UNSW

); DROP TABLE Teams;–

Miles Conway, Alex Wang, Yiheng You

2025-09-28

Contest (1)

template.cpp — 21 lines

```cpp
#include <bits/stdc++.h>

using namespace std;
using ll = long long;

#define FOR(i, a, b) for (int i = a; i < (b); i++)
#define F0R(i, b) FOR(i, 0, b)
#define all(x) begin(x), end(x)
#define vt vector
#define size(x) ((int) (x).size())
#define ROF(i, a, b) for (int i = (b) - 1; i >= (a); i--)
#define pb push_back
#define f first
#define s second
using vi = vt<int>;

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

}
```

hash.sh — 3 lines

```sh
# 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
```

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?
```

stress.sh — 10 lines

```sh
# A and B are executables you want to compare, gen takes int
# as command line arg. Usage: 'sh stress.sh'
for((i = 1; ; ++i)); do
    echo $i
    ./gen $i > int
    ./A < int > out1
    ./B < int > out2
    diff -w out1 out2 || break
    # diff -w <(./A < int) <(./B < int) || break
done
```

Mathematics (2)

2.1 Equations

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

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

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

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

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 Triangles

Side lengths: a, b, c

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

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

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

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

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

Length of bisector (divides angles in two):

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

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

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

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

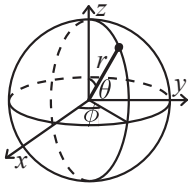
2.4.2 Quadrilaterals

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

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

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

2.4.3 Spherical coordinates



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

2.5 Derivatives/Integrals

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

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

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

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

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

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

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

2.7 Series

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

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

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

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

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

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) $\mu = \mathbb{E}(X) = \sum_x xp_X(x)$ 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 $\operatorname{Bin}(n,p)$, $n = 1, 2, \ldots$, $0 \leq p \leq 1$.

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

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

$\operatorname{Bin}(n,p)$ is approximately $\operatorname{Po}(np)$ 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 $\operatorname{Fs}(p)$, $0 \leq p \leq 1$.

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

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

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 $\operatorname{Po}(\lambda)$, $\lambda = t\kappa$.

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

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

2.8.2 Continuous distributions

Uniform distribution

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

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

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

Exponential distribution

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

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

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

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.

Stationary 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.

Ergodicity

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). $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{1}\pi$.

Absorption

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

782797, 16 lines

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

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

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

HashMap.h

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

d77092, 7 lines

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

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

668e20, 22 lines

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

FenwickTree2d.h

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

"FenwickTree.h" fa484c, 23 lines

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

SegmentTree.h

Description: Generic segment tree for associative operations. `id` is for the identity object.
Time: $\mathcal{O}(\log N)$

f519dc, 20 lines

```
struct Segtree {
  int n;
  vt<T> seg;
  void init(int _n) {
    for (n = 1; n < _n; n *= 2);
    seg.resize(2 * n, id);
  }
  void upd(int i, T v) {
    seg[i += n] = v;
    while (i /= 2) seg[i] = seg[2 * i] + seg[2 * i + 1];
  }
  T query(int l, int r) {
    T lhs = id, rhs = id;
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
      if (l & 1)) lhs = lhs + seg[l++];
      if (r & 1) rhs = seg[--r] + rhs;
    }
    return lhs + rhs;
  }
};
```

LazySegtree.h

Description: Generic lazy segment tree
Usage: Implement +, upd and id for Node object; += and id for Lazy object.
Time: $\mathcal{O}(\log N)$.

683133, 49 lines

```
struct LazySeg {
  int n;
  vt<Node> seg;
  vt<Lazy> lazy;
  void init(int _n) {
    for (n = 1; n < _n; n *= 2);
    seg.resize(2 * n, nid);
    lazy.resize(2 * n, lid);
  }
  void pull(int i) {
    seg[i] = seg[2 * i] + seg[2 * i + 1];
  }
  void push(int i, int l, int r) {
    seg[i].upd(lazy[i], l, r);
    if (r - l > 1) F0R (j, 2) lazy[2 * i + j] += lazy[i];
    lazy[i] = lid;
  }
  void build() {
    for (int i = n - 1; i > 0; i--) pull(i);
  }
  void upd(int lo, int hi, Lazy val) { upd(lo, hi, val, 1, 0, n); }
  void upd(int lo, int hi, Lazy val, int i, int l, int r) {
    if (r == -1) r = n;
    push(i, l, r);
    if (r <= lo || l >= hi) return;
    if (lo <= l && r <= hi) {
      lazy[i] += val;
      push(i, l, r);
      return;
    }
    int m = (l + r) / 2;
```

```
    upd(lo, hi, val, 2 * i, l, m);
    upd(lo, hi, val, 2 * i + 1, m, r);
    pull(i);
  }
  Node query() { return query(0, n, 1, 0, n); }
  Node query(int lo, int hi) { return query(lo, hi, 1, 0, n); }
  Node query(int lo, int hi, int i, int l, int r) {
    push(i, l, r);
    if (r <= lo || l >= hi) return nid;
    if (lo <= l && r <= hi) return seg[i];
    int m = (l + r) / 2;
    return query(lo, hi, 2 * i, l, m)
      + query(lo, hi, 2 * i + 1, m, r);
  }
  Node& operator[](int i) {
    return seg[i + n];
  }
};
```

SparseSegtree.h

Description: Generic-ish sparse segment tree (point update, range query).
Usage: Choose appropriate identity element and merge function.
Time: $\mathcal{O}(\log N)$.

d51c9b, 28 lines

```
using ptr = struct Node*;
const int sz = 1 << 30;
struct Node {
  #define func(a, b) min(a, b)
  #define ID INF
  ll val;
  ptr lc, rc;

  ptr get(ptr& p) { return p ? p : p = new Node {ID}; }

  ll query(int lo, int hi, int l = 0, int r = sz) {
    if (lo >= r || hi <= l) return ID;
    if (lo <= l && r <= hi) return val;
    int m = (l + r) / 2;
    return func(get(lc)->query(lo, hi, l, m),
      get(rc)->query(lo, hi, m, r));
  }

  ll upd(int i, ll nval, int l = 0, int r = sz) {
    if (r - l == 1) return val = nval;
    int m = (l + r) / 2;
    if (i < m) get(lc)->upd(i, nval, l, m);
    else get(rc)->upd(i, nval, m, r);
    return val = func(get(lc)->val, get(rc)->val);
  }
  #undef ID
  #undef func
};
```

PersistentSegtree.h

Description: Generic-ish persistent segment tree (point update, range query).
Usage: Choose appropriate identity element and merge function.
Time: $\mathcal{O}(\log N)$.

f155de, 30 lines

```
using ptr = struct Node*;
const int sz = 1 << 18;

struct Node {
  #define func(a, b) min(a, b)
  #define ID inf
  int v;
  ptr lc, rc;

  ptr pull(ptr lc, ptr rc) {
    return new Node {func(lc->v, rc->v), lc, rc};
  }

  ptr upd(int i, int nv, int l = 0, int r = sz) {
    if (r - l == 1) return new Node {nv};
    int m = (l + r) / 2;
    if (i < m) return pull(lc->upd(i, nv, l, m), rc);
    else return pull(lc, rc->upd(i, nv, m, r));
  }

  int query(int lo, int hi, int l = 0, int r = sz) {
    if (lo >= r || hi <= l) return ID;
    if (lo <= l && r <= hi) return v;
    int m = (l + r) / 2;
    return func(lc->query(lo, hi, l, m),
      rc->query(lo, hi, m, r));
  }
  #undef id
  #undef func
};
```

LiChaoTree.h

Description: LiChao tree
Usage: self explanatory i think
Time: $\mathcal{O}(\log N)$.

00d540, 36 lines

```
struct Line {
  ll m, c;
  ll operator()(ll x) {
    return m * x + c;
  }
};

const ll sz = 1ll << 30;

using ptr = struct Node*;
struct Node {
  ptr lc, rc;
  Line line;

  Node(Line _line) {
    line = _line;
    lc = rc = 0;
  }
};

// min tree (flip signs for max)
void add(ptr& n, Line loser, ll l = 0, ll r = sz) {
  if (n ? 0 : n = new Node(loser)) return;
  ll m = (l + r) / 2;
  if (loser(m) < n->line(m)) swap(loser, n->line);
  if (r - l == 1) return;
  if (loser(l) < n->line(l)) add(n->lc, loser, l, m);
  else add(n->rc, loser, m, r);
}

ll query(ptr n, ll x, ll l = 0, ll r = sz) {
  if (!n) return sz;
  ll m = (l + r) / 2;
  if (x < m) return min(n->line(x), query(n->lc, x, l, m));
  else return min(n->line(x), query(n->rc, x, m, r));
}
```

SparseTable.h

Description: Generic sparse table for idempotent operations.
Usage: Define the desired operation
Time: $\mathcal{O}(N \log N)$ build, $\mathcal{O}(1)$ query.

3e1539, 18 lines

```
template<class T> struct RMQ {
  #define func min
  vt<vt<T>> dp;
  void init(const vt<T>& v) {
    dp.resize(32 - __builtin_clz(size(v)), vt<T>(size(v)));
    copy(all(v), begin(dp[0]));
    for (int j = 1; 1 << j <= size(v); ++j) {
      for (int i = 0; i < size(v) - (1 << j) + 1; i++)
        dp[j][i] = func(dp[j - 1][i],
          dp[j - 1][i + (1 << (j - 1))]);
    }
  }
  T query(int l, int r) {
    int d = 31 - __builtin_clz(r - l);
    return func(dp[d][l], dp[d][r - (1 << d)]);
  }
  #undef func
};
```

StaticRangeQuery.h

Description: Generic static range query for associative operations.
Usage: Define the desired operation
Time: $\mathcal{O}(N \log N)$ build, $\mathcal{O}(1)$ query.

82b9ca, 34 lines

```
template<class T> struct RangeQuery {
  #define comb(a, b) (a) + (b)
  #define id 0
  int lg, n;
  vt<vt<T>> stor;
  vt<T> a;
  void fill(int l, int r, int ind) {
    if (ind < 0) return;
    int m = (l + r) / 2;
    T prod = id;
    FOR (i, m, r) stor[i][ind] = prod = comb(prod, a[i]);
    prod = id;
    ROF (i, l, m) stor[i][ind] = prod = comb(a[i], prod);
    fill(l, m, ind - 1);
    fill(m, r, ind - 1);
  }
  template <typename It>
  void build(It l, It r) {
    lg = 1;
    while ((1 << lg) < r - l) lg++;
    n = 1 << lg;
    a.resize(n, id);
    for (It i = l; i != r; i++) a[i - l] = *i;
    stor.resize(n, vt<T>(32 - __builtin_clz(n)));
    fill(0, n, lg - 1);
  }
  T query(int l, int r) {
    if (l == r) return a[l];
    int t = 31 - __builtin_clz(r ^ l);
    return comb(stor[l][t], stor[r][t]);
  }
  #undef id
  #undef comb
};
```

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, 30 lines

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

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

IncrementalMST.h

Description: Fast incremental MST where you can also delete max weight edges. Can be used for offline dynacon.
Time: $\mathcal{O}(\log n)$ expected

de1060, 97 lines

```
struct DSU {
  vi par, pri;
  vt<pi> weight;

  DSU(int n): par(n), weight(n), pri(n) {
    for (int i = 0; i < n; ++i) {
      par[i] = pri[i] = i;
      weight[i] = pi{inf, -1};
    }
    shuffle(pri.begin(), pri.end(), mt19937(random_device{}()));
  }

  int parent(int u) {
    if (par[u] == u) return par[u];
    while (weight[par[u]].f <= weight[u].f) {
      par[u] = par[par[u]];
    }
    return par[u];
  }

  int find(int u, int w = inf - 1) {
    while (weight[u].f <= w) u = parent(u);
    return u;
  }

  void disconnect(int v) {
    if (par[v] == v) return;
    disconnect(par[v]);
  }

  int connect(int v, int w = inf - 1) {
    while (weight[v].f <= w) {
      v = par[v];
    }
    return v;
  }

  void add_edge(int u, int v, pi w) {
    disconnect(u);
    disconnect(v);
    while (u != v) {
      u = connect(u, w.f);
      v = connect(v, w.f);
      if (pri[u] < pri[v]) swap(u, v);
      swap(par[v], u);
      swap(weight[v], w);
    }
    connect(u);
  }

  int max_edge(int u, int v) {
    if (find(u) != find(v)) return -1;
    while (true) {
      if (weight[u].f > weight[v].f) swap(u, v);
      if (par[u] == v) break;
      u = par[u];
    }
    return u;
  }

  void delete_edge(int v, int w) {
    while (par[v] != v) {
      if (weight[v].f == w) {
        int u = v;
        while (par[u] != u) {
          u = par[u];
        }
        par[v] = v;
        weight[v] = {inf, -1};
        return;
      }
      v = parent(v);
    }
  }

  // delete edge with weight
  void delete_edge(int u, int v, int w) {
    delete_edge(u, w);
    delete_edge(v, w);
  }

  // return weight of deleted edge; {inf, -1} otherwise
  pi merge(int u, int v, pi w) {
    if (u == v) return w;
    int p = max_edge(u, v);
    if (p == -1) {
      add_edge(u, v, w);
      return {inf, -1};
    } else if (weight[p].f > w.f) {
      pi res = weight[p];
      delete_edge(p, weight[p].f);
      add_edge(u, v, w);
      return res;
    }
    return w;
  }
};
```

DSURollback.h

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

b90249, 23 lines

```
struct DSU {
  int n;
  vt<int> e;
  vt<vt<pi>> stk;
  void init(int _n) { n = _n; e.resize(n + 1, -1); e[n] = n; }
  void push() { stk.pb({}); }
  void pop() {
    reverse(all(stk.back()));
    for (auto [i, v] : stk.back()) e[i] = v;
    stk.pop_back();
  }
  void upd(int i, int v) { stk.back().pb({i, e[i]}); e[i] = v; }
  int find(int x) { return e[x] < 0 ? x : find(e[x]); }
  void unite(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (e[x] < e[y]) swap(x, y);
    upd(y, e[x] + e[y]);
    upd(x, y);
    upd(n, e[n] - 1);
  }
  int comps() { return e[n]; }
};
```

Dynacon.h

Description: Dynamic connectivity. Alternatively use IncrementalMST, and weight edges by deletion time (Q if never deleted).
Usage: i forgot
Time: $\mathcal{O}\left(N \log^2 N\right)$

db95f3, 37 lines

```
struct DynaCon {
  int n, q, t = 0;
  vt<vt<pi>> seg;
  map<pi, int> eds;
  DSU dsu;
  void init(int _n, int _q) {
    for (q = 1; q < _q; q *= 2);
    seg.resize(2 * q);
    dsu.init(n = _n);
  }
  void toggle(int u, int v, bool erase = true) {
    if (u > v) swap(u, v);
    if (eds.count({u, v})) {
      for (int l = eds[{u, v}] + q, r = min(++t + q, 2 * q); l < r; l
           /= 2, r /= 2) {
        if (l & 1) seg[l++].pb({u, v});
        if (r & 1) seg[--r].pb({u, v});
      }
      if (erase) eds.erase({u, v});
    } else eds[{u, v}] = t++;
  }
  void query() { seg[q + t++].pb({-1, -1}); }
  void dfs(int i, vi &ans) {
    dsu.push();
    for (auto [u, v] : seg[i]) {
      if (u == -1) ans.pb(dsu.comps());
      else dsu.unite(u, v);
    }
    if (i < q) dfs(2 * i, ans), dfs(2 * i + 1, ans);
    dsu.pop();
  }
  vi ans() {
    for (auto [u, v] : eds) toggle(u.f, u.s, false);
    vi res;
    dfs(1, res);
    return res;
  }
};
```

CaterpillowTree.h

Description: 64-ary set
Usage: bruh
Time: $\mathcal{O}\left(\log_{64} N\right)$.

426eac, 89 lines

