 1 : -1;
  }
  void pushDown(ll x) {
    if (!x) return;
    if (nods[x].reverse) {
      auto [l, r] = nods[x].c;
      nods[l].reverse ^= 1, nods[r].reverse ^= 1;
      swap(nods[x].c[0], nods[x].c[1]);
      swap(nods[x].path[0], nods[x].path[1]);
      nods[x].reverse = false;
    }
  }
  void pushUp(ll x) {
    auto [l, r] = nods[x].c;
    pushDown(l), pushDown(r);
    nods[x].path[0] =
      f(f(nods[l].path[0], nods[x].self), nods[r].path[0]);
    nods[x].path[1] =
      f(f(nods[r].path[1], nods[x].self), nods[l].path[1]);
    nods[x].sub = f(
      f(f(nods[x].vir, nods[l].sub), nods[r].sub),
      nods[x].self);
  }
  void setChild(ll x, ll y, ll dir) {
    nods[x].c[dir] = y;
    nods[y].p = x;
    pushUp(x);
  }
  void rotate(ll x) {
    ll y = nods[x].p, dx = getDir(x);
    ll z = nods[y].p, dy = getDir(y);
    setChild(y, nods[x].c[!dx], dx), setChild(x, y, !dx);
    if (~dy) setChild(z, x, dy);
    nods[x].p = z;
  }
  void spa(ll x) { // splay
    for (pushDown(x); ~getDir(x); ) {
      ll y = nods[x].p, z = nods[y].p;
      pushDown(z), pushDown(y), pushDown(x);
      ll dx = getDir(x), dy = getDir(y);
      if (~dy) rotate(dx != dy ? x : y);
      rotate(x);
    }
  }
};

struct LinkCutTree : SplayTree {
  LinkCutTree(ll n) : SplayTree(n) {}
  ll access(ll x) {
```

```
    ll u = x, v = 0;
    for (; u; v = u, u = nods[u].p) {
      spa(u);
      ll& ov = nods[u].c[1];
      nods[u].vir = f(nods[u].vir, nods[ov].sub);
      nods[u].vir = neg(nods[u].vir, nods[v].sub); // SUBTREE
      ov = v; pushUp(u);
    }
    return spa(x), v;
  }

  void mkR(ll u) { // make root of its tree
    access(u);
    nods[u].reverse ^= 1;
    pushDown(u);
  }
  ll lca(ll u, ll v) { // least common ancestor, 0 if not conn
    if (u == v) return u;
    access(u);
    ll ret = access(v);
    return nods[u].p ? ret : 0;
  }
  bool connected(ll u, ll v) {  // are u and v in the same tree
    return lca(u, v) > 0;
  }
  T query(ll u) { // query single element
    return nods[u].self;
  }
  T querySub(ll u) { // query SUBTREE of u
    access(u);
    return f(nods[u].vir, nods[u].self);
  }
  void upd(ll u, T val) { // update value of u
    access(u);
    nods[u].self = val;
    pushUp(u);
  }
  // The following functions change the root
  void link(ll u, ll v) { // add edge between u and v
    mkR(u), access(v);
    nods[v].vir = f(nods[v].vir, nods[u].sub);
    nods[u].p = v;
    pushUp(v);
  }
  void cut(ll u, ll v) { // remove edge u v
    mkR(u), access(v);
    nods[v].c[0] = nods[u].p = 0;
    pushUp(v);
  }
  T queryPath(ll u, ll v) { // query on path from u to v
    mkR(u), access(v);
    return nods[v].path[1];
  }
};
```

LinkCutTree-PathUpdates.h

Description: Represents a forest of rooted trees with nodes **indexed from one**. You can add and remove edges (as long as the result is still a forest), make path queries and path updates and select a node as root of its subtree.
Time: All operations take amortized $\mathcal{O}(\log N)$.

e79eae, 133 lines

```
struct LinkCutTree {
  typedef ll T; typedef ll L; // T: data type, L: lazy type
  static constexpr T tneut = 0; static constexpr L lneut = 0;
  T f(T a, T b) { return a + b; } // associative & commutative
  T apply(T v, L l, ll len) { // new st according to lazy
    return v + l * len; }
  L comb(L a, L b) { return a + b; }//cumulative effect of lazy
  struct Node {
    ll s = 1; bool rev = 0;
    T val, t; L d = lneut;
    array<ll, 2> c = {0, 0}; ll p = 0;
    Node(T v = tneut) : val(v), t(v) {}
  };
  vector<Node> nods;
  LinkCutTree(ll n) : nods(n + 1) {
    nods[0].s = 0;
  }
  LinkCutTree(vector<T>& a) : nods(SZ(a) + 1) {
    fore(i, 0, SZ(a)) nods[i + 1] = Node(a[i]);
    nods[0].s = 0;
  }
  bool isRoot(ll u) {
    Node N = nods[nods[u].p];
    return N.c[0] != u && N.c[1] != u;
  }
  void push(ll u) {
    Node& N = nods[u];
    if (N.rev) {
      N.rev = 0, swap(N.c[0], N.c[1]);
      fore(x, 0, 2) if (N.c[x]) nods[N.c[x]].rev ^= 1;
    }
    N.val = apply(N.val, N.d, 1);
    N.t = apply(N.t, N.d, N.s);
    fore(x, 0, 2) if (N.c[x])
      nods[N.c[x]].d = comb(nods[N.c[x]].d, N.d);
    N.d = lneut;
  }
  T get(ll u) {
    return apply(nods[u].t, nods[u].d, nods[u].s);
  }
  void calc(ll u) {
    Node& N = nods[u];
    N.t = f(f(get(N.c[0]), apply(N.val, N.d, 1)), get(N.c[1]));
    N.s = 1 + nods[N.c[0]].s + nods[N.c[1]].s;
  }
  void conn(ll c, ll p, ll il) {
    if (c) nods[c].p = p;
    if (il >= 0) nods[p].c[!il] = c;
  }
  void rotate(ll u) {
    ll p = nods[u].p, g = nods[p].p;
    bool gCh = isRoot(p), isl = u == nods[p].c[0];
    conn(nods[u].c[isl], p, isl);
    conn(p, u, !isl);
    conn(u, g, gCh ? -1 : (p == nods[g].c[0]));
    calc(p);
  }
  void spa(ll u) { // splay
    while (!isRoot(u)) {
      ll p = nods[u].p, g = nods[p].p;
      if (!isRoot(p)) push(g);
      push(p), push(u);
      if (!isRoot(p))
        rotate((u==nods[p].c[0]) ^ (p==nods[g].c[0]) ? u : p);
      rotate(u);
    }
    push(u), calc(u);
  }
  ll lift_rec(ll u, ll t) {
    if (!u) return 0;
    Node N = nods[u];
    ll s = nods[N.c[0]].s;
    if (t == s) return spa(u), u;
    if (t < s) return lift_rec(N.c[0], t);
    return lift_rec(N.c[1], t - s - 1);
  }
  ll exv(ll u) { // expose
    ll last = 0;
    for (ll v = u; v; v = nods[v].p)
      spa(v), nods[v].c[0] = last, calc(v), last = v;
    spa(u);
    return last;
  }