```
using ull = unsigned long long;
const int depth = 3;
const int sz = 1 << (depth * 6);

struct Tree {
  vt<ull> seg[depth];

  Tree() {
    F0R (i, depth) seg[i].resize(1 << (6 * i));
  }

  void insert(int x) {
    ROF (d, 0, depth) {
      seg[d][x >> 6] |= 1ull << (x & 63);
      x >>= 6;
    }
  }

  void erase(int x) {
    ull b = 0;
    ROF (d, 0, depth) {
      seg[d][x >> 6] &= ~(1ull << (x & 63));
      seg[d][x >> 6] |= b << (x & 63);
      x >>= 6;
      b = bool(seg[d][x]);
    }
  }

  int next(int x) {
    if (x >= sz) return sz;
    x = std::max(x, 0);
    int d = depth - 1;
    while (true) {
      if (ull m = seg[d][x >> 6] >> (x & 63)) {
        x += __builtin_ctzll(m);
        break;
      }
      x = (x >> 6) + 1;
      if (d == 0 || x >= (1 << (6 * d))) return sz;
      d--;
    }
    while (++d < depth) {
      x = (x << 6) + __builtin_ctzll(seg[d][x]);
    }
    return x;
  }

  int prev(int x) {
    if (x < 0) return -1;
    x = std::min(x, sz - 1);
    int d = depth - 1;
    while (true) {
      if (ull m = seg[d][x >> 6] << (63 - (x & 63))) {
        x -= __builtin_clzll(m);
        break;
      }
      x = (x >> 6) - 1;
      if (d == 0 || x == -1) return -1;
      d--;
    }
    while (++d < depth) {
      x = (x << 6) + 63 - __builtin_clzll(seg[d][x]);
    }
    return x;
  }

  int min() {
    if (empty()) return sz;
    int ans = 0;
    F0R (d, depth) {
      ans <<= 6;
      ans += __builtin_ctzll(seg[d][ans >> 6]);
    }
    return ans;
  }

  int max() {
    if (empty()) return -1;
    int ans = 0;
    F0R (d, depth) {
      ans <<= 6;
      ans += 63 - __builtin_clzll(seg[d][ans >> 6]);
    }
    return ans;
  }

  inline bool empty() { return !seg[0][0]; }
  inline int operator[](int i) { return 1 & (seg[depth - 1][i >> 6] >>
        (i & 63)); }
};
```

Treap.h

Description: Treap with too many operations
Time: $\mathcal{O}(\log N)$

817442, 218 lines

```
using K = ll;
random_device rd;
mt19937 mt(rd());

struct Lazy {
  ll v;
  bool inc, rev;
  void operator+=(const Lazy &b) {
    if (b.inc) v += b.v;
    else v = b.v, inc = false;
    rev ^= b.rev;
  }
};

struct Value {
  ll mx, sum;
  void upd(const Lazy &b, int sz) {
    if (!b.inc) mx = sum = 0;
    mx += b.v, sum += b.v * sz;
  }
  Value operator+(const Value &b) const {
    return {max(mx, b.mx), sum + b.sum};
  }
};

const Lazy LID = {0, true, false};
const Value VID = {INF, 0};

using ptr = struct Node*;

struct Node {
  int pri;
  K key;
  ptr l, r;
  int sz;

  Value val, agg;
  Lazy lazy;

  Node(K key, Value val) : key(key), val(val), agg(val) {
    sz = 1;
    pri = mt();
    l = r = 0;
    lazy = LID;
  }

  ~Node() {
    delete l;
    delete r;
  }
};

int sz(ptr n) { return n ? n->sz : 0; }
Value val(ptr n) { return n ? n->val : VID; }
Value agg(ptr n) { return n ? n->agg : VID; }

ptr push(ptr n) {
  if (!n) return n;
  if (n->lazy.rev) swap(n->l, n->r);
  ptr l = n->l, r = n->r;
  n->val.upd(n->lazy, 1);
  n->agg.upd(n->lazy, n->sz);
  if (l) n->l->lazy += n->lazy;
  if (r) n->r->lazy += n->lazy;
  n->lazy = LID;
  return n;
}

ptr pull(ptr n) {
  ptr l = n->l, r = n->r;
  push(l), push(r);
  n->sz = sz(l) + 1 + sz(r);
  n->agg = agg(l) + n->val + agg(r);
  return n;
}

pair<ptr, ptr> split(ptr n, K k) {
  if (!n) return {n, n};
  push(n);
  if (k <= n->key) {
    auto [l, r] = split(n->l, k);
    n->l = r;
    return {l, pull(n)};
  } else {
    auto [l, r] = split(n->r, k);
    n->r = l;
    return {pull(n), r};
  }
}

pair<ptr, ptr> spliti(ptr n, int i) {
  if (!n) return {n, n};
  push(n);
  if (i <= sz(n->l)) {
    auto [l, r] = spliti(n->l, i);
    n->l = r;
    return {l, pull(n)};
  } else {
    auto [l, r] = spliti(n->r, i - sz(n->l) - 1);
    n->r = l;
    return {pull(n), r};
  }
}

ptr merge(ptr l, ptr r) {
  if (!l || !r) return l ? l : r;
  push(l), push(r);
  ptr t;
  if (l->pri > r->pri) l->r = merge(l->r, r), t = l;
  else r->l = merge(l, r->l), t = r;
  return pull(t);
}
```

```
ptr ins(ptr n, K k, Value val) { // insert k
  auto [l, r] = split(n, k);
  return merge(l, merge(new Node(k, val), r));
}

ptr insi(ptr n, int i, K k, Value val) { // insert before i
  auto [l, r] = spliti(n, i);
  return merge(l, merge(new Node(k, val), r));
}

ptr del(ptr n, K k) { // delete k
  auto a = split(n, k), b = spliti(a.s, 1);
  return merge(a.f, b.s);
}

ptr deli(ptr n, int i) {
  auto b = spliti(n, i + 1), a = spliti(b.f, i);
  return merge(a.f, b.s);
}

ptr find(ptr n, K k) {
  push(n);
  if (!n || n->key == k) return n;
  if (k < n->key) return find(n->l, k);
  else return find(n->r, k);
}

ptr findi(ptr n, int i) {
  push(n);
  if (!n || i == sz(n->l)) return n;
  if (i < sz(n->l)) return find(n->l, i);
  else return find(n->r, i);
}

ptr upd(ptr n, K lo, K hi, Lazy nv) {
  if (lo > hi) return n;
  auto [lhs, r] = split(n, hi + 1);
  auto [l, m] = split(lhs, lo);
  m->lazy += nv;
  return merge(l, merge(m, r));
}

ptr updi(ptr n, int lo, int hi, Lazy nv) {
  if (lo > hi) return n;
  auto [lm, r] = spliti(n, hi + 1);
  auto [l, m] = spliti(lm, lo);
  m->lazy += nv;
  return merge(l, merge(m, r));
}

Value query(ptr &n, K lo, K hi)  {
  auto [lm, r] = split(n, hi + 1);
  auto [l, m] = split(lm, lo);
  Value res = agg(m);
  n = merge(l, merge(m, r));
  return res;
}

Value queryi(ptr &n, int lo, int hi) {
  auto [lm, r] = spliti(n, hi + 1);
  auto [l, m] = spliti(lm, lo);
  Value res = agg(m);
  n = merge(l, merge(m, r));
  return res;
}

int mn(ptr n) {
  assert(n);
  push(n);
  if (n->l) return mn(n->l);
  else return n->key;
}

ptr unite(ptr l, ptr r) {
  if (!l || !r) return l ? l : r;
  // l has the smallest key
  if (mn(l) > mn(r)) swap(l, r);
  ptr res = 0;
  while (r) {
    auto [lt, rt] = split(l, mn(r) + 1);
    res = merge(res, lt);
    tie(l, r) = make_pair(r, rt);
  }
  return merge(res, l);
}

void heapify(ptr n) {
  if (!n) return;
  ptr mx = n;
  if (n->l && n->l->pri > mx->pri) mx = n->l;
  if (n->r && n->r->pri > mx->pri) mx = n->r;
  if (mx != n) swap(n->pri, mx->pri), heapify(mx);
}

ptr build(int l, int r, vt<ptr>& ns) {
  if (l > r) return nullptr;
  if (l == r) return ns[l];
  int m = (r + l) / 2;
  ns[m]->l = build(l, m - 1, ns);
  ns[m]->r = build(m + 1, r, ns);
  heapify(ns[m]);
  return pull(ns[m]);
}

Node* tree;
```

DynamicBitset.h

Description: Workaround for dynamic bitsets. Adds compile time and memory factor. Reduce multiplying factor depending on desired tradeoff.

e31360, 9 lines

```
const int MAX_LEN = 1 << 20;
template <int len = 1>
void solve(int n) {
  if (n > len) {
    solve<min(len * 2, MAX_LEN)>(n);
    return;
  }
  // solution using bitset<len>
}
```

BitsetFindPrev.h

Description: Implements find_prev for bitsets.

2e6302, 28 lines

```
template <size_t Nb>
struct Bitset : bitset<Nb> {
  template <typename... Args>
  Bitset(Args... args) : bitset<Nb>(args...) {}

  static constexpr int N = Nb;
  static constexpr int array_size = (Nb + 63) / 64;

  union raw_cast {
  array<uint64_t, array_size> a;
  Bitset b;
  };

  int _Find_prev(size_t i) const {
  if (i == 0) return -1;
  if ((*this)[--i] == true) return i;
  size_t M = i / 64;
  const auto& a = ((raw_cast*)(this))->a;
  uint64_t buf = a[M] & ((1ull << (i & 63)) - 1);
  if (buf != 0) return M * 64 + 63 - __builtin_clzll(buf);
  while (M--) {
    if (a[M] != 0) return M * 64 + 63 - __builtin_clzll(a[M]);
  }
  return -1;
  }

  inline int _Find_last() const { return _Find_prev(N); }
};
```

MoQueries.h

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

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

92a58e, 49 lines

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

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

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

```
  return res;
}
```

FastMo.h

Description: Faster mo's probably.
Time: $\mathcal{O}\left(N\sqrt{Q}\right)$ but faster?

0ee660, 78 lines

```
struct Fast_Mo {
  int N, Q, width;
  vector<int> L, R, order;
  bool is_build;
  int nl, nr;

  Fast_Mo(int _n, int _q) : N(_n), Q(_q), order(Q), is_build(false) {
  width = max<int>(1, 1.0 * N / max<double>(1.0, sqrt(Q / 2.0)));
  iota(begin(order), end(order), 0);
  }
  // [l, r)
  void insert(int l, int r) {
  assert(0 <= l and l <= r and r <= N);
  L.push_back(l), R.push_back(r);
  }

  void build() { sort(), climb(), is_build = true; }

  template <typename AL, typename AR, typename DL, typename DR,
       typename REM>
  void run(const AL &add_left, const AR &add_right, const DL &
       delete_left,
       const DR &delete_right, const REM &rem) {
  if (!is_build) build();
  nl = nr = 0;
  for (auto idx : order) {
    while (nl > L[idx]) add_left(--nl);
    while (nr < R[idx]) add_right(nr++);
    while (nl < L[idx]) delete_left(nl++);
    while (nr > R[idx]) delete_right(--nr);
    rem(idx);
  }
  }

 private:
  void sort() {
  assert((int)order.size() == Q);
  vector<int> cnt(N + 1), buf(Q);
  for (int i = 0; i < Q; i++) cnt[R[i]]++;
  for (int i = 1; i < (int)cnt.size(); i++) cnt[i] += cnt[i - 1];
  for (int i = 0; i < Q; i++) buf[--cnt[R[i]]] = i;
  vector<int> b(Q);
  for (int i = 0; i < Q; i++) b[i] = L[i] / width;
  cnt.resize(N / width + 1);
  fill(begin(cnt), end(cnt), 0);
  for (int i = 0; i < Q; i++) cnt[b[i]]++;
  for (int i = 1; i < (int)cnt.size(); i++) cnt[i] += cnt[i - 1];
  for (int i = 0; i < Q; i++) order[--cnt[b[buf[i]]]] = buf[i];
  for (int i = 0, j = 0; i < Q; i = j) {
    int bi = b[order[i]];
    j = i + 1;
    while (j != Q and bi == b[order[j]]) j++;
    if (!(bi & 1)) reverse(begin(order) + i, begin(order) + j);
  }
  }

  int dist(int i, int j) { return abs(L[i] - L[j]) + abs(R[i] - R[j]);
       }

  void climb(int iter = 3, int interval = 5) {
  vector<int> d(Q - 1);
  for (int i = 0; i < Q - 1; i++) d[i] = dist(order[i], order[i + 1]);
  while (iter--) {
    for (int i = 1; i < Q; i++) {
    int pre1 = d[i - 1];
    int js = i + 1, je = min<int>(i + interval, Q - 1);
    for (int j = je - 1; j >= js; j--) {
      int pre2 = d[j];
      int now1 = dist(order[i - 1], order[j]);
      int now2 = dist(order[i], order[j + 1]);
      if (now1 + now2 < pre1 + pre2) {
      reverse(begin(order) + i, begin(order) + j + 1);
      reverse(begin(d) + i, begin(d) + j);
      d[i - 1] = pre1 = now1;
      d[j] = now2;
      }
    }
    }
  }
  }
};
```

LeftistHeap.h

Description: Persistent meldable heap.
Memory: $\mathcal{O}\left(\log N\right)$ per meld
Time: $\mathcal{O}\left(\log N\right)$ per meld

369c01, 17 lines

```
typedef pair<ll,int> T;
typedef struct heap* ph;
struct heap { // min heap
  ph l = NULL, r = NULL;
  int s = 0; T v; // s: path to leaf
  heap(T _v):v(_v) {}
};

ph meld(ph p, ph q) {
  if (!p || !q) return p?:q;
  if (p->v > q->v) swap(p,q);
  ph P = new heap(*p); P->r = meld(P->r,q);
  if (!P->l || P->l->s < P->r->s) swap(P->l,P->r);
  P->s = (P->r?P->r->s:0)+1; return P;
}
ph ins(ph p, T v) { return meld(p, new heap(v)); }
ph pop(ph p) { return meld(p->l,p->r); }
```

Numerical (4)

4.1 Polynomials and recurrences

Polynomial.h

e9feba, 17 lines

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

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" f1a961, 23 lines

```
vt<db> poly_roots(Poly p, double xmin, double xmax) {
  if (size(p.a) == 2) { return {-p.a[0] / p.a[1]}; }
  vt<db> ret;
  Poly der = p;
  der.diff();
  auto dr = poly_roots(der, xmin, xmax);
  dr.push_back(xmin - 1);
  dr.push_back(xmax + 1);
  sort(all(dr));
  F0R (i, size(dr) - 1) {
    db l = dr[i], h = dr[i + 1];
    bool sign = p(l) > 0;
    if (sign ^ (p(h) > 0)) {
      F0R (it, 60) { // while (h - l > 1e-8)
        double m = (l + h) / 2, f = p(m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.push_back((l + h) / 2);
    }
  }
  return ret;
}
```

PolyInterpolate.h

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

cc9fb9, 13 lines

```
using vd = vt<db>;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  F0R (k, n - 1) FOR (i, k + 1, n)
    y[i] = (y[i] - y[k]) / (x[i] - x[k]);
  double last = 0; temp[0] = 1;
  F0R (k, n) F0R (i, n) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}
```

BerlekampMassey.h

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

"../number-theory/ModPow.h" ce4d42, 20 lines

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

  ll b = 1;
  F0R (i, n) { ++m;
    ll d = s[i] % mod;
    FOR (j, 1, L + 1) d = (d + C[j] * s[i - j]) % mod;
    if (!d) continue;
    T = C; ll coef = d * mpow(b, mod - 2) % mod;
```

```
    FOR (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 $S[0 \ldots \geq n-1]$ and $tr[0 \ldots n-1]$. Faster than matrix multiplication. Useful together with Berlekamp–Massey.
Usage: `linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number`
Time: $\mathcal{O}\left(n^2 \log k\right)$

602ede, 26 lines

```
using Poly = vt<ll>;
ll linearRec(Poly S, Poly tr, ll k) {
  int n = size(tr);

  auto combine = [&](Poly a, Poly b) {
    Poly res(n * 2 + 1);
    F0R (i, n + 1) F0R (j, n + 1)
      res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
    for (int i = 2 * n; i > n; --i) F0R (j, 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;
  F0R (i, 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: `db func(db x) { return 4+x+.3*x*x; }`
`db xmin = gss(-1000,1000,func);`
Time: $\mathcal{O}\left(\log((b-a)/\epsilon)\right)$

dad647, 14 lines

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

HillClimbing.h

Description: Poor man's optimization for unimodal functions.