  void mkR(ll u) { // make root of its tree
    exv(u), nods[u].rev ^= 1;
  }
  ll getR(ll u) { // get root of u tree
    exv(u);
    while (nods[u].c[1]) u = nods[u].c[1];
    spa(u);
    return u;
  }
  ll father(ll u) { // father of u, 0 if u is root
    exv(u), u = nods[u].c[1];
    while (nods[u].c[0]) u = nods[u].c[0];
    return u;
  }
  ll lift(ll u, ll t) {//t ancestor of x, lift(x,1) is x father
    exv(u);
    return lift_rec(u, t);
  }
  ll depth(ll u) { // distance from u to its tree root
    exv(u);
    return nods[u].s - 1;
  }
  ll lca(ll u, ll v) { // least common ancestor, 0 if not conn
    exv(u);
    return exv(v);
  }
  bool connected(ll u, ll v) { // are u and v in the same tree
    exv(u), exv(v);
    return u == v || nods[u].p != 0;
  }
  void cut(ll u) { // cuts u from father, becoming a root
    exv(father(u)), nods[u].p = 0;
  }
  // The following functions change the root
  void link(ll u, ll v) { // add edge between u and v
    mkR(u), nods[u].p = v;
  }
  void cut(ll u, ll v) { // remove edge u v
    mkR(u), cut(v);
  }
  T query(ll u, ll v) { // query on path from u to v
    mkR(u), exv(v);
    return get(v);
  }
  void upd(ll u, ll v, L d) { // modify path from u to v
    mkR(u), exv(v), nods[v].d = comb(nods[v].d, d);
  }
};
```

Reroot.h

Description: define `Data`, `finalize`, `acc` (as explained below), adding any necessary fields to `Reroot`. Then instantiate, add the data, and call `reroot()`.
`finalize` is applied before returning answer for `p` if `g[p][ei]` was its parent; usually identity function. When `p` is the root, `ei` is -1.
Let accumulated$(p, S) = a$ such that `finalize(a, ...)` is the answer for p if S were its only children. `acc` should, given accumulated(p, S) and the answer for `g[p][ei]`, compute accumulated$(p, S \cup \{\texttt{g[p][ei]}\})$ in `p_ans`. `neuts[p]` should be accumulated$(p, \varnothing)$.

root_dp[v] is the answer for the whole tree with v as root, fdp[v][ei] is the answer for g[v][ei] if v is the root and bdp[v][ei] is the answer for v if g[v][ei] is the root.
Equivalent to the following code, but for all roots:

```
Data dfs(ll u, ll p) {
 Data res = neuts[u];
 fore(ei, 0, SZ(g[u])) if (g[u][ei] != p)
  acc(res, dfs(g[u][ei], u), u, ei);
 ll pid = p == -1 ?  -1 :  find(ALL(g[u]), p) - begin(g[u]);
 return dp[u] = finalize(res, u, pid);
}
```

Time: Fast $\mathcal{O}(n \log n)$ assuming $\mathcal{O}(1)$ operations.

af6687, 47 lines

```
struct Data {};
typedef vector<Data> vd;

struct Reroot {
  ll n; vector<vi>& g; vd& neuts;

  Data finalize(const Data& a, ll p, ll ei) {
    return a;
  }
  void acc(Data& p_ans, const Data& child_ans, ll p, ll ei) {
    p_ans = Data{};
  }

  vd root_dp; vector<vd> fdp, bdp;
  Reroot(vector<vi>& g, vd& neuts) : n(SZ(g)), g(g),
      neuts(neuts), root_dp(n), fdp(n), bdp(n) {}
  void reroot() {
    if (n==1) { root_dp[0]=finalize(neuts[0], 0, -1); return; }
    vd dp = neuts, e(n); vi o = {0}, p(n); o.reserve(n);
    fore(i, 0, n) for (ll v : g[o[i]]) if (v != p[o[i]])
      p[v] = o[i], o.pb(v);
    for (ll u : views::reverse(o)) {
      ll pei = -1;
      fore(ei, 0, SZ(g[u]))
        if (g[u][ei] == p[u]) pei = ei;
        else acc(dp[u], dp[g[u][ei]], u, ei);
      dp[u] = finalize(dp[u], u, pei);
    }
    for (ll u : o) {
      dp[p[u]] = dp[u];
      fdp[u].reserve(SZ(g[u])), bdp[u].reserve(SZ(g[u]));
      for (ll v : g[u]) fdp[u].pb(dp[v]);
      ex(e, fdp[u], neuts[u], u);
      fore(i, 0, SZ(fdp[u])) bdp[u].pb(finalize(e[i], u, i));
      acc(e[0], fdp[u][0], u, 0);
      root_dp[u] = finalize(n > 1 ? e[0] : neuts[u], u, -1);
      fore(i, 0, SZ(g[u])) dp[g[u][i]] = bdp[u][i];
    }
  }
  void ex(vd& e, vd& a, Data& ne, ll v) {
    ll d = SZ(a); fill(begin(e), begin(e) + d, ne);
    for (ll b = bit_width((unsigned)d); b--;) {
      for (ll i = d; i--;) e[i] = e[i >> 1];
      fore(i, 0, d - (d & !b)) acc(e[(i >> b)^1], a[i], v, i);
    }
  }
};
```

RerootInv.h

Description: Make rerooting linear by defining the inverse of acc. Add unacc to the struct, keep acc and finalize, and change ex. Don't use inheritance.
unacc should, given accumulated$(p, g[p])$ and the answer for g[p][ei], compute accumulated$(p, g[p] \setminus \{g[p][ei]\})$ in ans.
Time: Fast $\mathcal{O}(n)$

"Reroot.h" f16d08, 11 lines

```
struct RerootInv : Reroot {
  void unacc(Data& ans, const Data& child_ans, ll p, ll ei) {
    ans = Data{};
  }
  void ex(vd& e, vd& a, Data& ne, ll v) {
    ll d = SZ(a); Data b = ne;
    fore(i, 0, d) acc(b, a[i], v, i);
    fill(begin(e), begin(e) + d, b);
    fore(i, 0, d) unacc(e[i], a[i], v, i);
  }
};
```

RerootLinear.h

Description: Use two operations instead of one to make rerooting linear. Usually only worth it for non- $\mathcal{O}(1)$ operations. Add merge and extend, and change acc and exclusive. Don't use inheritance.
merge should, given accumulated(p, S) and accumulated(p, T), with S and T disjoint, return accumulated$(p, S \cup T)$.
extend should, given the answer for g[p][ei], return b such that merge(neuts[p], b, p) = accumulated$(p, \{$g[p][ei]$\})$.
Time: Slow $\mathcal{O}(n)$

"Reroot.h" 4cb523, 17 lines

```
struct RerootLinear : Reroot {
  Data merge(const Data& a, const Data& b, ll p) {
    return Data{};
  }
  Data extend(const Data& a, ll p, ll ei) {
    return Data{};
  }
  void acc(Data& p_ans, const Data& child_ans, ll p, ll ei) {
    p_ans = merge(p_ans, extend(child_ans, p, ei), p);
  }
  void ex(vd& e, vd& a, Data& ne, ll v) {
    ll d = SZ(a); vd p(d + 1, ne), s(d + 1, ne);
    fore(i,0,d) p[i+1] = merge(p[i], a[i]=extend(a[i],v,i), v);
    for (ll i = d; i--;) s[i] = merge(a[i], s[i + 1], v);
    fore(i, 0, d) e[i] = merge(p[i], s[i + 1], v);
  }
};
```

7.8 Math

7.8.1 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).

7.8.2 Theorems

Erdős-Gallai: 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 $k = 1 \ldots n$,

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

Tutte's: A graph (V, E) has a perfect matching iff for all $U \subseteq V$, the subgraph induced by $V \setminus U$ has at most $|U|$ components with an odd number of vertices.

Petersens: Every cubic, bridgeless graph contains a perfect matching.

Dilworth's: In a finite poset, the maximum size of an antichain equals the minimum number of chains needed to partition the poset.

König's: In a bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.

Hall's: A bipartite graph (X, Y, E) has an X-saturating matching iff for all $W \subseteq X$, $|N(W)| \geq |W|$, i.e. it has as many neighbors as elements.

Geometry (8)

8.1 Geometric primitives

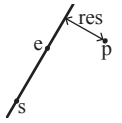
Point.h

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

b838b5, 28 lines

```
template <class T> ll 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 ll. It uses products in intermediate steps so watch out for overflow if using int or ll. Using Point3D will always give a non-negative distance. For Point3D, call .dist on the result of the cross product.

"Point.h" f6bf6b, 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();
}
```

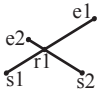
SegmentDistance.h
Description:

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" 5c88f4, 6 lines

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

SegmentIntersection.h

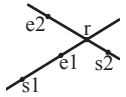


Description:
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.
Usage: `vector<P> inter = segInter(s1,e1,s2,e2); if (SZ(inter)==1) cout << "segments intersect at " << inter[0] << endl;`

"Point.h", "OnSegment.h" 36c2d7, 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.fst == 1) cout << "intersection point at " << res.snd << endl;`

"Point.h" 6a9e96, 8 lines

```
template<class P>
pair<ll, 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 ll. It uses products in intermediate steps so watch out for overflow if using int or ll.
Usage: `bool left = sideOf(p1,p2,q)==1;`

"Point.h" 89ff34, 9 lines

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

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

OnSegment.h

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

"Point.h" c597e8, 3 lines

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

HalfplaneIntersection.h

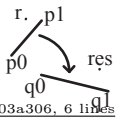
Description: Computes the intersection of a set of half-planes. Input is given as a set of planes, facing left. The intersection must form a convex polygon or be empty. Output is the convex polygon representing the intersection in CCW order. The points may have duplicates and be collinear.
Time: $\mathcal{O}(n \log n)$