94e3f2, 14 lines

```
using P = arrray<db, 2>;

template<class F> pair<db, P> hillClimb(P start, F f) {
  pair<db, P> cur(f(start), start);
  for (db jmp = 1e9; jmp > 1e-20; jmp /= 2) {
    F0R (j, 100) FOR (dx, -1, 2) F0R (dy, -1, 2) {
      P p = cur.second;
      p[0] += dx * jmp;
      p[1] += dy * jmp;
      cur = min(cur, make_pair(f(p), p));
    }
  }
  return cur;
}
```

Integrate.h

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

4756fc, 7 lines

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

IntegrateAdaptive.h

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

92dd79, 15 lines

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

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

Simplex.h

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

4ae05c, 60 lines

```
#define mp make_pair
using T = db;
using vd = vt<db>;
using vvd = vt<vd>;
const db eps = 1e-9;
const int inf = 1e9;

#define ltj(X) if (s == -1 || mp(X[j], N[j]) < mp(X[s], N[s])) s = j
struct LPSolver {
  int m, n;
  vt<int> N, B;
  vvd D;
  LPSolver(const vvd& A, const vd& b, const vd& c) :
    m(size(b)), n(size(c)), N(n + 1), B(m), D(m + 2, vd(n + 2)) {
    F0R(i, m) F0R(j, n) D[i][j] = A[i][j];
    F0R(i, m) B[i] = n+i, D[i][n] = -1, D[i][n + 1] = b[i];
    F0R(j, n) N[j] = j, D[m][j] = -c[j];
    N[n] = -1; D[m + 1][n] = 1;
  }
  void pivot(int r, int s) {
    T inv = 1 / D[r][s];
    F0R(i, m + 2) if (i != r && abs(D[i][s]) > eps) {
      T binv = D[i][s]*inv;
      F0R (j, n + 2) if (j != s) D[i][j] -= D[r][j]*binv;
      D[i][s] = -binv;
    }
    D[r][s] = 1; F0R(j, n + 2) D[r][j] *= inv; // scale r-th row
    swap(B[r],N[s]);
  }
  bool simplex(int phase) {
    int x = m + phase - 1;
    while (1) {
      int s = -1; F0R (j, n + 1) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return 1;
      int r = -1;
      F0R (i, 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 0;
      pivot(r, s);
    }
  }
  T solve(vd &x) {
    int r = 0; FOR (i, 1, m) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -eps) {
      pivot(r,n);
      assert(simplex(2));
      if (D[m + 1][n + 1] < -eps) return -inf;
      F0R (i, m) if (B[i] == -1) {
        int s = 0; FOR (j, 1, n + 1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    F0R (i, m) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return ok ? D[m][n + 1] : inf;
  }
};
```

4.3 Matrices

Determinant.h

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

ef2d92, 15 lines

```
db det(vt<vt<db>>& a) {
  int n = size(a); db res = 1;
  F0R (i, n) {
    int b = i;
    FOR (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;
    FOR (j, i + 1, n) {
      double v = a[j][i] / a[i][i];
      if (v != 0) FOR (k, i + 1, n) a[j][k] -= v * a[i][k];
    }
  }
  return res;
}
```

IntDeterminant.h

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

3313dc, 18 lines

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

SolveLinear.h

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

0ff65f, 38 lines

```
using vd = vt<db>;
const double eps = 1e-12;

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

  F0R (i, n) {
    double v, bv = 0;
    FOR (r, i, n) FOR (c, i, m)
      if ((v = fabs(A[r][c])) > bv)
        br = r, bc = c, bv = v;
    if (bv <= eps) {
      FOR (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]);
    F0R (j, n) swap(A[j][i], A[j][bc]);
    bv = 1 / A[i][i];
    FOR (j, i + 1, n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      FOR (k, i + 1, m) A[j][k] -= fac * A[i][k];
    }
    rank++;
  }

  x.assign(m, 0);
  for (int i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    F0R (j, 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" 77b9bb, 7 lines

```
F0R (j, n) if (j != i) // instead of FOR (j, i + 1, n)
// ... then at the end:
x.assign(m, undefined);
F0R (i, rank) {
  FOR (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: $\mathcal{O}\left(n^2m\right)$

042fea, 34 lines

```
typedef bitset<1000> bs;

int solveLinear(vt<bs>& A, vi& b, bs& x, int m) {
  int n = size(A), rank = 0, br;
  assert(m <= size(x));
  vi col(m); iota(all(col), 0);
  F0R (i, n) {
    FOR (br, i, n) if (A[br].any()) break;
    if (br == n) {
      F0R (j, i, n) if(b[j]) return -1;
      break;
    }
    int bc = (int) A[br]._Find_next(i - 1);
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    F0R (j, n) if (A[j][i] != A[j][bc]) {
      A[j].flip(i); A[j].flip(bc);
    }
    FOR (j, i + 1, n) if (A[j][i]) {
      b[j] ^= b[i];
      A[j] ^= A[i];
    }
    rank++;
  }

  x = bs();
  for (int i = rank; i--;) {
    if (!b[i]) continue;
    x[col[i]] = 1;
    F0R (j, i) b[j] ^= A[j][i];
  }
  return rank; // (multiple solutions if rank < m)
}
```

MatrixInverse.h

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

cb738d, 35 lines

```
int matInv(vt<vt<db>>& A) {
  int n = size(A); vi col(n);
  vt<vt<db>> tmp(n, vt<db>(n));
  F0R (i, n) tmp[i][i] = 1, col[i] = i;

  F0R (i, n) {
    int r = i, c = i;
    FOR (j, i, n) FOR (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]);
    F0R (j, 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];
    FOR (j, i + 1, n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      FOR (k, i + 1, n) A[j][k] -= f * A[i][k];
      FOR (k, 0, n) tmp[j][k] -= f * tmp[i][k];
    }
    FOR (j, i + 1, n) A[i][j] /= v;
    F0R (j, n) tmp[i][j] /= v;
    A[i][i] = 1;
  }

  for (int i = n - 1; i > 0; --i) F0R (j, i) {
    double v = A[j][i];
    F0R (k, n) tmp[j][k] -= v * tmp[i][k];
  }

  F0R (i, n) F0R (j, n) A[col[i]][col[j]] = tmp[i][j];
  return n;
}
```

MatrixInverse-mod.h

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

"../number-theory/ModPow.h" 0b7b13, 37 lines

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

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

```
    A[i][i] = 1;
  }

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

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

Tridiagonal.h

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

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

This is useful for solving problems on the type

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

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

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

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

8f9fa8, 26 lines

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

4.4 Fourier transforms

FastFourierTransform.h

Description: fft(a) computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. N must be a power of 2. Useful for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, divide by n, reverse(start+1, end), FFT back. Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs). Otherwise, use NTT/FFTMod.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 0.2s$ for $N = 2^{20}$)

636a47, 36 lines

```
using C = complex<db>;
using vd = vt<db>;

void fft(vt<C> &a) {
  int n = size(a), L = 31 - __builtin_clz(n);
  static vt<complex<long double>> R(2, 1);
  static vt<C> rt(2, 1);  // (^ 10% faster if db)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    FOR (i, k, 2 * k) rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
  }
  vi rev(n);
  F0R (i, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  F0R (i, n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) F0R (j, 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(size(a) + size(b) - 1);
  int L = 32 - __builtin_clz(size(res)), n = 1 << L;
  vt<C> in(n), out(n);
  copy(all(a), begin(in));
  F0R (i, size(b)) in[i].imag(b[i]);
  fft(in);
  for (C& x : in) x *= x;
  F0R (i, n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  F0R (i, size(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) (but seemed +10% on yosupo?)

"FastFourierTransform.h" c4a07e, 22 lines

```
using vl = vt<ll>;
template<int M> vl convMod(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  vl res(size(a) + size(b) - 1);
  int B = 32 - __builtin_clz(size(res)), n = 1 << B, cut = int(sqrt(M)
      );
  vt<C> L(n), R(n), outs(n), outl(n);
  F0R (i, size(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
  F0R (i, size(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
  fft(L), fft(R);
  F0R (i, n) {
    int j = -i & (n - 1);
    outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
    outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
  }
  fft(outl), fft(outs);
  F0R (i, size(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: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 0.2s$ for $N = 2^{20}$)

"../number-theory/ModPow.h" 53740f, 37 lines

```
const ll 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.

template<class T>
void ntt(vt<T> &a) {
  int n = size(a), L = 31 - __builtin_clz(n);
  static vt<ll> rt(2, 1);
  for (static int k = 2, s = 2; k < n; k *= 2, s++) {
    rt.resize(n);
    ll z[] = {1, mpow(root, mod >> s)};
    FOR (i, k, 2 * k) rt[i] = rt[i / 2] * z[i & 1] % mod;
  }
  vi rev(n);
  F0R (i, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  F0R (i, n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) FOR (j, 0, k) {
      T z = (ll) 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);
    }
}

template<class T>
vt<T> conv(const vt<T> &a, const vt<T> &b) {
  if (a.empty() || b.empty()) return {};
  int s = size(a) + size(b) - 1, B = 32 - __builtin_clz(s),
    n = 1 << B;
  int inv = mpow(n, mod - 2);
  vt<T> L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  F0R (i, 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. Replace with long longs and do operations under mod if needed.
Time: $\mathcal{O}(N \log N)$

97380b, 17 lines

```
using pi = pair<int, int>;
void FST(vi &a, bool inv) {
  for (int n = size(a), step = 1; step < n; step *= 2) {
    for (int i = 0; i < n; i += 2 * step) FOR (j, i, i + step) {
      int &u = a[j], &v = a[j + step]; tie(u, v) =
        inv ? pi(v - u, u) : pi(v, u + v); // AND
```

```
      inv ? pi(v, u - v) : pi(u + v, u); // OR
      pi(u + v, u - v);          // XOR
    }
  }
  if (inv) for (int& x : a) x /= size(a); // XOR only
}
vi conv(vi a, vi b) {
  FST(a, 0); FST(b, 0);
  F0R (i, size(a)) a[i] *= b[i];
  FST(a, 1); return a;
}
```

Number theory (5)

5.1 Modular arithmetic

ModularArithmetic.h

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

"euclid.h" 35bfea, 18 lines

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

ModInverse.h

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

e403d4, 3 lines

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

ModPow.h

d63292, 8 lines

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

ll mpow(ll b, ll e) {
  ll ans = 1;
  for (; e; b = b * b % mod, e /= 2)
    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: $\mathcal{O}\left(\sqrt{m}\right)$

c040b8, 11 lines

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

ModSum.h

Description: Sums of mod'ed arithmetic progressions.

modsum(to, c, k, m) $= \sum_{i=0}^{to-1} (ki + c)\%m$. 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

bb4d60, 11 lines

```
typedef unsigned long long ull;
ull mmul(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 mpow(ull b, ull e, ull mod) {
  ull ans = 1;
  for (; e; b = mmul(b, b, mod), e /= 2)
    if (e & 1) ans = mmul(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" 4ba904, 24 lines

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

5.2 Primality

Sieve.h

Description: Linear sieve implementation. Around 0.5s for 1e8. Also computes smallest prime divisor for each number in lp.

6f6405, 22 lines

```
const int mx = 1e8; // ~0.5s
vi lp(mx + 1), primes;

FOR (i, 2, mx + 1) {
  if (lp[i] == 0) lp[i] = i, primes.pb(i);
  for (int j = 0; i * primes[j] <= mx; j++) {
    lp[i * primes[j]] = primes[j];
    // store i to avoid division when factoring
    if (primes[j] == lp[i]) break;
  }
}

auto factor = [&] (int x) {
  vt<pair<int, int>> res;
  while (x > 1) {
    if (!size(res) || res.back().first != lp[x])
      res.pb({lp[x], 0});
    res.back().second++;
    x /= lp[x]; // can avoid division by storing extra array
  }
  return res;
};
```

FactorSqrt.h

Description: Easy factorisation. Returns pairs of prime factor, multiplicity

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

56249c, 15 lines

```
vt<pair<int, int>> factor(int x) {
  if (x == 1) return {};
  vt<pair<int, int>> res;
  for (int i = 2; i * i <= x; i++) {
    if (x % i == 0) {
      res.pb({i, 0});
      while (x % i == 0) {
        res.back().second++;
        x /= i;
      }
    }
  }
  if (x > 1) res.pb({x, 1});
  return res;
}
```

FastEratosthenes.h

Description: Prime sieve for generating all primes smaller than LIM.

Time: LIM=1e9 $\approx$ 1.5s

6b2912, 20 lines

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

```
  for (int L = 1; L <= R; L += S) {
    array<bool, S> block{};
    for (auto &[p, idx] : cp)
      for (int i=idx; i < S+L; idx = (i+=p)) block[i-L] = 1;
    rep(i,0,min(S, R - L))
      if (!block[i]) pr.push_back((L + i) * 2 + 1);
  }
  for (int i : pr) isPrime[i] = 1;
  return pr;
}
```

MillerRabin.h

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger numbers, use Python and extend A randomly.

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

"ModMulLL.h" 168797, 12 lines

```
bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
  ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
    s = __builtin_ctzll(n - 1), d = n >> s;
  for (ull a : A) {   // ^ count trailing zeroes
    ull p = mpow(a % n, d, n), i = s;
    while (p != 1 && p != n - 1 && a % n && i--)
      p = mmul(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}). Consider manually trying small primes for better runtime. In one second: average 350k numbers at 1e9, 200k numbers at 1e12, and 100k at 1e16.

Time: $\mathcal{O}\left(n^{1/4}\right)$, less for numbers with small factors.

"ModMulLL.h", "MillerRabin.h" bf9dbb, 18 lines

```
ull pollard(ull n) {
  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  auto f = [&](ull x) { return mmul(x, x, n) + i; };
  while (t++ % 40 || __gcd(prd, n) == 1) {
    if (x == y) x = ++i, y = f(x);
    if ((q = mmul(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;
}
```

PrimeCnt.h

Description: Counts number of primes up to N. Can also count sum of primes.

Time: $\mathcal{O}\left(N^{3/4}/\log N\right)$, 60ms for $N = 10^{11}$, 2.5s for $N = 10^{13}$

c04e96, 20 lines

```
ll count_primes(ll N) { // count_primes(1e13) == 346065536839
  if (N <= 1) return 0;
  int sq = (int)sqrt(N);
  vl big_ans((sq+1)/2), small_ans(sq+1);
  FOR(i,1,sq+1) small_ans[i] = (i-1)/2;
  F0R(i,sz(big_ans)) big_ans[i] = (N/(2*i+1)-1)/2;
  vb skip(sq+1); int prime_cnt = 0;
  for (int p = 3; p <= sq; p += 2) if (!skip[p]) { // primes
    for (int j = p; j <= sq; j += 2*p) skip[j] = 1;
    F0R(j,min((ll)sz(big_ans),(N/p/p+1)/2)) {
      ll prod = (ll)(2*j+1)*p;
      big_ans[j] -= (prod > sq ? small_ans[(double)N/prod]
            : big_ans[prod/2])-prime_cnt;
    }
    for (int j = sq, q = sq/p; q >= p; --q) for (;j >= q*p;--j)
      small_ans[j] -= small_ans[q]-prime_cnt;
    ++prime_cnt;
  }
  return big_ans[0]+1;
}
```

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;
}
```

Euclid.java

Description: Finds {x, y, d} s.t. ax + by = d = gcd(a, b).

6aba01, 11 lines

```
static BigInteger[] euclid(BigInteger a, BigInteger b) {
  BigInteger x = BigInteger.ONE, yy = x;
  BigInteger y = BigInteger.ZERO, xx = y;
  while (b.signum() != 0) {
    BigInteger q = a.divide(b), t = b;
    b = a.mod(b); a = t;
    t = xx; xx = x.subtract(q.multiply(xx)); x = t;
    t = yy; yy = y.subtract(q.multiply(yy)); y = t;
  }
  return new BigInteger[]{x, y, 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 thm: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod n$.

Fermat's little thm: p prime $\Rightarrow a^{p-1} \equiv 1 \pmod p \ \forall a$.

5bf43c, 8 lines

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

void calculatePhi() {
  F0R (i, LIM) phi[i] = i & 1 ? i : i / 2;
  for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
    for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

5.4 Fractions

5.4.1 Continued Fractions

Let $a_0, a_1, \ldots, a_n \in \mathbb{Z}$ and $a_1, a_2, \ldots, a_n \ge 1$. Then the expression

$$r = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}},$$

is called the *continued fraction representation* of the rational number r and is denoted shortly as

$$r = [a_0; a_1, a_2, \ldots, a_k].$$

If $a_0 > 0$ then $\frac{1}{r}$ is given by sticking a 0 in front. If $a_0 = 0$ then ditch the first term.
$[a_0, a_1, \ldots, a_n - 1]$ gives the run-length encoding (right first) of r's path in the Stern-Brocot tree.
$r_k = [a_0; a_1, a_2, \ldots, a_k] = \frac{p_k}{q_k}$ is called the k-th *convergent* of r.
Semiconvergents are convergents but you can lower the last number down to 1.
Consider the convex hull of points above and below $y = rx$. Odd convergents $(q_k; p_k)$ give vertices of the upper hull, while even convergents give the vertices of the lower hull.
The set of semiconvergents = set of lattice points on the hulls.
Combine these facts with Pick's theorem for funny stuff.

SBT.h

Description: Given p/q, find the shorter of its two unique continued fraction representations. The other representation is given by vec.back()-, vec.pb(1);

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

fb1c2d, 8 lines

```
vi cont_frac(int p, int q) {
  vi a;
  while (q) {
    a.push_back(p / q);
    tie(p, q) = make_pair(q, p % q);
  }
  return a;
}
```

ContinuedFractions.h

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

For consecutive convergents, $p_{k+1}q_k - q_{k+1}p_k = (-1)^k$. ($p_k/q_k$ alternates between $> x$ and $< x$.) If x is rational, y eventually becomes ∞; if x is the root of a degree 2 polynomial the a's eventually become cyclic.

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

dd6c5e, 21 lines

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

FracBinarySearch.h

Description: Given f and N, finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p, q \leq N$. You may want to throw an exception from f if it finds an exact solution, in which case N can be removed.

Usage: `fracBS([](Frac f) { return f.p>=3*f.q; }, 10); // {1,3}`

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

27ab3e, 25 lines

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

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

5.5 Pythagorean Triples

The Pythagorean triples are uniquely generated by

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

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

5.6 Primes

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

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

5.7 Estimates

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

The number of divisors of n is at most around 100 for $n < 5e4$, 250 for $n < 1e6$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

5.8 Mobius Function

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

Mobius Inversion:

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

Other useful formulas/forms:

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

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

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

Combinatorial (6)

6.1 Permutations

6.1.1 Factorial

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

IntPerm.h

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

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

044568, 6 lines

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

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

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

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 $n = n_k p^k + \ldots + n_1 p + n_0$ and $m = m_k p^k + \ldots + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod{p}$.

6.2.3 Binomials

multinomial.h

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

a0a312, 5 lines

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

6.3 General purpose numbers

6.3.1 Bernoulli numbers

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

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.

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

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

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. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$. For p prime,

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

6.3.6 Labeled unrooted trees

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

6.3.7 Catalan numbers

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

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

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

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

Graph (7)

7.0.1 Theorems

A matching M is maximum **iff** there is no M-augmenting path.

7.0.2 Hall's Marriage Theorem

$G = (L, R; E)$ has a matching covering all of L iff $|N(s)| \geq |S|$ for every $S \subseteq L$.
Corollary: Let $\delta = \max_{S \subseteq L}\left(|S| - |N(S)|\right)$. The size of the maximum matching is $\tau = |L| - \delta$.

7.0.3 Kőnig's Theorem

In bipartite graphs: **max matching = min vertex cover**, i.e. $\nu(G) = \tau(G)$. Hence max independent set size $\alpha(G) = |V| - \tau(G) = |V| - \nu(G)$.
To recover min vertex cover: From all unmatched vertices in L, BFS alternating paths (use non-matching edges $L \to R$; matching edges $R \to L$). Let Z be reachable vertices. Min vertex cover is $(L \setminus Z) \cup (R \cap Z)$.

7.0.4 Dilworth's Theorem (posets)

Run Floyd-Warshall for transitivity.
Width (max antichain size) = min number of chains in a partition.
Reduction: make bipartite graph with two copies of elements; edge $x_L - y_R$ if $x < y$. Then min chain decomposition size $= |P| - \nu$; width $= |P| - (|P| - \nu) = \nu$.

Mirsky's Theorem (dual): Height (max chain size) = min number of antichains in a partition.

7.0.5 Tutte's 1-Factor Theorem

$odd(G)$ is the number of components in G with an odd number of vertices. G has a perfect matching iff for all $S \subseteq V$, $\text{odd}(G-S) \leq |S|$. Tutte–Berge formula: $\nu(G) = \frac{1}{2}\min_{S \subseteq V}\left(|V| - \text{odd}(G-S) + |S|\right)$.

7.0.6 Edge cover

In any graph without isolated vertices, the size of the minimum edge cover (set of edges that touch every vertex) is given by:

$$\rho'(G) = |V| - \nu(G)$$

.

7.1 Fundamentals

BellmanFord.h

Description: Calculates shortest paths from s in a graph that might have negative edge weights. Unreachable nodes get dist = inf; nodes reachable through negative-weight cycles get dist = -inf. Assumes $V^2 \max|w_i| < \sim 2^{63}$.
Time: $\mathcal{O}(VE)$ f021ee, 23 lines

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

void bellmanFord(vt<Node> &nodes, vt<Ed> &eds, int s) {
  nodes[s].dist = 0;
  sort(all(eds), [] (Ed a, Ed b) { return a.s() < b.s(); });

  int lim = size(nodes) / 2 + 2; // /3 + 100 with shuffled vertices
  F0R (i, lim) for (Ed ed : eds) {
    Node cur = nodes[ed.a], &dest = nodes[ed.b];
    if (abs(cur.dist) == inf) continue;
    ll d = cur.dist + ed.w;
    if (d < dest.dist) {
      dest.prev = ed.a;
      dest.dist = (i < lim-1 ? d : -inf);
    }
  }
  F0R (i, lim) for (Ed e : eds) {
    if (nodes[e.a].dist == -inf)
      nodes[e.b].dist = -inf;
  }
}
```

DEsopoPape.h

Description: Mystery shortest path algorithm with exponential breaking cases. Works with negative weights, but dies on negative cycles.
Time: Linear to exponential, depending on how good data is. 24b4d7, 24 lines

```
const ll INF = 1e18;
vt<vt<pair<int, ll>>> adj;

void shortest_paths(int s, vt<ll> &d, vi &p) {
  d.resize(n, INF);
  d[s] = 0;
  vi m(n, 2);
  deque<int> q {s};
  p.resize(n, -1);

  while (!q.empty()) {
    int u = q.front(); q.pop_front();
```

```
    m[u] = 0;
    for (auto [v, w] : adj[u]) {
      if (d[v] > d[u] + w) {
        d[v] = d[u] + w;
        p[v] = u;
        if (m[v] == 2) q.pb(v);
        else if (m[v] == 0) q.push_front(v);
        m[v] = 1;
      }
    }
  }
}
```

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

35be6b, 10 lines

```
const ll inf = 1ll << 62;
void floydWarshall(vt<vt<ll>>& m) {
  int n = size(m);
  F0R (i, n) m[i][i] = min(m[i][i], 0LL);
  F0R (k, n) F0R (i, n) F0R (j, n)
    if (m[i][k] != inf && m[k][j] != inf)
      m[i][j] = min(m[i][j], max(m[i][k] + m[k][j], -inf));
  F0R (k, n) if (m[k][k] < 0) F0R (i, n) F0R (j, n)
    if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

DirectedCycle.h

Description: stack

81fac0, 17 lines

```
template<int sz> struct DirCyc {
  vi adj[sz], stk, cyc; vt<bool> in_stk, vis;
  void dfs(int x) {
    stk.pb(x); in_stk[x] = vis[x] = 1;
    for (int i : adj[x]) {
      if (in_stk[i]) cyc = {find(all(stk),i), end(stk)};
      else if (!vis[i]) dfs(i);
      if (size(cyc)) return;
    }
    stk.pop_back(); in_stk[x] = 0;
  }
  vi init(int n) {
    in_stk.resize(n), vis.resize(n);
    F0R (i, n) if (!vis[i] && !size(cyc)) dfs(i);
    return cyc;
  }
};
```

7.2 Network flow

FordFulkerson.h

Description: Short algo for computing maximum flows with a bounded answer.

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

86c231, 16 lines

```
const int mx = 2000;
int seen[mx], tim;
unordered_map<int, int> adj[mx];

int flow(int s, int t) {
  auto dfs = [&] (auto &&self, int u) {
    if (u == t) return 1;
    if (exchange(seen[u], tim) == tim) return 0;
    for (auto &[v, c] : adj[u])
      if (c && self(self, v)) return --adj[u][v], ++adj[v][u];
    return 0;
  };
  int flow = 0;
  while (tim++, dfs(dfs, s)) flow++;
  return flow;
}
```

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.

5505a1, 32 lines

```
template<class T> T edmondsKarp(vector<unordered_map<int, T>>&
    adj, int s, int t) {
  assert(s != t);
  T flow = 0;
  vt<int> par(size(adj)), q = par;
  while (1) {
    fill(all(par), -1);
    int ptr = 1;
    q[0] = s, par[s] = 0;
    F0R (i, ptr) {
      int u = q[i];
      for (auto &[v, c] : adj[u]) {
        if (par[v] == -1 && c) {
          par[v] = u, q[ptr++] = v;
          if (v == t) goto out;
        }
      }
    }
    return flow;
  out:
    T inc = numeric_limits<T>::max();
    for (int y = t; y != s; y = par[y])
      inc = min(inc, adj[par[y]][y]);

    flow += inc;
    for (int y = t; y != s; y = par[y]) {
      int p = par[y];
      if ((adj[p][y] -= inc) <= 0) adj[p].erase(y);
      adj[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.

8046dd, 47 lines

```
struct Dinic {
  struct Edge {
    int to, rev;
    ll c, oc;
    // ll flow() { return max(oc - c, 0LL); } // if you need flows
  };
  vi lvl, ptr, q;
  vt<vt<Edge>> adj;

  void init(int n) {
    lvl = ptr = q = vi(n);
    adj.resize(n);
  }

  void ae(int a, int b, ll c, ll rcap = 0) {
    adj[a].pb({b, size(adj[b]), c, c});
    adj[b].pb({a, size(adj[a]) - 1, rcap, rcap});
  }
  ll dfs(int v, int t, ll f) {
    if (v == t || !f) return f;
    for (int& i = ptr[v]; i < size(adj[v]); i++) {
      auto &[to, rev, c, _] = adj[v][i];
      if (lvl[to] == lvl[v] + 1)
        if (ll p = dfs(to, t, min(f, c))) {
          c -= p, adj[to][rev].c += p;
          return p;
        }
    }
    return 0;
  }
  ll calc(int s, int t) {
    ll flow = 0; q[0] = s;
    F0R (L, 31) do { // 'int L = 30' maybe faster for random data
      lvl = ptr = vi(size(q));
      int qi = 0, qe = lvl[s] = 1;
      while (qi < qe && !lvl[t]) {
        int v = q[qi++];
        for (Edge e : adj[v])
          if (!lvl[e.to] && e.c >> (30 - L))
            q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
      }
      while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
    } while (lvl[t]);
    return flow;
  }
  // bool left_of_min_cut(int a) { return lvl[a] != 0; }
};
```

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: $\mathcal{O}\left(V^2\sqrt{E}\right)$

14ba90, 66 lines

```
template<typename flow_t = long long>
struct PushRelabel {
  struct Edge {
    int to, rev;
    flow_t f, c;
  };
  vt<vt<Edge>> g;
  vt<flow_t> ec;
  vt<Edge*> cur;
  vt<vt<int>> hs;
  vt<int> h;

  void init(int n) {
    g.resize(n);
    ec.resize(n);
    cur.resize(n);
    hs.resize(2 * n);
    h.resize(n);
  }

  void ae(int s, int t, flow_t cap, flow_t rcap = 0) {
    if (s == t) return;
    g[s].push_back({t, size(g[t]), 0, cap});
    g[t].push_back({s, size(g[s]) - 1, 0, rcap});
  }
  void add_flow(Edge& e, flow_t f) {
    Edge &back = g[e.to][e.rev];
    if (!ec[e.to] && f)
      hs[h[e.to]].push_back(e.to);
    e.f += f; e.c -= f;
    ec[e.to] += f;
    back.f -= f; back.c += f;
    ec[back.to] -= f;
  }
```