"Point.h", "sideOf.h", "lineIntersection.h" e5a703, 34 lines

```
typedef Point<double> P;
struct Line {
  P p, q;
  double a;
  Line() {}
  Line(P p, P q) : p(p), q(q), a((q - p).angle()) {}
  bool operator<(Line o) const { return a < o.a; }
};
#define L(a) a.p, a.q
#define PQ(a) (a.q - a.p)

vector<P> halfPlaneIntersection(vector<Line> v) {
  sort(ALL(v));
  ll n = SZ(v), q = 1, h = 0;
  const double eps = 1e-9;
  vector<Line> c(n+2);
  #define I(j, k) lineInter(L(c[j]), L(c[k])).snd
  fore(i, 0, n) {
    while (q < h && sideOf(L(v[i]), I(h, h-1), eps) < 0) h--;
    while (q < h && sideOf(L(v[i]), I(q, q+1), eps) < 0) q++;
    c[++h] = v[i];
    if (q < h && abs(PQ(c[h]).cross(PQ(c[h-1]))) < eps) {
      if (PQ(c[h]).dot(PQ(c[h-1])) <= 0) return {};
      if (sideOf(L(v[i]), c[--h].p, eps) < 0) c[h] = v[i];
    }
  }
  while (q < h - 1 && sideOf(L(c[q]), I(h, h-1), eps) < 0) h--;
  while (q < h - 1 && sideOf(L(c[h]), I(q, q+1), eps) < 0) q++;
  if (h - q <= 1) return {};
  c[++h] = c[q];
  vector<P> s;
  fore(i, q, h) s.pb(I(i, i+1));
  return s;
}
```

linearTransformation.h



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

"Point.h" 03a306, 6 lines

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

LineProjectionReflection.h

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

"Point.h" b5562d, 5 lines

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

Angle.h

Description: A class for ordering angles (as represented by ll 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 ll j = 0; fore(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`

c0c63f, 35 lines

```
struct Angle {
  ll x, y;
  ll t;
  Angle(ll x, ll y, ll t=0) : x(x), y(y), t(t) {}
  Angle operator-(Angle b) const { return {x-b.x, y-b.y, t}; }
  ll 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
  ll 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" 84d6d3, 11 lines

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

CircleTangents.h

Description: Finds the external tangents of two circles, or internal if r2 is negated. Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .fst = .snd and the tangent line is perpendicular to the line between the centers). .fst and .snd 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" 31cca4, 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.pb({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" e0cfba, 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

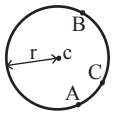
Description: Returns the area of the intersection of a circle with a ccw polygon.

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

"../../content/geometry/Point.h" b4f879, 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;
  fore(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" 1caa3a, 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" aec76e, 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;
  fore(i,0,SZ(ps)) if ((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    fore(j,0,i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      fore(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: $\mathcal{O}(n)$

"Point.h", "OnSegment.h", "SegmentDistance.h" 3ab66b, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
  ll cnt = 0, n = SZ(p);
  fore(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" b649a8, 6 lines

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

PolygonCenter.h

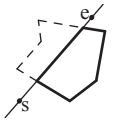
Description: Returns the center of mass for a polygon.

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

"Point.h" 3bf296, 9 lines

```
typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (ll 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" 5ea2a9, 13 lines

```
typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  fore(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.pb(lineInter(s, e, cur, prev).snd);
    if (side)
      res.pb(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: $\mathcal{O}(N^2)$, where N is the total number of points

"Point.h", "sideOf.h" de5582, 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;
  fore(i,0,SZ(poly)) fore(v,0,SZ(poly[i])) {
```

```
    P A = poly[i][v], B = poly[i][(v + 1) % SZ(poly[i])];
    vector<pair<double, ll>> segs = {{0, 0}, {1, 0}};
    fore(j,0,SZ(poly)) if (i != j) {
      fore(u,0,SZ(poly[j])) {
        P C = poly[j][u], D = poly[j][(u + 1) % SZ(poly[j])];
        ll 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.pb({sa / (sa - sb), sgn(sc - sd)});
        } else if (!sc && !sd && j<i && sgn((B-A).dot(D-C))>0){
          segs.pb({rat(C - A, B - A), 1});
          segs.pb({rat(D - A, B - A), -1});
        }
      }
    }
    sort(ALL(segs));
    for (auto& s : segs) s.fst = min(max(s.fst, 0.0), 1.0);
    double sum = 0;
    ll cnt = segs[0].snd;
    fore(j,1,SZ(segs)) {
      if (!cnt) sum += segs[j].fst - segs[j - 1].fst;
      cnt += segs[j].snd;
    }
    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" a592da, 13 lines

```
template<class P>
vector<P> convexHull(vector<P> pts) {
  if (SZ(pts) <= 1) return pts;
  sort(ALL(pts));
  vector<P> h(SZ(pts)+1);
  ll s = 0, t = 0;
  for (ll 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).
Time: $\mathcal{O}(n)$

"Point.h" 9df441, 12 lines

```
typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
  ll n = SZ(S), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
  for (ll i = 0; i < j; i++)
    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.snd;
}
```

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" a211ec, 14 lines

```
typedef Point<ll> P;

bool inHull(const vector<P>& l, P p, bool strict = true) {
  ll 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) {
    ll 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" 5eb565, 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> ll extrVertex(vector<P>& poly, P dir) {
  ll n = SZ(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    ll m = (lo + hi) / 2;
    if (extr(m)) return m;
    ll 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<ll, 2> lineHull(P a, P b, vector<P>& poly) {
  ll endA = extrVertex(poly, (a - b).perp());
  ll endB = extrVertex(poly, (b - a).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<ll, 2> res;
  fore(i,0,2) {
    ll lo = endB, hi = endA, n = SZ(poly);
    while ((lo + 1) % n != hi) {
      ll 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;
}
```

MinkowskiSum.h

Description: Compute Minkowski sum of two strictly convex non empty polygons (i.e. two hulls). Returns answer in CCW order. The Minkowski sum of two polygons P and Q viewed as sets of $\mathbb{R}^2$ is defined as $\{p+q : p \in P, q \in Q\}$
Time: $\mathcal{O}(n \log n + m \log m)$