```
  flow_t calc(int s, int t) {
    int v = size(g);
    h[s] = v;
    ec[t] = 1;
    vt<int> co(2 * v);
    co[0] = v - 1;
    F0R (i, v) cur[i] = g[i].data();
    for(auto &e : g[s]) add_flow(e, e.c);
    if (size(hs[0]))
    for (int hi = 0; hi >= 0;) {
      int u = hs[hi].back();
      hs[hi].pop_back();
      while (ec[u] > 0) // discharge u
        if (cur[u] == g[u].data() + size(g[u])) {
          h[u] = 1e9;
          for (auto &e : g[u])
            if (e.c && h[u] > h[e.to] + 1)
              h[u] = h[e.to] + 1, cur[u] = &e;
          if (++co[h[u]], !--co[hi] && hi < v)
            F0R (i, 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]->to] + 1)
          add_flow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
      while (hi >= 0 && hs[hi].empty()) --hi;
    }
    return -ec[s];
  }
  // bool leftOfMinCut(int a) { return h[a] >= size(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. bbe6a2, 79 lines

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

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

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

  void ae(int from, int to, ll cap, ll cost) {
    if (from == to) return;
    ed[from].push_back(edge{ from, to, size(ed[to]),
      cap,cost,  0 });
    ed[to].push_back(edge{ to,  from, size(ed[from]) - 1,
      0, -cost, 0 });
  }

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

    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vt<decltype(q)::point_iterator> its(N);
    q.push({ 0, s });

    while (!q.empty()) {
      s = q.top().second; 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 });
        }
      }
    }
    F0R (i, N) pi[i] = min(pi[i] + dist[i], INF);
  }

  pair<ll, ll> maxflow(int s, int t) {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = INF;
      for (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;
      }
    }
    FOR (i, 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(int s) { // (otherwise, leave this out)
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--)
      FOR (i, 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
  }
};
```

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: $\mathcal{O}(V^3)$ d4d36f, 21 lines

```
pair<int, vt<int>> globalMinCut(vt<vt<int>> mat) {
  pair<int, vt<int>> best = {INT_MAX, {}};
  int n = size(mat);
  vt<vt<int>> co(n);
  F0R (i, n) co[i] = {i};
  FOR(ph, 1, n) {
    vt<int> w = mat[0];
    size_t s = 0, t = 0;
    F0R (it, n - ph) { // O(V^2) -> O(E log V) with prio. queue
      w[t] = INT_MIN;
      s = t, t = max_element(all(w)) - w.begin();
      F0R (i, 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]));
    F0R (i, n) mat[s][i] += mat[t][i];
    F0R (i, n) mat[i][s] = mat[s][i];
    mat[0][t] = INT_MIN;
  }
  return best;
}
```

NetworkSimplex.h

Description: Solves minimum cost circulation problem. To convert to a "normal" mcmf, add an edge from t -> s of big capacity and negative inf cost (remember to add this back on to answer!). If you don't necessarily need to maximise flow, add free edge from s -> t. Edge i (one indexed) is ns.edges[2 * i]. Works with negative cost cycles.
Time: $\mathcal{O}(VE)$ on average maybe bde2f1, 93 lines

```
struct NetworkSimplex {
  using Flow = int;
  using Cost = ll;
  struct Edge {
    int nxt, to;
    Flow cap;
    Cost cost;
  };
  vector<Edge> edges;
  vector<int> head, fa, fe, mark, cyc;
  vector<Cost> dual;
  int ti;

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

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

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

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

  void push_flow(int e, Cost &cost) {
```

```
    int pen = edges[e ^ 1].to, lca = edges[e].to;
    ti++;
    while (pen) mark[pen] = ti, pen = fa[pen];
    while (mark[lca] != ti) mark[lca] = ti, lca = fa[lca];

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

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

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

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

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" 0418b3, 13 lines

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

7.3 Matching

7.3.1 Reductions

Min cut reduction

Consider min cut as an instance of this problem: Given $A[N]$ and $B[N]$ (any sign) and $C[N][N]$ (non-negative), find a binary string S of length N that minimises the following:

- Pay $A[i]$ if $S[i] = 1$
- Pay $B[i]$ if $S[i] = 0$
- Pay $C[i][j]$ if $S[i] = 1$ and $S[j] = 0$.

To calculate the answer:

- Add edges i to t of capacity $B[i] + \texttt{inf}$.
- Add edges s to i of capacity $A[i] + \texttt{inf}$.
- Add edges i to j of capcity $C[i][j]$.
- The answer is the maximum flow in this graph minus $n \times \texttt{inf}$.

7.3.2 Circulation

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: $\mathcal{O}(VE)$

c2ed08, 22 lines

```
bool find(int j, vt<vi> &g, vi &btoa, vi &vis) {
  if (btoa[j] == -1) return 1;
  vis[j] = 1; int di = btoa[j];
  for (int e : g[di])
    if (!vis[e] && find(e, g, btoa, vis)) {
      btoa[e] = di;
      return 1;
    }
  return 0;
}
int dfsMatching(vt<vi> &g, vi &btoa) {
  vi vis;
  F0R (i, size(g)) {
    vis.assign(size(btoa), 0);
    for (int j : g[i])
      if (find(j, g, btoa, vis)) {
        btoa[j] = i;
        break;
      }
  }
  return size(btoa) - (int) 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" 6cc1d8, 20 lines

```
vi cover(vector<vi>& g, int n, int m) {
  vi match(m, -1);
  int res = dfsMatching(g, match);
  vt<bool> lfound(n, true), seen(m);
  for (int it : match) if (it != -1) lfound[it] = false;
  vi q, cover;
  F0R (i, n) if (lfound[i]) q.push_back(i);
  while (!q.empty()) {
    int i = q.back(); q.pop_back();
    lfound[i] = 1;
    for (int e : g[i]) if (!seen[e] && match[e] != -1) {
      seen[e] = true;
      q.pb(match[e]);
    }
  }
  F0R (i, n) if (!lfound[i]) cover.push_back(i);
  F0R (i, m) if (seen[i]) cover.push_back(n+i);
  assert(size(cover) == res);
  return cover;
}
```

hopcroftKarp.h

Description: Fast incremental bipartite matching. Zero-indexed.
Usage: `{operator[]} for the pair of right node i, {n} is the size of the rhs, {add(v)} to add adjacency list of node on lhs`
Time: $\mathcal{O}\left(\sqrt{V}E\right)$

ad03a4, 39 lines

```
struct Matching : vi {
  vt<vi> adj;
  vi rank, low, pos, vis, seen;
  int k{0};
  // n = size of rhs
  Matching(int n) :vi(n, -1), rank(n) {}
  bool add(vi vec) {
    adj.pb(std::move(vec));
    low.pb(0), pos.pb(0), vis.pb(0);
    if (size(adj.back())) {
      int i = k;
    nxt:
      seen.clear();
      if (dfs(size(adj)-1, ++k-i)) return 1;
      for (auto &v : seen) for (auto &e : adj[v])
        if (rank[e] < inf && vis[at(e)] < k)
          goto nxt;
      for (auto &v : seen) {
        low[v] = inf;
        for (auto &w : adj[v]) rank[w] = inf;
      }
    }
    return 0;
  }
  bool dfs(int v, int g) {
    if (vis[v] < k) vis[v] = k, seen.pb(v);
    while (low[v] < g) {
      int e = adj[v][pos[v]];
      if (at(e) != v && low[v] == rank[e]) {
        rank[e]++;
        if (at(e) == -1 || dfs(at(e), rank[e]))
          return at(e) = v, 1;
      } else if (++pos[v] == size(adj[v])) {
        pos[v] = 0; low[v]++;
      }
    }
    return 0;
  }
};
```

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

2e27e7, 31 lines

```
pair<int, vi> hungarian(const vt<vi> &a) {
```

```
  if (a.empty()) return {0, {}};
  int n = size(a) + 1, m = size(a[0]) + 1;
  vi u(n), v(m), p(m), ans(n - 1);
  FOR (i, 1, n) {
    p[0] = i;
    int j0 = 0;
    vi dist(m, INT_MAX), pre(m, -1);
    vector<bool> done(m + 1);
    do {
      done[j0] = true;
      int i0 = p[j0], j1, delta = INT_MAX;
      FOR (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;
      }
      FOR (j, m) {
        if (done[j]) u[p[j]] += delta, v[j] -= delta;
        else dist[j] -= delta;
      }
      j0 = j1;
    } while (p[j0]);
    while (j0) { // update alternating path
      int j1 = pre[j0];
      p[j0] = p[j1], j0 = j1;
    }
  }
  FOR (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" cb1912, 40 lines

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

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

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

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

Blossom.h

Description: Matching for general graphs. 1-indexed!
Time: $\mathcal{O}\left(NM\right)$

0b8afb, 59 lines

```
// 1-indexed!
struct Blossom {
  int n, h, t, cnt;
  vpi edges;
  vi vis, q, mate, col, fa, pre, he;
  void ae(int u, int v) {
    assert(u && v);
    edges.pb({he[u], v}); he[u] = size(edges) - 1;
    edges.pb({he[v], u}); he[v] = size(edges) - 1;
  }
  inline int get(int u) { return fa[u] == u ? u : fa[u] = get(fa[u]);
      }
  void aug(int u, int v) {
    for (int p; u; u = p, v = pre[p])
      p = mate[v], mate[mate[u] = v] = u;
  }
  void init(int _n) {
    n = _n;
    vis = q = mate = col = fa = pre = he = vi(n + 1);
  }
  int lca(int u, int v) {
    for (cnt++;; u = pre[mate[u]]) {
      if (v) swap(u, v);
      if (vis[u = get(u)] == cnt) return u;
      vis[u] = cnt;
    }
  }
  void blo(int u, int v, int f) {
    for (int p; get(u) != f; v = p, u = pre[p]) {
      p = mate[u]; pre[u] = v; fa[u] = fa[p] = f;
      if (col[p] != 1) col[q[++t] = p] = 1;
    }
  }
  bool bfs(int u) {
    FOR (i, 1, n + 1) col[i] = 0, fa[i] = i;
    h = 0; q[t = 1] = u; col[u] = 1;
    while (h != t) {
      int x = q[++h];
      for (int i = he[x]; i; i = edges[i].f) {
        int y = edges[i].s;
        if (!col[y]) {
          if (!mate[y]) { aug(y, x); return 1; }
          pre[y] = x;
          col[y] = 2;
          col[q[++t] = mate[y]] = 1;
        } else if (col[y] == 1 && get(x) != get(y)) {
          int p = lca(x, y);
          blo(x, y, p);
          blo(y, x, p);
        }
      }
    }
    return 0;
  }
  int solve() {
    int ans = 0;
    FOR (i, 1, n + 1) if (!mate[i]) ans += bfs(i);
    return ans;
  }
};
```

7.4 DFS algorithms

SCC.h

Description: Finds strongly connected components in a directed graph. `comps` is top-sorted.
Time: $\mathcal{O}\left(E+V\right)$

ae000b, 30 lines

```
struct SCC {
  int n;
  vt<vt<int>> adj, radj;
  vt<int> todo, seen, comp, comps; // comps is top sorted
  void init(int _n) {
    n = _n;
    adj = radj = vt<vt<int>>(n);
    comp.resize(n, -1);
    seen.resize(n);
  }
  void ae(int u, int v) {
    adj[u].pb(v);
    radj[v].pb(u);
  }
  void dfs(int u) {
    if (seen[u]++) return;
    for (int v : adj[u]) dfs(v);
    todo.pb(u);
  }
  void rdfs(int u, int w) {
    comp[u] = w;
    for (int v : radj[u]) if (comp[v] == -1) rdfs(v, w);
  }
  void gen() {
    F0R (i, n) dfs(i);
    reverse(all(todo));
    for (int u : todo) if (comp[u] == -1)
      rdfs(u, u), comps.pb(u);
  }
};
```

BiconnectedComponents.h

Description: Finds all biconnected components in an undirected graph. 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. Note that degree 0 nodes are not considered components.
Time: $\mathcal{O}\left(E+V\right)$

8f3a23, 46 lines

```
struct BCC {
  int n, m, t;
  vt<vt<pair<int, int>>> adj;
  vector<pair<int, int>> edges;
  vt<vi> comps; // lists of edges of bcc
  vi tin, stk, is_art, is_bridge; // stk for bcc only

  void init(int _n, vt<pair<int, int>> &_edges) {
    n = _n;
    edges = _edges;
    m = size(edges);
    adj.resize(n);
    F0R (i, m) {
      auto [u, v] = edges[i];
      adj[u].pb({v, i});
      adj[v].pb({u, i});
    }
    t = 0;
    tin = is_art = vt<int>(n);
    is_bridge.resize(m);
    F0R (u, n) if (!tin[u]) dfs(u, -1);
```

```
    // if we include bridges as 2-node bcc
    F0R (i, m) if (is_bridge[i]) comps.pb({i});
  }

  int dfs(int u, int par) {
    int me = tin[u] = ++t, dp = me;
    for (auto [v, ei] : adj[u]) if (ei != par) {
      if (tin[v]) {
        dp = min(dp, tin[v]);
        if (tin[v] < me) stk.pb(ei); // bcc
      } else {
        int si = size(stk), up = dfs(v, ei);
        dp = min(dp, up);
        if (up == me) { // bcc and art
          is_art[u] = 1;
          stk.pb(ei);
          comps.pb({si + all(stk)});
          stk.resize(si);
        } else if (up < me) stk.push_back(ei); // bcc
        else is_bridge[ei] = 1; // up > me
      }
    }
    return dp;
  }
};
```

BlockCutTree.h

Description: First, use BiconnectedComponents to locate VERTEX components (including degree 0 nodes). To build a block cut tree, make a bipartite graph: Put all the normal nodes on the left, and make a new node for each bcc on the right. Draw edges from normal nodes to BCC that contain them. Note that the graph may be disconnected.

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.force(2); // var 2 is true`
`ts.at_most_one({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.

4f0e72, 51 lines

```
struct TwoSAT {
  int n = 0;
  vt<pi> edges;
  void init(int _n) { n = _n; }
  int add() { return n++; }
  void either(int x, int y) { // x | y
    x = max(2 * x, -1 - 2 * x); // ~(2 * x)
    y = max(2 * y, -1 - 2 * y); // ~(2 * y)
    edges.pb({x, y});
  }
  void implies(int x, int y) { either(~x, y); }
  void force(int x) { either(x, x); }
  void exactly_one(int x, int y) {
    either(x, y), either(~x, ~y);
  }
  void tie(int x, int y) {
    implies(x, y), implies(~x, ~y);
  }
  void nand(int x, int y ) { either(~x, ~y); }
  void at_most_one(const vt<int>& li) {
    if (size(li) <= 1) return;
    int cur = ~li[0];
    FOR (i, 2, size(li)) {
      int next = add();
      either(cur, ~li[i]);
      either(cur,next);
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  }
  vt<bool> solve() {
    SCC scc;
    scc.init(2 * n);
    for(auto& e : edges) {
      scc.ae(e.f ^ 1, e.s);
      scc.ae(e.s ^ 1, e.f);
    }
    scc.gen();
    reverse(all(scc.comps)); // reverse topo order
    for (int i = 0; i < 2 * n; i += 2)
      if (scc.comp[i] == scc.comp[i ^ 1]) return {};
    vt<int> tmp(2 * n);
    for (auto i : scc.comps) {
      if (!tmp[i]) tmp[i] = 1, tmp[scc.comp[i ^ 1]] = -1;
    }
    vt<bool> ans(n);
    F0R (i, n) ans[i] = tmp[scc.comp[2 * i]] == 1;
    return ans;
  }
};
```

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. For edges, push the edge u came from instead of current node. First, the graph (after removing directivity) must be connected. For undirected graphs, a tour exists when all nodes have even degree. For directed graphs, a tour exists when all nodes have equal in and out degree. For trails, the condition is the same as if you added an edge from t -> s.