"Point.h", "ConvexHull.h", "sideOf.h" 19e096, 23 lines

```
typedef Point<ll> P;

void reorder(vector<P> &p) {
  if (sideOf(p[0], p[1], p[2]) < 0) reverse(ALL(p));
  rotate(p.begin(), min_element(ALL(p)), p.end());
}
vector<P> minkowskiSum(vector<P> p, vector<P> q) {
  if (min(SZ(p), SZ(q)) < 3) {
    vector<P> v;
    for (P pp : p) for (P qq : q) v.pb(pp + qq);
    return convexHull(v);
  }
  reorder(p), reorder(q);
  fore(i, 0, 2) p.pb(p[i]), q.pb(q[i]);
  vector<P> r;
  ll i = 0, j = 0;
  while (i + 2 < SZ(p) || j + 2 < SZ(q)) {
    r.pb(p[i] + q[j]);
    ll cross = (p[i + 1] - p[i]).cross(q[j + 1] - q[j]);
    i += cross >= 0, j += cross <= 0;
  }
  return r;
}
```

8.4 Misc. Point Set Problems

ClosestPair.h

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

"Point.h" 4a68f8, 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()}};
  ll j = 0;
  for (P p : v) {
    P d{1 + (ll)sqrt(ret.fst), 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.snd;
}
```

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" aa0ef5, 23 lines

```
typedef Point<ll> P;
vector<array<ll, 3>> manhattanMST(vector<P> ps) {
```

```
  vi id(SZ(ps));
  iota(ALL(id), 0);
  vector<array<ll, 3>> edges;
  fore(k,0,4) {
    sort(ALL(id), [&](ll i, ll j) {
      return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;});
    map<ll, ll> sweep;
    for (ll i : id) {
      for (auto it = sweep.lower_bound(-ps[i].y);
          it != sweep.end(); sweep.erase(it++)) {
        ll j = it->snd;
        P d = ps[i] - ps[j];
        if (d.y > d.x) break;
        edges.pb({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" 58a1a5, 63 lines

```
typedef ll 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)
      ll 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->fst) {
      // 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->fst, *s = node->snd;
    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.fst)
      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);
  }
};
```

DelaunayTriangulation.h

Description: Computes the Delaunay triangulation of a set of points. Each circumcircle contains none of the input points. If any three points are collinear or any four are on the same circle, behavior is undefined.

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

"Point.h", "3dHull.h" bb23ab, 8 lines

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

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" 55602c, 88 lines

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

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

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
      B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B > 0;
}
Q makeEdge(P orig, P dest) {
  Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}}};
  H = r->o; r->r()->r() = r;
  fore(i,0,4) r = r->rot, r->p = arb, r->o = i&1 ? r : r->r();
  r->p = orig; r->F() = dest;
  return r;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

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

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

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

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

8.5 3D

PolyhedronVolume.h

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

3058c3, 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 ll.

8058ae, 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;
  }
};
```

PlaneDistance.h

Description: Returns the signed distance between point p and the plane containing points a, b and c. a, b and c must not be collinear.

"Point3D.h" c94293, 5 lines

```
template<class P>
double planeDist(P a, P b, P c, P p) {
  P n = (b-a).cross(c-a);
  return (double)n.dot(p-a) / n.dist();
}
```

sideOfPlane.h

Description: Returns at witch side of the plane defined by a, b and c the point p is. If the point is on the plane 0 is returned, else 1 or -1. a, b and c must not be collinear. For double add epsilon checks.

"Point.h" b4ebf8, 5 lines

```
template<class P>
ll sideOf(P a, P b, P c, P p) {
  ll x = (b-a).cross(c-a).dot(p-a);
  return (x > 0) - (x < 0);
}
```

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" 235fcd, 49 lines

```
typedef Point3D<double> P3;

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

struct F { P3 q; ll 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 = [&](ll i, ll j, ll k, ll 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.pb(f);
  };
  fore(i,0,4) fore(j,i+1,4) fore(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);

  fore(i,4,SZ(A)) {
    for (ll j = 0; j < SZ(FS); j++) {
      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();
      }
    }
    ll nw = SZ(FS);
    fore(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.

611f07, 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: $\mathcal{O}(n)$

1bc4d4, 16 lines

```
vi pi(const string& s) {
  vi p(SZ(s));
  fore(i,1,SZ(s)) {
    ll 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;
  fore(i,SZ(p)-SZ(s),SZ(p))
    if (p[i] == SZ(pat)) res.pb(i - 2 * SZ(pat));
  return res;
}
```

Zfunc.h

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

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

9e784e, 12 lines

```
vi Z(const string& S) {
  vi z(SZ(S));
  ll l = -1, r = -1;
  fore(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: $\mathcal{O}(N)$

c6bbec, 13 lines

```
array<vi, 2> manacher(const string& s) {
  ll n = SZ(s);
  array<vi,2> p = {vi(n+1), vi(n)};
  fore(z,0,2) for (ll i=0,l=0,r=0; i < n; i++) {
    ll t = r-i+!z;
    if (i<r) p[z][i] = min(t, p[z][l+t]);
    ll 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: $\mathcal{O}(N)$

b4d34c, 11 lines

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

SuffixArray.h

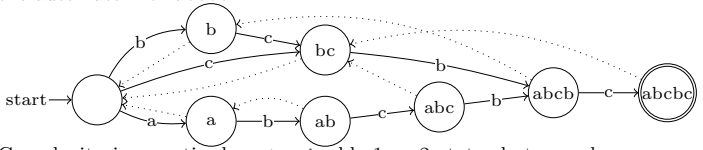
Description: Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. `rank` is the inverse of the suffix array: `rank[sa[i]] = i`. `lim` should be strictly larger than all elements. For larger alphabets use `basic_string<ll>` instead of `string`. The input string must not contain any zero bytes.
Time: $\mathcal{O}(n \log n)$

6f3b67, 17 lines

```
array<vi, 3> suffixArray(string& s, ll lim = 'z' + 1) {
  ll n = SZ(s) + 1, k = 0, a, b;
  vi rank(ALL(s)+1), y(n), ws(max(n,lim)), sa(n), lcp(n);
  iota(ALL(sa), 0);
  for (ll j = 0, p = 0; p < n; j = max(1ll, j * 2), lim = p) {
    p = j, iota(ALL(y), n - j), fill(ALL(ws), 0);
    fore(i,0,n) if (ws[rank[i]]++, sa[i]>=j) y[p++] = sa[i]-j;
    fore(i, 1, lim) ws[i] += ws[i - 1];
    for (ll i = n; i--;) sa[--ws[rank[y[i]]]] = y[i];
    swap(rank, y), p = 1, rank[sa[0]] = 0;
    fore(i, 1, n) a = sa[i - 1], b = sa[i], rank[b] =
      (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
  }
  for(ll 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++);
  return {sa, lcp, rank};
}
```

SuffixAutomaton.h

Description: Online algorithm for minimal deterministic finite automaton that accepts the suffixes of a string `s`.
Exactly all substrings of `s` are represented by the states, each state representing one or more substrings. Let `t` the longest string represented by state `v`. Then `v.len == SZ(t)`, all strings represented by `v` only appear in `s` as a suffix of `t` and they are the longest suffixes of `t`. The rest of the suffixes of `t` are found by following the suffix links `v.l`. `p` is the state representing `s` so terminal states are the ones in the path from `p` to the root through suffix links. Also suffix links form the suffix tree of reversed `s`. Here you can see the automaton for abcbc:



Complexity is amortized: `extend` adds 1 or 2 states but can change many suffix links. Up to 2N states and 3N transitions. For larger alphabets, use `T = ll`. For performance consider `s.reserve(2*N)`, and replacing `map` with `vector` or `unordered_map`.
Time: $\mathcal{O}(N \log K)$.

ada074, 16 lines

```
template <class T = char> struct SuffixAutomaton {
  struct State { ll len = 0, l = -1; map<T, ll> t; };
  vector<State> s{1}; ll p = 0;
  void extend(T c) {
    ll k = SZ(s), q; s.pb({s[p].len+1});
    for(;p != -1 && !s[p].t.count(c); p = s[p].l)s[p].t[c] = k;
    if (p == -1) s[k].l = 0;
    else if (s[p].len + 1 == s[q = s[p].t[c]].len) s[k].l = q;
    else {
      s.pb(s[q]), s.back().len = s[p].len + 1;
      for (; p!=-1 && s[p].t[c]==q; p=s[p].l) s[p].t[c] = k+1;
      s[q].l = s[k].l = k+1;
    }
    p = k;
  }
};
```

SuffixTree.h

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).
Time: $\mathcal{O}(26N)$

242d5e, 48 lines

```
struct SuffixTree {
  static constexpr ll ALPHA = 26; // alphabet size
  ll toi(char c) { return c - 'a'; }
  string a;
  ll N, v = 0, q = 0, m = 2; // v = cur node, q = cur position
  vector<vi> t; // transitions
  vi r, l, p, s; // a[l[i]:r[i]] is substring on edge to i

  void ukkadd(ll i, ll c) {
    if (r[v] <= q)
      if (t[v][c] == -1) return l[t[v][c] = m] = i,
        q = r[v = s[p[m++] = v]], ukkadd(i, c);
      else q = l[v = t[v][c]];
    if (q == -1 || c == toi(a[q])) q++;
    else {
      l[m+1] = i, l[m] = l[v], r[m] = l[v] = q, p[m] = p[v];
      p[t[m][c] = m+1] = p[v] = t[p[m]][toi(a[l[m]])] = m;
      t[m][toi(a[q])] = v, 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, ukkadd(i, c);
    }
  }

  SuffixTree(string a) : a(a), N(2*SZ(a)+2), t(N,vi(ALPHA,-1)){
    r = vi(N, SZ(a)), l = p = s = vi(N), t[1] = vi(ALPHA, 0);
    s[0] = 1, l[0] = l[1] = -1, r[0] = r[1] = p[0] = p[1] = 0;
    fore(i,0,SZ(a)) ukkadd(i, toi(a[i]));
  }

  // example: find longest common substring (uses ALPHA = 28)
  ii best;
  ll lcs(ll node, ll i1, ll i2, ll olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    ll mask = 0, len = node ? olen + (r[node] - l[node]) : 0;
    fore(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 ii 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;
  }
};
```

Hashing.h

Description: Self-explanatory methods for string hashing.

088baa, 48 lines

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

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

vector<H> getHashes(string& str, ll length) {
  if (SZ(str) < length) return {};
  H h = 0, pw = 1;
  fore(i,0,length)
    h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h};
  fore(i,length,SZ(str)) {
    ret.pb(h = h * C + str[i] - pw * str[i-length]);
  }
  return ret;
}

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

// hashString(s + t) = concat(hashString(s), hashString(t), pw)
// Where pw is C**|t| and can be obtained from a HashInterval
H concat(H h0, H h1, H h1pw) { return h0 * h1pw + h1; }
```

AhoCorasick.h

Description: AhoCorasick automaton. It consists of a trie in with each node except the root has a link to the longest suffix that is also a node in the trie. That link is used for transitions that are not defined in the trie. Works with vectors, but for lower case latin strings you can to convert it to `vi` and subtract 'a' for each character. Use the function `go` to get the next state of the automaton given the current state. Use `t[v].leaf` to know witch strings end in the state `v`.
Time: $\mathcal{O}(N)$ for constructing where N is the sum of lengths of the words and $\mathcal{O}(1)$ for each transition query.

072030, 30 lines