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

378922, 14 lines

```
int n, m;
vt<vt<pair<int, int>>> adj;
vt<int> ret, used;

//
void dfs(int u) {
  while (adj[u].size()) {
    auto [v, ei] = adj[u].back();
    adj[u].pop_back();
    if (used[ei]++) continue;
    dfs(v);
  }
  ret.push_back(u);
}
```

DominatorTree.h

Description: Used only a few times. Assuming that all nodes are reachable from *root*, a dominates b iff every path from *root* to b passes through a.

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

4b8836, 41 lines

```
template<int SZ> struct Dominator {
  vi adj[SZ], ans[SZ]; // input edges, edges of dominator tree
  vi radj[SZ], child[SZ], sdomChild[SZ];
  int label[SZ], rlabel[SZ], sdom[SZ], dom[SZ], co = 0;
  int par[SZ], bes[SZ];
  void ae(int a, int b) { adj[a].pb(b); }
  int get(int x) { // DSU with path compression
    // get vertex with smallest sdom on path to root
    if (par[x] != x) {
      int t = get(par[x]); par[x] = par[par[x]];
      if (sdom[t] < sdom[bes[x]]) bes[x] = t;
    }
    return bes[x];
  }
  void dfs(int x) { // create DFS tree
    label[x] = ++co; rlabel[co] = x;
    sdom[co] = par[co] = bes[co] = co;
    each(y,adj[x]) {
      if (!label[y]) {
        dfs(y); child[label[x]].pb(label[y]); }
      radj[label[y]].pb(label[x]);
    }
  }
  void init(int root) {
    dfs(root);
    ROF(i,1,co+1) {
      each(j,radj[i]) ckmin(sdom[i],sdom[get(j)]);
      if (i > 1) sdomChild[sdom[i]].pb(i);
      each(j,sdomChild[i]) {
        int k = get(j);
        if (sdom[j] == sdom[k]) dom[j] = sdom[j];
        else dom[j] = k;
      }
      each(j,child[i]) par[j] = i;
    }
    FOR(i,2,co+1) {
      if (dom[i] != sdom[i]) dom[i] = dom[dom[i]];
      ans[rlabel[dom[i]]].pb(rlabel[i]);
    }
  }
};
```

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: $\mathcal{O}(NM)$

e210e2, 31 lines

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

```
    for (tie(u, v) = eds[i]; adj[u][ret[i]] != v;) ++ret[i];
  return ret;
}
```

7.6 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

b0d5b1, 12 lines

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

MaximumClique.h

Description: Quickly finds a maximum clique of a graph (given as symmetric bitset matrix; self-edges not allowed). Can be used to find a maximum independent set by finding a clique of the complement graph.

Time: Runs in about 1s for n=155 and worst case random graphs (p=.90). Runs faster for sparse graphs.

f7c0bc, 49 lines

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

MaximumIndependentSet.h

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

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

9fc465, 25 lines

```
vt<vi> build_table(vi &par){
  int d = 1;
  while ((1 << d) < size(par)) d++;
  vt<vt<int>> jmp(d, par);
  FOR (i, 1, d) FOR (j, 0, size(par))
    jmp[i][j] = jmp[i - 1][jmp[i - 1][j]];
  return jmp;
}

int jump(vt<vi>& jmp, int u, int d){
  F0R (i, size(jmp))
    if (1 & d >> i) u = jmp[i][u];
  return u;
}

int lca(vt<vi> &jmp, vi &depth, int a, int b) {
  if (depth[a] < depth[b]) swap(a, b);
  a = jump(jmp, a, depth[a] - depth[b]);
  if (a == b) return a;
  for (int i = size(jmp); i--;) {
    int c = jmp[i][a], d = jmp[i][b];
    if (c != d) a = c, b = d;
  }
  return jmp[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" bf464a, 20 lines

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

  LCA(vt<vi>& adj) : time(size(adj)), rmq((dfs(0, -1, adj), ret)) {}
  void dfs(int u, int p, vt<vi> &adj) {
    time[u] = t++;
    for (int v : adj[u]) if (v != p) {
      path.push_back(u), ret.push_back(time[u]);
      dfs(v, u, adj);
    }
  }
  int operator()(int u, int v) {
    if (u == v) return u;
    tie(u, v) = minmax(time[u], time[v]);
    return path[rmq.query(u, v)];
  }
};
```

VirtualTree.h

Description: Computes virtual tree. `pos` is inorder dfs time, and returns pairs of `(par, child)`.

Time: $\mathcal{O}(|S| \log |S|)$

"LCA.h" 4ff13b, 14 lines

```
// pos is dfs time
// pairs of {ancestor, child}
vt<pl> virtual_tree(vt<ll>& nodes) {
  auto cmp = [&] (ll u, ll v) { return pos[u] < pos[v]; };
  sort(all(nodes), cmp);
  int sz = size(nodes);
  FOR (i, sz - 1) nodes.pb(lca(nodes[i], nodes[i + 1]));
  sort(all(nodes), cmp);
  nodes.erase(unique(all(nodes)), nodes.end());
  vt<pl> res;
  FOR (i, (int) size(nodes) - 1)
  res.pb({lca(nodes[i], nodes[i + 1]), nodes[i + 1]});
  return res;
}
```

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 max queries, but can support commutative segtree modifications/queries on paths and subtrees. Takes as input the full adjacency list. in_edges being true means that values are stored in the edges, as opposed to the nodes. All values initialized to the segtree default. Root must be 0.

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

"../data-structures/LazySegmentTree.h" b1418d, 69 lines

```
template<bool in_edges> struct HLD {
  int n;
  vt<vi> adj;
  vi par, root, depth, sz, pos;
  int time;
  SegTree tree;
  void ae(int u, int v) {
    adj[u].pb(v);
    adj[v].pb(u);
  }
  void dfs_sz(int u) {
    sz[u] = 1;
    for (int& v : adj[u]) {
      par[v] = u;
      depth[v] = depth[u] + 1;
      adj[v].erase(find(all(adj[v]), u));
      dfs_sz(v);
      sz[u] += sz[v];
      if (sz[v] > sz[adj[u][0]]) swap(v, adj[u][0]);
    }
  }
  void dfs_hld(int u) {
    pos[u] = time++;
    for (int& v : adj[u]) {
      root[v] = (v == adj[u][0] ? root[u] : v);
      dfs_hld(v);
    }
  }
  void init(int _n) {
    n = _n;
```

```
    adj.resize(n);
    par = root = depth = sz = pos = vi(n);
  }
  void gen(int r = 0) {
    par[r] = depth[r] = time = 0;
    dfs_sz(r);
    root[r] = r;
    dfs_hld(r);
    tree.init(n);
  }
  int lca(int u, int v) {
    while (root[u] != root[v]) {
      if (depth[root[u]] > depth[root[v]]) swap(u, v);
      v = par[root[v]];
    }
    return depth[u] < depth[v] ? u : v;
  }
  template <class Op>
  void process(int u, int v, Op op) {
    for (;; v = par[root[v]]) {
      if (pos[u] > pos[v]) swap(u, v);
      if (root[u] == root[v]) break;
      op(pos[root[v]], pos[v] + 1);
    }
    op(pos[u] + in_edges, pos[v] + 1);
  }
  void upd(int u, int v, ll upd) {
    process(u, v, [&] (int l, int r) {
      tree.upd(l, r, upd);
    });
  }
  ll query(int u, int v) {
    ll res = 0;
    process(u, v, [&] (int l, int r) {
      res = res + tree.query(l, r);
    });
    return res;
  }
};
```

CentroidDecomp.h

Description: Give order to call dfs on.
Time: $\mathcal{O}(N)$

14b923, 15 lines

```
vector<pair<int, int>> ord;
auto dfs1 = [&] (auto &&self, int u, int p) -> int {
  int msk1 = 0, msk2 = 0;
  for (auto [v, w] : adj[u]) if (v - p) {
    int res = self(self, v, u);
    msk2 |= msk1 & res;
    msk1 |= res;
  }
  int res = (msk1 | ((1 << __lg(2 * msk2 + 1)) - 1)) + 1;
  pri[u] = __builtin_ctz(res);
  ord.push_back({-__builtin_ctz(res), u});
  return res;
};
dfs1(dfs1, 0, -1);
sort(all(ord));
```

LinkCutTree.h

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

0fb462, 90 lines

```
struct Node { // Splay tree. Root's pp contains tree's parent.
  Node *p = 0, *pp = 0, *c[2];
  bool flip = 0;
  Node() { c[0] = c[1] = 0; fix(); }
  void fix() {
    if (c[0]) c[0]->p = this;
    if (c[1]) c[1]->p = this;
    // (+ update sum of subtree elements etc. if wanted)
  }
  void pushFlip() {
    if (!flip) return;
    flip = 0; swap(c[0], c[1]);
    if (c[0]) c[0]->flip ^= 1;
    if (c[1]) c[1]->flip ^= 1;
  }
  int up() { return p ? p->c[1] == this : -1; }
  void rot(int i, int b) {
    int h = i ^ b;
    Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ? y : x;
    if ((y->p = p)) p->c[up()] = y;
    c[i] = z->c[i ^ 1];
    if (b < 2) {
      x->c[h] = y->c[h ^ 1];
      y->c[h ^ 1] = x;
    }
    z->c[i ^ 1] = this;
    fix(); x->fix(); y->fix();
    if (p) p->fix();
    swap(pp, y->pp);
  }
  void splay() {
    for (pushFlip(); p; ) {
      if (p->p) p->p->pushFlip();
      p->pushFlip(); pushFlip();
      int c1 = up(), c2 = p->up();
      if (c2 == -1) p->rot(c1, 2);
      else p->p->rot(c2, c1 != c2);
    }
  }
  Node* first() {
    pushFlip();
    return c[0] ? c[0]->first() : (splay(), this);
  }
};

struct LinkCut {
  vector<Node> node;
  LinkCut(int N) : node(N) {}

  void link(int u, int v) { // add an edge (u, v)
    assert(!connected(u, v));
    makeRoot(&node[u]);
    node[u].pp = &node[v];
  }
  void cut(int u, int v) { // remove an edge (u, v)
    Node *x = &node[u], *top = &node[v];
    makeRoot(top); x->splay();
    assert(top == (x->pp ?: x->c[0]));
    if (x->pp) x->pp = 0;
    else {
      x->c[0] = top->p = 0;
      x->fix();
    }
  }
  bool connected(int u, int v) { // are u, v in the same tree?
    Node* nu = access(&node[u])->first();
    return nu == access(&node[v])->first();
  }
  void makeRoot(Node* u) {
    access(u);
    u->splay();
    if(u->c[0]) {
      u->c[0]->p = 0;
      u->c[0]->flip ^= 1;
      u->c[0]->pp = u;
      u->c[0] = 0;
      u->fix();
    }
  }
  Node* access(Node* u) {
    u->splay();
    while (Node* pp = u->pp) {
      pp->splay(); u->pp = 0;
      if (pp->c[1]) {
        pp->c[1]->p = 0; pp->c[1]->pp = pp; }
      pp->c[1] = u; pp->fix(); u = pp;
    }
    return u;
  }
};
```

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" 39e620, 60 lines

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

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

```
        cycs.push_front({u, time, {&Q[qi], &Q[end]}});
      }
    }
    rep(i,0,qi) in[uf.find(Q[i].b)] = Q[i];
  }

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

7.8 Other

KthWalk.h

Description: K-th shortest walk from `src` to `des` in digraph. All edge weights must be non-negative.
Memory: $\mathcal{O}((M+N)\log N + K)$
Time: $\mathcal{O}((M+N)\log N + K\log K)$

"LeftistHeap.h" 55008f, 61 lines

```
int N,M,src,des,K;

ph cand[MX];
vector<array<int,3>> adj[MX], radj[MX];
pi pre[MX];
ll dist[MX];

struct state {
  int vert; ph p; ll cost;
  bool operator<(const state& s) const { return cost > s.cost; }
};
priority_queue<state> ans;

void genHeaps() {
  F0R(i,N) dist[i] = INF, pre[i] = {-1,-1};
  priority_queue<T,vector<T>,greater<T>> pq;
  auto ad = [&](int a, ll b, pi ind) {
    if (dist[a] <= b) return;
    pre[a] = ind; pq.push({dist[a] = b,a});
  };
  ad(des,0,{-1,-1});
  vi seq; // reachable vertices in order of dist
  while (sz(pq)) {
    auto a = pq.top(); pq.pop();
    if (a.f > dist[a.s]) continue;
    seq.pb(a.s); each(t,radj[a.s]) ad(t[0],a.f+t[1],{t[2],a.s}); //
        edge index, vert
  }
  each(t,seq) {
    each(u,adj[t]) if (u[2] != pre[t].f && dist[u[0]] != INF) {
      ll cost = dist[u[0]]+u[1]-dist[t]; assert(cost >= 0);
      cand[t] = ins(cand[t],{cost,u[0]});
    }
    if (pre[t].f != -1) cand[t] = meld(cand[t],cand[pre[t].s]);
    if (t == src) {
      ps(dist[t]); K --;
      if (cand[t]) ans.push(state{t,cand[t],dist[t]+cand[t]->v.f});
    }
  }
}

void solve() {
  re(N,M,src,des,K);
  F0R(i,M) {
    int u,v,w; re(u,v,w);
    adj[u].pb({v,w,i}); radj[v].pb({u,w,i}); // vert, weight, label
  }
  genHeaps();
  F0R(i,K) {
    if (!sz(ans)) {
      ps(-1);
      continue;
    }
    auto a = ans.top(); ans.pop();
    int vert = a.vert;
    ps(a.cost);
    if (a.p->l) ans.push(state{vert,a.p->l,a.cost+a.p->l->v.f-a.p->v.f
        });
    if (a.p->r) ans.push(state{vert,a.p->r,a.cost+a.p->r->v.f-a.p->v.f
        });
    int V = a.p->v.s;
    if (cand[V]) ans.push(state{V,cand[V],a.cost+cand[V]->v.f});
  }
}
```

7.9 Math

7.9.1 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).

7.9.2 Erdős–Gallai theorem

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

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

Geometry (8)

8.1 Notes

For checking imprecise equalities use `abs(x) < eps`.
Clamp values to $[-1, 1]$ for `acos`, etc.

8.2 Precision

double can precisely store integers up to 9×10^{15}, and **long double** can store up to 1.8×10^{19}.
Operations $(+, \times, -, /, \sqrt{x})$ will be rounded to the nearest representable value. Hence, the **relative** error on the result of one of these operations is bounded by $\epsilon = 1.2 \times 10^{-16}$ for **double** and $\epsilon = 5.5 \times 10^{-20}$ for **long double**.
If computations are precise up to a relative error of ϵ, and the magnitude of our values never exceeds M, then the absolute error of an operation is at most $M\epsilon$.
It is typically **more useful to consider absolute error** instead of relative error (1).
A series of n + and − operations result in an absolute error of at most $nM\epsilon$.
With $\times$, the absolute error of a d-dimensional value computed in n operations is $M^d((1+\epsilon)^n - 1) \approx nM^d\epsilon$ (excluding multiplication by adimensional values).
Imprecisions on x in $\frac{1}{x}$ can cause an arbitrary error (2).
Imprecisions on x in $\sqrt{x}$ are typically bad, particularly near 0. (3)
Note that $\frac{1}{x}$ and $\sqrt{x}$ perform poorly on imprecise inputs: when working with exact inputs, the `IEEE 754` standard guarantees a relative error of ϵ/absolute error of $M\epsilon$.
In particular, circle/line/circle-line intersections on exact coordinates have relative error proportional to ϵ/absolute error proportional to $M\epsilon$.

(1) subtracting two large imprecise values can blow up relative errors
(2) division by a value near 0 can result in an arbitrarily large error in both positive and negative
(3) assuming arguments to $\sqrt{x}$ are non-negative, imprecisions on x will have the most impact when x is near 0: for a given imprecision δ, the biggest imprecision on $\sqrt{x}$ it might cause is $\sqrt{\delta}$. This is quite bad: an argument with imprecision $nM^2\epsilon$ will become an imprecision of $\sqrt{n}M\sqrt{\epsilon}$. This can appear with circle intersections and computing roots to quadratics.

StableSum.h

Description: Accumulates positive floating point numbers with better precision.

ad65a7, 16 lines