```
struct AhoCorasick {
  static const ll alpha = 26; // Size of the alphabet
  struct Node {
    array<ll, alpha> next, go; vi leaf; ll p, link, pch;
    Node(ll p = -1, ll pch = -1) : p(p), link(-1), pch(pch) {
      next.fill(-1), go = next;
    }
  };
  vector<Node> t;
  AhoCorasick(vector<vi>& words) : t(1) {
    fore(i, 0, SZ(words)) {
      ll v = 0;
      for (ll c : words[i]) {
        if (t[v].next[c]<0) t[v].next[c]=SZ(t),t.pb(Node(v,c));
        v = t[v].next[c];
      }
      t[v].leaf.pb(i);
    }
  }
  ll getLink(ll v) {
    if (t[v].link < 0) t[v].link = v && t[v].p ?
      go(getLink(t[v].p), t[v].pch) : 0;
    return t[v].link;
  }
  ll go(ll v, ll c) {
    if (t[v].go[c] < 0) t[v].go[c] = t[v].next[c] >= 0 ?
      t[v].next[c] : v ? go(getLink(v),c) : 0;
    return t[v].go[c];
  }
};
```

Various (10)

10.1 Dates

Dates.h

Description: Convert dates to numbers and vice versa. Days and months start from 1. 1/1/1 is day number 1721426.
Time: $\mathcal{O}(1)$

a52f52, 14 lines

```
ll dateToInt(ll y, ll m, ll d) {
  return 1461*(y+4800+(m-14)/12)/4 + 367*(m-2-(m-14)/12*12)/12
    - 3*((y+4900+(m-14)/12)/100)/4 + d - 32075;
}
tuple<ll, ll, ll> intToDate(ll jd) {
  ll x = jd + 68569, n = 4*x/146097;
  x -= (146097*n + 3)/4;
  ll i = (4000*(x + 1))/1461001;
  x -= 1461*i/4 - 31;
  ll j = 80*x/2447, d = x - 2447*j/80;
  x = j/11;
  ll m = j + 2 - 12*x, y = 100*(n - 49) + i + x;
  return {y, m, d};
}
```

DayOfWeek.h

Description: Get the day of the week for a given date. Days and months start from 1 and days of the week are returned starting on Sunday from 0.
Time: $\mathcal{O}(1)$

ef94b0, 5 lines

```
ll DayOfWeek(ll y, ll m, ll d) {
  vi ttt = {0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4};
  y -= m < 3;
  return (y + y/4 - y/100 + y/400 + ttt[m-1] + d) % 7;
}
```

10.2 Intervals

IntervalContainer.h

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

76239e, 23 lines

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

void removeInterval(set<ii>& is, ll L, ll R) {
  if (L == R) return;
  auto it = addInterval(is, L, R);
  auto r2 = it->snd;
  if (it->fst == L) is.erase(it);
  else (ll&)it->snd = L;
  if (R != r2) is.emplace(R, r2);
}
```

IntervalCover.h

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

9ddd82, 19 lines

```
template<class T>
vi cover(pair<T, T> G, vector<pair<T, T>> I) {
  vi S(SZ(I)), R;
  iota(ALL(S), 0);
  sort(ALL(S), [&](ll a, ll b) { return I[a] < I[b]; });
  T cur = G.fst;
  ll at = 0;
  while (cur < G.snd) { // (A)
    pair<T, ll> mx = {cur, -1};
    while (at < SZ(I) && I[S[at]].fst <= cur) {
      mx = max(mx, {I[S[at]].snd, S[at]});
      at++;
    }
    if (mx.snd == -1) return {};
    cur = mx.fst;
    R.pb(mx.snd);
  }
  return R;
}
```

ConstantIntervals.h

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

eb2579, 17 lines

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

10.3 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: `ll ind = ternSearch(0,n-1,[&](ll i) {return a[i];});`
Time: $\mathcal{O}(\log(b-a))$

310703, 10 lines

```
ll ternSearch(ll a, ll b, auto&& f) {
  assert(a <= b);
  while (b - a >= 5) {
    ll mid = (a + b) / 2;
    if (f(mid) < f(mid+1)) a = mid; // (A)
    else b = mid+1;
  }
  fore(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
  return a;
}
```

LIS.h

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

00efff, 17 lines

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

FastKnapsack.h

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

315e89, 16 lines

```
ll knapsack(vi w, ll t) {
  ll a = 0, b = 0, x;
  while (b < SZ(w) && a + w[b] <= t) a += w[b++];
  if (b == SZ(w)) return a;
  ll m = *max_element(ALL(w));
  vi u, v(2*m, -1);
```

```
  v[a+m-t] = b;
  fore(i,b,SZ(w)) {
    u = v;
    fore(x,0,m) v[x+w[i]] = max(v[x+w[i]], u[x]);
    for (x = 2*m; --x > m;) fore(j, max(0ll,u[x]), v[x])
      v[x-w[j]] = max(v[x-w[j]], j);
  }
  for (a = t; v[a+m-t] < 0; a--) ;
  return a;
}
```

10.4 Dynamic programming

KnuthDP.h

Description: When doing DP on intervals: $a[i][j] = \min_{i<k<j}(a[i][k] + a[k][j]) + f(i,j)$, where the (minimal) optimal k increases with both i and j, one can solve intervals in increasing order of length, and search $k = p[i][j]$ for $a[i][j]$ only between $p[i][j-1]$ and $p[i+1][j]$. This is known as Knuth DP. Sufficient criteria for this are if $f(b,c) \le f(a,d)$ and $f(a,c) + f(b,d) \le f(a,d) + f(b,c)$ for all $a \le b \le c \le d$. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.
Time: $\mathcal{O}\left(N^2\right)$