```
struct StableSum {
  int cnt = 0;
  vt<db> v, pref{0};
  void operator+=(db a) {
    assert(a >= 0);
    int s = ++cnt;
    while (s % 2 == 0) {
      a += v.back();
      v.pop_back(), pref.pop_back();
      s /= 2;
    }
    v.pb(a);
    pref.pb(pref.back() + a);
  }
  double val() { return pref.back(); }
};
```

8.3 Common

8.3.1 Equation of line through two points

Given points p and q, equation of the line $ax + by + c = 0$ is given by:

$$a = p_y - q_y$$
$$b = q_x - p_x$$
$$c = p \times q$$

8.4 Geometric primitives

Point.h

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

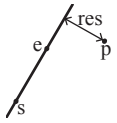
09a101, 29 lines

```
template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<class T>
```

```
struct Point {
  using P = Point<T>;
  T x, y;
  #define op1(o) P operator o (P p) const { return {x o p.x, y o p.y}; }
  op1(+) op1(-)
  #define op2(o) P operator o (T z) const { return {x o z, y o z}; }
  op2(*) op2(/)
  T dot(P p) const { return x * p.x + y * p.y; }
  T cross(P p) const { return x * p.y - y * p.x; }
  #define op3(o) T o (P a, P b) const { return (a - *this). o (b - *this); }
  op3(dot) op3(cross)
  #define op4(o) bool operator o (P p) const { return tie(x, y) o tie(p.x, p.y); }
  op4(<) op4(==)
  T dist2() const { return x * x + y * y; }
  db dist() const { return sqrtl((db) dist2()); }
  // angle to x-axis in interval [-pi, pi]
  db angle() const { return atan2l(y, x); }
  P unit() const { return *this / dist(); }
  P perp() const { return {-y, x}; }
  P normal() const { return perp().unit(); }
  P rotate(db a) const {
    return P(x * cos(a) - y * sin(a), x * sin(a) + y * cos(a));
  }
  friend ostream& operator<<(ostream& os, P p) {
    return os << "(" << p.x << ", " << p.y << ")";
  }
};
```

lineDistance.h

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

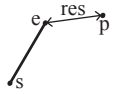


"Point.h" 3b34c4, 4 lines

```
template<class P>
double line_dist(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 on_segment = seg_dist(a,b,p) < 1e-10;`

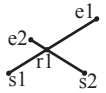


"Point.h" 579797, 5 lines

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

SegmentIntersection.h

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

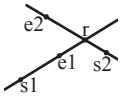


"Point.h", "OnSegment.h" bd6e14, 13 lines

```
template<class P> vt<P> seg_inter(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 (on_segment(c, d, a)) s.insert(a);
  if (on_segment(c, d, b)) s.insert(b);
  if (on_segment(a, b, c)) s.insert(c);
  if (on_segment(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 = line_inter(s1,e1,s2,e2);`
`if (res.first == 1)`
`cout << "intersection point at " << res.second << endl;`



"Point.h" 5d933a, 17 lines

```
template<class P>
pair<int, P> line_inter(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};
}

template<class P>
pair<int, P> line_inter(P s0, P d0, P s1, P d1) {
  auto d = (d0).cross(d1);
  if (d == 0) // if parallel
    return {0, {}};
  auto p = (s0 + d0 - s1).cross(d1), q = (d1).cross(s0 - s1);
  return {1, (s0 * p + (s0 + d0) * q) / d}
}
```

sideOf.h

Description: Returns where p is as seen from s towards e. 1/0/-1 $\Leftrightarrow$ left/on line/right. If the optional argument eps is given 0 is returned if p is within distance eps from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.
Usage: `bool left = side_of(p1,p2,q)==1;`

"Point.h" 9e71fb, 9 lines

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

template<class P>
int side_of(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

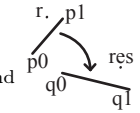
Description: Returns true iff p lies on the line segment from s to e. Use `(seg_dist(s,e,p) < epsilon)` instead when using Point<double>.

"Point.h" 0c1f75, 3 lines

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

linearTransformation.h

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



"Point.h" 8e73be, 6 lines

```
typedef Point<double> P;
P linear_transformation(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" da84d5, 5 lines

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

Angle.h

Description: A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping. Sometimes also represents points or vectors.
Usage: `vector<Angle> v = {w[0], w[0].t360() ...}; // sorted`
`int j = 0; rep(i,0,n) { while (v[j] < v[i].t180()) ++j; }`
`// sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i`

0f0602, 35 lines

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

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

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

8.5 Circles

CircleIntersection.h

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

"Point.h" b70280, 11 lines

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

CircleTangents.h

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

"Point.h" b0153d, 13 lines

```
template<class P>
vector<pair<P, P>> tangents(P c1, double r1, P c2, double r2) {
  P d = c2 - c1;
  double dr = r1 - r2, d2 = d.dist2(), h2 = d2 - dr * dr;
  if (d2 == 0 || h2 < 0)  return {};
  vector<pair<P, P>> out;
  for (double sign : {-1, 1}) {
    P v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;
    out.push_back({c1 + v * r1, c2 + v * r2});
  }
  if (h2 == 0) out.pop_back();
  return out;
}
```

CircleLine.h

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

"Point.h" afd5f9, 9 lines

```
template<class P>
vector<P> circle_line(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: $\mathcal{O}(n)$

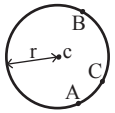
"../../content/geometry/Point.h" e876aa, 19 lines

```
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 = q + d * (t - 1);
    return arg(p, u) * r2 + u.cross(v) / 2 + arg(v, q) * r2;
  };
  auto sum = 0.0;
  F0R (i, size(ps))
    sum += tri(ps[i] - c, ps[(i + 1) % size(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" 30a12d, 9 lines

```
typedef Point<double> P;
double cc_radius(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 cc_center(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" 256373, 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;
  F0R (i, size(ps)) if ((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    F0R (j, i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      F0R (k, j) if ((o - ps[k]).dist() > r * EPS) {
        o = cc_center(ps[i], ps[j], ps[k]);
        r = (o - ps[i]).dist();
      }
    }
  }
  return {o, r};
}
```

8.6 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 = in_polygon(v, P{3, 3}, false);

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

"Point.h", "OnSegment.h", "SegmentDistance.h" b915a1, 11 lines

```
template<class P>
bool in_polygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = sz(p);
  F0R (i, n) {
    P q = p[(i + 1) % n];
    if (on_segment(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" c9f086, 6 lines

```
template<class T>
T polygon_area(vector<Point<T>>& v) {
  T a = v.back().cross(v[0]);
  F0R (i, size(v) - 1) a += v[i].cross(v[i + 1]);
  return a;
}
```

PolygonCenter.h

Description: Returns the center of mass for a polygon.

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

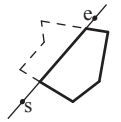
"Point.h" ce7a6a, 9 lines

```
typedef Point<db> P;
P polygon_center(const vector<P>& v) {
  P res(0, 0); db a = 0;
  for (int i = 0, j = size(v) - 1; i < size(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" d8e942, 13 lines

```
typedef Point<db> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  F0R (i, size(poly)) {
    P cur = poly[i], prev = i ? poly[i - 1] : poly.back();
    auto a = s.cross(e, cur), b = s.cross(e, prev);
```

```
    if ((a < 0) != (b < 0))
      res.push_back(cur + (prev - cur) * (a / (a - b)));
    if (a < 0)
      res.push_back(cur);
  }
  return res;
}
```

PolygonUnion.h

Description: Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)

Time: $\mathcal{O}\left(N^2\right)$, where N is the total number of points

"Point.h", "sideOf.h" b287c2, 33 lines

```
typedef Point<db> 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;
  F0R (i, size(poly)) F0R (v, size(poly[i])) {
    P A = poly[i][v], B = poly[i][(v + 1) % sz(poly[i])];
    vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
    F0R (j, size(poly)) if (i != j) {
      rep(u,0,sz(poly[j])) {
        P C = poly[j][u], D = poly[j][(u + 1) % sz(poly[j])];
        int sc = side_of(A, B, C), sd = side_of(A, B, D);
        if (sc != sd) {
          db sa = C.cross(D, A), sb = C.cross(D, B);
          if (min(sc, sd) < 0)
            segs.emplace_back(sa / (sa - sb), sgn(sc - sd));
        } else if (!sc && !sd && j < i && sgn((B - A).dot(D-C)) > 0){
          segs.emplace_back(rat(C - A, B - A), 1);
          segs.emplace_back(rat(D - A, B - A), -1);
        }
      }
    }
    sort(all(segs));
    for (auto& s : segs) s.first = min(max(s.first, 0.0), 1.0);
    double sum = 0;
    int cnt = segs[0].second;
    FOR (j, 1, size(segs)) {
      if (!cnt) sum += segs[j].first - segs[j - 1].first;
      cnt += segs[j].second;
    }
    ret += A.cross(B) * sum;
  }
  return ret / 2;
}
```

ConvexHull.h

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

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

"Point.h" 5e9915, 12 lines

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

HullDiameter.h

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

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

"Point.h" 8279d4, 12 lines

```
typedef Point<ll> P;
array<P, 2> hull_diameter(vector<P> s) {
  int n = size(s), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {s[0], s[0]}});
  F0R (i, j)
    for (;; j = (j + 1) % n) {
      res = max(res, {(s[i] - s[j]).dist2(), {s[i], s[j]}});
      if ((s[(j + 1) % n] - s[j]).cross(s[i + 1] - s[i]) >= 0)
        break;
    }
  return res.second;
}
```

PointInsideHull.h

Description: Determine whether a point t lies inside a convex hull (CCW order, with no collinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.

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

"Point.h", "sideOf.h", "OnSegment.h" c366a3, 13 lines

```
using P = Point<ll>;
bool in_hull(const vector<P>& l, P p, bool strict = true) {
  int a = 1, b = size(l) - 1, r = !strict;
  if (size(l) < 3) return r && on_segment(l[0], l.back(), p);
  if (side_of(l[0], l[a], l[b]) > 0) swap(a, b);
  if (side_of(l[0], l[a], p) >= r || side_of(l[0], l[b], p) <= -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (side_of(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: $\mathcal{O}\left(\log n\right)$

"Point.h" bf84fc, 39 lines

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

#define cmpL(i) sgn(a.cross(poly[i], b))
template <class P>
array<int, 2> lineHull(P a, P b, vector<P>& poly) {
  int endA = extrVertex(poly, (a - b).perp());
  int endB = extrVertex(poly, (b - a).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<int, 2> res;
  F0R (i, 2) {
    int lo = endB, hi = endA, n = size(poly);
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
      (cmpL(m) == cmpL(endB) ? lo : hi) = m;
    }
    res[i] = (lo + !cmpL(hi)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res[0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
      case 0: return {res[0], res[0]};
      case 2: return {res[1], res[1]};
    }
  return res;
}
```

MinkowskiSum.h

Description: Minkowski sum of two convex polygons given in CCW order.

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

e5d935, 29 lines

```
vP minkowski_sum(vP a, vP b) {
  if (sz(a) > sz(b)) swap(a, b);
  if (!sz(a)) return {};
  if (sz(a) == 1) {
    each(t, b) t += a.ft;
    return b;
  }
  rotate(begin(a), min_element(all(a)), end(a));
  rotate(begin(b), min_element(all(b)), end(b));
  a.pb(a[0]), a.pb(a[1]);
  b.pb(b[0]), b.pb(b[1]);
  vP result;
  int i = 0, j = 0;
  while (i < sz(a)-2 || j < sz(b)-2) {
    result.pb(a[i]+b[j]);
    T crs = cross(a[i+1]-a[i],b[j+1]-b[j]);
    i += (crs >= 0);
    j += (crs <= 0);
  }
  return result;
}

T diameter2(vP p) { // example application: squared diameter
  vP a = hull(p);
  vP b = a; each(t, b) t *= -1;
  vP c = minkowski_sum(a, b);
  T ret = 0; each(t, c) ckmax(ret, abs2(t));
  return ret;
}
```

8.7 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.

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

"Point.h" be22ff, 16 lines

```
pair<P, P> closest(vector<P> v) {
  assert(size(v) > 1);
  set<P> S;
  sort(all(v), [](P a, P b) { return a.y < b.y; });
  pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
  int j = 0;
  for (P p : v) {
    P d{1 + (ll)sqrt(ret.first), 0};
```

```
    while (v[j].y <= p.y - d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d);
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).dist2(), {*lo, p}});
    S.insert(p);
  }
  return ret.second;
}
```

ManhattanMST.h

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

"Point.h" c36e89, 23 lines

```
typedef Point<int> P;
vt<array<int, 3>> manhattanMST(vt<P> ps) {
  vi id(size(ps));
  iota(all(id), 0);
  vector<array<int, 3>> edges;
  F0R (k, 4) {
    sort(all(id), [&](int i, int j) {
       return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;});
    map<int, int> sweep;
    for (int i : id) {
      for (auto it = sweep.lower_bound(-ps[i].y);
            it != sweep.end(); sweep.erase(it++)) {
        int j = it->second;
        P d = ps[i] - ps[j];
        if (d.y > d.x) break;
        edges.push_back({d.y + d.x, i, j});
      }
      sweep[-ps[i].y] = i;
    }
    for (P& p : ps) if (k & 1) p.x = -p.x; else swap(p.x, p.y);
  }
  return edges;
}
```

kdTree.h

Description: KD-tree (2d)

"Point.h" ad9b75, 53 lines

```
using P = array<int, 2>;
struct Node {
  #define _sq(x) (x) * (x)
  P lo, hi;
  struct Node *lc, *rc;

  ll dist2(const P &a, const P &b) const {
    return 1ll * _sq(a[0] - b[0]) + 1ll * _sq(a[1] - b[1]);
  }

  ll dist2(P &p) {
    #define _loc(i) (p[i] < lo[i] ? lo[i] : (p[i] > hi[i] ? hi[i] : p[
        i]))
    return dist2(p, {_loc(0), _loc(1)});
    // ll res = 0;
    // F0R (i, 2) {
    //   ll tmp = (p[i] < lo[i] ? lo[i] - p[i] : 0) + (hi[i] < p[i] ?
        p[i] - hi[i] : 0);
    //   res += tmp * tmp;
    // }
    // return res;
  }

  template<class ptr>
  Node (ptr l, ptr r, int d) : lc(0), rc(0) {
    lo = {inf, inf}, hi = {-inf, -inf};
    for (ptr p = l; p < r; p++) {
      F0R (i, 2) lo[i] = min(lo[i], (*p)[i]), hi[i] = max(hi[i], (*p)[
          i]);
    }
    if (r - l == 1) return;
    ptr m = l + (r - l) / 2;
    nth_element(l, m, r, [&] (auto a, auto b) { return a[d] < b[d]; })
        ;
    lc = new Node(l, m, d ^ 1);
    rc = new Node(m, r, d ^ 1);
  }

  void search(P p, ll &best) {
    if (lc) { // rc will also exist
      ll dl = lc->dist2(p), dr = rc->dist2(p);
      if (dl > dr) swap(lc, rc), dr = dl;
      lc->search(p, best);
      if (dr < best) rc->search(p, best);
    } else best = min(best, dist2(p, lo));
  }

  // fill pq with k infinities for nearest k points
  void search(P p, priority_queue<ll> &pq) {
    if (lc) {
      ll dl = lc->dist2(p), dr = rc->dist2(p);
      if (dl > dr) swap(lc, rc), dr = dl;
      lc->search(p, pq);
      if (dr < pq.top()) rc->search(p, pq);
    } else pq.push(dist2(p, lo)), pq.pop();
  }
};
```