DivideAndConquerDP.h

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

64b06e, 18 lines

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

  void rec(ll L, ll R, ll LO, ll HI) {
    if (L >= R) return;
    ll mid = (L + R) >> 1;
    ii best(LLONG_MAX, LO);
    for (ll k = max(LO, lo(mid)); k < min(HI, hi(mid)); k++)
      best = min(best, make_pair(f(mid, k), k));
    store(mid, best.snd, best.fst);
    rec(L, mid, LO, best.snd+1);
    rec(mid+1, R, best.snd, HI);
  }
  void solve(ll L, ll R) { rec(L, R, LLONG_MIN, LLONG_MAX); }
};
```

10.5 Debugging tricks

- `signal(SIGSEGV, [](int) { _Exit(0); });` converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions). `_GLIBCXX_DEBUG` failures generate SIGABRT (or SIGSEGV on gcc 5.4.0 apparently).
- `feenableexcept(29);` kills the program on NaNs (1), 0-divs (4), infinities (8) and denormals (16).
- `-fsanitize=address -g` in compilation to detect some memory access errors at run time.
- Several runtime checks:
  `#define _GLIBCXX_DEBUG 1`
  `#define _GLIBCXX_DEBUG_PEDANTIC 1`
  `#define _GLIBCXX_CONCEPT_CHECKS 1`
  `#define _GLIBCXX_SANITIZE_VECTOR 1`

10.6 Optimization tricks

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

10.6.1 Bit hacks

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

10.6.2 Pragmas

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

10.6.3 Other

- `cin.exceptions(cin.failbit);` will make `cin` throw on bad input.

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 $[0, 2b)$.

751a02, 8 lines

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

Randin.h

Description: Fast and secure integer uniform random numbers. For floating point numbers use uniform_real_distribution.
Time: 2x faster than rand().

1108e7, 7 lines

```
template <typename T>
T randin(T a, T b) { // Random number in [a, b)
  static random_device rd;
  static mt19937_64 gen(rd());
  uniform_int_distribution<T> dis(a, b - 1);
  return dis(gen);
}
```

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.

d42732, 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
}

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

BumpAllocator.h

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

d23557, 8 lines

```
// Either globally or in a single class:
static char buf[900 << 20];
void* operator new(size_t s) {
  static size_t i = sizeof buf;
  assert(s < i);
  return (void*)&buf[i -= s];
}
void operator delete(void*) {}
```

SmallPtr.h

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

"BumpAllocator.h" 2dd6c9, 10 lines

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

BumpAllocatorSTL.h

Description: BumpAllocator for STL containers.
Usage: `vector<vector<ll, small<ll>>> ed(N);`

bb66d4, 14 lines

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

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

```
};
```

SIMD.h

Description: Cheat sheet of SSE/AVX intrinsics, for doing arithmetic on several numbers at once. Can provide a constant factor improvement of about 4, orthogonal to loop unrolling. Operations follow the pattern "_mm(256)?_name_(si(128|256)|epi(8|16|32|64)|pd|ps)". Not all are described here; grep for _mm_ in /usr/lib/gcc/*/4.9/include/ for more. If AVX is unsupported, try 128-bit operations, "emmintrin.h" and #define __SSE__ and __MMX__ before including it. For aligned memory use _mm_malloc(size, 32) or int buf[N] alignas(32), but prefer loadu/storeu.

d1ecdc, 43 lines

```
#pragma GCC target ("avx2") // or sse4.1
#include "immintrin.h"

typedef __m256i mi;
#define L(x) _mm256_loadu_si256((mi*)&(x))

// High-level/specific methods:
// load(u)?_si256, store(u)?_si256, setzero_si256, _mm_malloc
// blendv_(epi8|ps|pd) (z?y:x), movemask_epi8 (hibits of bytes)
// i32gather_epi32(addr, x, 4): map addr[] over 32-b parts of x
// sad_epu8: sum of absolute differences of u8, outputs 4xi64
// maddubs_epi16: dot product of unsigned i7's, outputs 16xi15
// madd_epi16: dot product of signed i16's, outputs 8xi32
// extractf128_si256(, i) (256->128), cvtsi128_si32 (128->lo32)
// permute2f128_si256(x,x,1) swaps 128-bit lanes
// shuffle_epi32(x, 3*64+2*16+1*4+0) == x for each lane
// shuffle_epi8(x, y) takes a vector instead of an imm

// Methods that work with most data types (append e.g. _epi32):
// set1, blend (i8?x:y), add, adds (sat.), mullo, sub, and/or,
// andnot, abs, min, max, sign(1,x), cmp(gt|eq), unpack(lo|hi)

int sumi32(mi m) { union {int v[8]; mi m;} u; u.m = m;
  int ret = 0; fore(i,0,8) ret += u.v[i]; return ret; }
mi zero() { return _mm256_setzero_si256(); }
mi one() { return _mm256_set1_epi32(-1); }
bool all_zero(mi m) { return _mm256_testz_si256(m, m); }
bool all_one(mi m) { return _mm256_testc_si256(m, one()); }

int example_filteredDotProduct(int n, short* a, short* b) {
  int i = 0; int r = 0;
  mi zero = _mm256_setzero_si256(), acc = zero;
  while (i + 16 <= n) {
    mi va = L(a[i]), vb = L(b[i]); i += 16;
    va = _mm256_and_si256(_mm256_cmpgt_epi16(vb, va), va);
    mi vp = _mm256_madd_epi16(va, vb);
    acc = _mm256_add_epi64(_mm256_unpacklo_epi32(vp, zero),
      _mm256_add_epi64(acc, _mm256_unpackhi_epi32(vp, zero)));
  }
  union {int v[4];mi m;} u; u.m=acc; fore(i,0,4) r += u.v[i];
  for (;i<n;++i) if (a[i] < b[i]) r += a[i]*b[i]; // <- equiv
  return r;
}
```

Techniques (A)

techniques.txt 159 lines

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