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" eefdf5, 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;
  rep(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;
  int 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).first;
  vector<Q> q = {e};
  int 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.push_back(c->p); \
  q.push_back(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

8.8 3D

PolyhedronVolume.h

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

840bbc, 6 lines

```
template<class V, class L>
db signedPolyVolume(const V& p, const L& trilist) {
  db v = 0;
```

```
  for (auto i : trilist) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
  return v / 6;
}
```

Point3D.h

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

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;
  }
};
```

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

"Point3D.h"

5b45fc, 49 lines

```
typedef Point3D<double> P3;

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

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

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

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

sphericalDistance.h

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

67c08d, 8 lines

```
db sphericalDistance(db f1, db t1,
    db f2, db t2, db radius) {
  db dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  db dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  db dz = cos(t2) - cos(t1);
  db 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: $\mathcal{O}(n)$

d4375c, 16 lines

```
vi pi(const string& s) {
  vi p(sz(s));
  rep(i,1,sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

vi match(const string& s, const string& pat) {
  vi p = pi(pat + '\0' + s), res;
  rep(i,sz(p)-sz(s),sz(p))
    if (p[i] == sz(pat)) res.push_back(i - 2 * sz(pat));
  return res;
}
```

Zfunc.h

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

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

98e47b, 12 lines

```
vi Z(const string& S) {
  vi z(size(S));
  int l = -1, r = -1;
  FOR (i, 1, size(S)) {
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    while (i + z[i] < size(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: $\mathcal{O}(N)$

1c24a1, 13 lines

```
array<vi, 2> manacher(const string &s) {
  int n = size(s);
  array<vi, 2> p = {vi(n + 1), vi(n)};
  FOR (z, 2) for (int i = 0, l = 0, r = 0; i < n; i++) {
    int t = r - i + !z;
    if (i < r) p[z][i] = min(t, p[z][l + t]);
    int L = i - p[z][i], R = i + p[z][i] - !z;
    while (L >= 1 && R + 1 < n && s[L - 1] == s[R + 1])
      p[z][i]++, L--, R++;
    if (R > r) l = L, r = R;
  }
  return p;
}
```

MinRotation.h

Description: Finds the lexicographically smallest rotation of a string.

Usage: `rotate(v.begin(), v.begin() + minRotation(v), v.end());`

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

9bd725, 8 lines

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

SuffixArray.h

Description: Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. The input string must not contain any nul chars.

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

a982c3, 22 lines

```
struct SuffixArray {
  vi sa, lcp;
  SuffixArray(string s, int lim = 256) { // or vector<int>
    s.push_back(0); int n = size(s), k = 0, a, b;
```

```
    vi x(all(s)), y(n), ws(max(n, lim));
    sa = lcp = y, iota(all(sa), 0);
    for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
      p = j, iota(all(y), n - j);
      F0R (i, n) if (sa[i] >= j) y[p++] = sa[i] - j;
      fill(all(ws), 0);
      F0R (i, n) ws[x[i]]++;
      FOR (i, 1, lim) ws[i] += ws[i - 1];
      for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      FOR (i, 1, n) a = sa[i - 1], b = sa[i], x[b] =
        (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
    }
    for (int i = 0, j; i < n - 1; lcp[x[i++]] = k)
      for (k && k--, j = sa[x[i] - 1];
          s[i + k] == s[j + k]; 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: $\mathcal{O}(26N)$

aae0b8, 50 lines

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

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

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

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

BigMod.h

Description: $2^{64} - 1$ mod int

b4cfda, 18 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.
using ull = unsigned long long;
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)
```

Hashing.h

Description: Self-explanatory methods for string hashing. Skip the stuff that starts with r if you don't care about reverse hashes etc.

"BigMod.h" ee3cb7, 39 lines

```
struct HashInterval {
  vt<H> ha, pw, rha;
  template<class T>
  HashInterval(T& str) : ha(size(str) + 1), pw(ha), rha(ha) {
    pw[0] = 1;
    F0R (i, size(str)) {
      ha[i + 1] = ha[i] * C + str[i] + 1;
      pw[i + 1] = pw[i] * C;
    }
    ROF (i, 0, size(str)) rha[i] = rha[i + 1] * C + str[i] + 1;
  }
  H hash_interval(int a, int b) { // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
  H rhash_interval(int a, int b) { // hash [a, b) from right to left
    return rha[a] - rha[b] * pw[b - a];
  }
};

// get all hashes of length <len>
template<class T>
vector<H> get_hashes(T& str, int length) {
  if (size(str) < length) return {};
  H h = 0, pw = 1;
  F0R (i, length) h = h * C + str[i] + 1, pw = pw * C;
  vector<H> ret = {h};
  FOR (i, length, size(str)) {
    ret.push_back(h = h * C + str[i] + 1
      - pw * (str[i - length] + 1));
  }
  return ret;
}
```

```
template<class T>
H hash_string(T& s) {
  H h = 1;
  for (auto c : s) h = h * C + c + 1;
  return h;
}
```

AhoCorasick.h

Description: Aho-Corasick automaton, used for multiple pattern matching. Initialize with AhoCorasick ac(patterns); the automaton start node will be at index 0. find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll(−, word) finds all words (up to $N\sqrt{N}$ many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. To find the longest words that start at each position, reverse all input. For large alphabets, split each symbol into chunks, with sentinel bits for symbol boundaries.
Time: construction takes $\mathcal{O}(26N)$, where N = sum of length of patterns. find(x) is $\mathcal{O}(N)$, where N = length of x. findAll is $\mathcal{O}(NM)$.

f35677, 66 lines

```
struct AhoCorasick {
  enum {alpha = 26, first = 'A'}; // change this!
  struct Node {
    // (nmatches is optional)
    int back, next[alpha], start = -1, end = -1, nmatches = 0;
    Node(int v) { memset(next, v, sizeof(next)); }
  };
  vector<Node> N;
  vi backp;
  void insert(string& s, int j) {
    assert(!s.empty());
    int n = 0;
    for (char c : s) {
      int& m = N[n].next[c - first];
      if (m == -1) { n = m = sz(N); N.emplace_back(-1); }
      else n = m;
    }
    if (N[n].end == -1) N[n].start = j;
    backp.push_back(N[n].end);
    N[n].end = j;
    N[n].nmatches++;
  }
  AhoCorasick(vector<string>& pat) : N(1, -1) {
    rep(i,0,sz(pat)) insert(pat[i], i);
    N[0].back = sz(N);
    N.emplace_back(0);

    queue<int> q;
    for (q.push(0); !q.empty(); q.pop()) {
      int n = q.front(), prev = N[n].back;
      rep(i,0,alpha) {
        int &ed = N[n].next[i], y = N[prev].next[i];
        if (ed == -1) ed = y;
        else {
          N[ed].back = y;
          (N[ed].end == -1 ? N[ed].end : backp[N[ed].start])
            = N[y].end;
          N[ed].nmatches += N[y].nmatches;
          q.push(ed);
        }
      }
    }
  }
  vi find(string word) {
    int n = 0;
    vi res; // ll count = 0;
    for (char c : word) {
      n = N[n].next[c - first];
      res.push_back(N[n].end);
```

```
    // count += N[n].nmatches;
   }
   return res;
  }
  vector<vi> findAll(vector<string>& pat, string word) {
   vi r = find(word);
   vector<vi> res(sz(word));
   rep(i,0,sz(word)) {
    int ind = r[i];
    while (ind != -1) {
     res[i - sz(pat[ind]) + 1].push_back(ind);
     ind = backp[ind];
    }
   }
   return res;
  }
};
```

SuffixAutomaton.h

Description: Used infrequently. Constructs minimal deterministic finite automaton (DFA) that recognizes all suffixes of a string. `len` corresponds to the maximum length of a string in the equivalence class, `pos` corresponds to the first ending position of such a string, `lnk` corresponds to the longest suffix that is in a different class. Suffix links correspond to suffix tree of the reversed string!
Time: $\mathcal{O}(N \log \sum)$

a99c6d, 67 lines

```
struct SuffixAutomaton {
  int N = 1; vi lnk{-1}, len{0}, pos{-1}; // suffix link,
  // max length of state, last pos of first occurrence of state
  V<map<char,int>> nex{1}; V<bool> isClone{0};
  // transitions, cloned -> not terminal state
  V<vi> iLnk; // inverse links
  int add(int p, char c) { // ~p nonzero if p != -1
    auto getNex = [&]() {
      if (p == -1) return 0;
      int q = nex[p][c]; if (len[p]+1 == len[q]) return q;
      int clone = N++; lnk.pb(lnk[q]); lnk[q] = clone;
      len.pb(len[p]+1), nex.pb(nex[q]),
      pos.pb(pos[q]), isClone.pb(1);
      for (; ~p && nex[p][c] == q; p = lnk[p]) nex[p][c]=clone;
      return clone;
    };
    // if (nex[p].count(c)) return getNex();
    // ^ need if adding > 1 string
    int cur = N++; // make new state
    lnk.eb(), len.pb(len[p]+1), nex.eb(),
    pos.pb(pos[p]+1), isClone.pb(0);
    for (; ~p && !nex[p].count(c); p = lnk[p]) nex[p][c] = cur;
    int x = getNex(); lnk[cur] = x; return cur;
  }
  void init(str s) { int p = 0; each(x,s) p = add(p,x); }
  // inverse links
  void genIlnk() {iLnk.rsz(N);FOR(v,1,N)iLnk[lnk[v]].pb(v);}
  // APPLICATIONS
  void getAllOccur(vi& oc, int v) {
    if (!isClone[v]) oc.pb(pos[v]); // terminal position
    each(u,iLnk[v]) getAllOccur(oc,u); }
  vi allOccur(str s) { // get all occurrences of s in automaton
    int cur = 0;
    each(x,s) {
      if (!nex[cur].count(x)) return {};
      cur = nex[cur][x]; }
    // convert end pos -> start pos
    vi oc; getAllOccur(oc,cur); each(t,oc) t += 1-sz(s);
    sort(all(oc)); return oc;
  }
  vl distinct;
  ll getDistinct(int x) {
    // # distinct strings starting at state x
    if (distinct[x]) return distinct[x];
    distinct[x]=1;each(y,nex[x]) distinct[x]+=getDistinct(y.s);
    return distinct[x]; }
  ll numDistinct() { // # distinct substrings including empty
    distinct.rsz(N); return getDistinct(0); }
  ll numDistinct2() { // assert(numDistinct()==numDistinct2());
    ll ans = 1; FOR(i,1,N) ans += len[i]-len[lnk[i]];
    return ans; }
};

SuffixAutomaton S;
vi sa; str s;
void dfs(int x) {
  if (!S.isClone[x]) sa.pb(sz(s)-1-S.pos[x]);
  V<pair<char,int>> chr;
  each(t,S.iLnk[x]) chr.pb({s[S.pos[t]-S.len[x]],t});
  sort(all(chr)); each(t,chr) dfs(t.s);
}

int main() {
  re(s); reverse(all(s));
  S.init(s); S.genIlnk();
  dfs(0); ps(sa); // generating suffix array for s
}
```

PalTree.h

Description: Used infrequently. Palindromic tree computes number of occurrences of each palindrome within string. `ans[i][0]` stores min even x such that the prefix $s[1..i]$ can be split into exactly x palindromes, `ans[i][1]` does the same for odd x.
Time: $\mathcal{O}(N \sum)$ for addChar, $\mathcal{O}(N \log N)$ for updAns

8a7d31, 41 lines

```
struct PalTree {
  static const int ASZ = 26;
  struct node {
    AR<int,ASZ> to = AR<int,ASZ>();
    int len, link, oc = 0; // # occurrences of pal
    int slink = 0, diff = 0;
    AR<int,2> seriesAns;
    node(int _len, int _link) : len(_len), link(_link) {}
  };
  str s = "@"; V<AR<int,2>> ans = {{0,MOD}};
  V<node> d = {{0,1},{-1,0}}; // dummy pals of len 0,-1
  int last = 1;
  int getLink(int v) {
    while (s[sz(s)-d[v].len-2] != s.bk) v = d[v].link;
    return v;
  }
  void updAns() { // serial path has O(log n) vertices
    ans.pb({MOD,MOD});
    for (int v = last; d[v].len > 0; v = d[v].slink) {
      d[v].seriesAns=ans[sz(s)-1-d[d[v].slink].len-d[v].diff];
      if (d[v].diff == d[d[v].link].diff)
        F0R(i,2) ckmin(d[v].seriesAns[i],
              d[d[v].link].seriesAns[i]);
      // start of previous oc of link[v]=start of last oc of v
      F0R (i,2) ckmin(ans.bk[i],d[v].seriesAns[i^1]+1);
    }
  }
  void addChar(char C) {
    s += C; int c = C-'a'; last = getLink(last);
    if (!d[last].to[c]) {
      d.eb(d[last].len+2,d[getLink(d[last].link)].to[c]);
      d[last].to[c] = sz(d)-1;
      auto& z = d.bk; z.diff = z.len-d[z.link].len;
      z.slink = z.diff == d[z.link].diff
        ? d[z.link].slink : z.link;
    } // max suf with different dif
    last = d[last].to[c]; ++d[last].oc;
    updAns();
  }
  void numOc() { ROF(i,2,sz(d)) d[d[i].link].oc += d[i].oc; }
};
```

Trie.h

Description: insert int, query max xor with some int in the trie
Time: $\mathcal{O}(MXBIT)$

cc2950, 23 lines

```
template<int SZ, int MXBIT> struct Trie {
  int nex[SZ][2], sz[SZ], num = 0; // num is last node in trie
  // change 2 to 26 for lowercase letters
  Trie() { memset(nex,0,sizeof nex); memset(sz,0,sizeof sz); }
  void ins(ll x, int a = 1) { // insert or delete
    int cur = 0; sz[cur] += a;
    R0F(i,MXBIT) {
      int t = (x>>i)&1;
      if (!nex[cur][t]) nex[cur][t] = ++num;
      sz[cur = nex[cur][t]] += a;
    }
  }
  ll test(ll x) { // compute max xor
    if (!sz[0]) return -INF; // no elements in trie
    int cur = 0;
    R0F(i,MXBIT) {
      int t = ((x>>i)&1)^1;
      if (!nex[cur][t] || !sz[nex[cur][t]]) t ^= 1;
      cur = nex[cur][t]; if (t) x ^= 1LL<<i;
    }
    return x;
  }
};
```

Heuristics (10)

10.1 Tips

10.1.1 Simulated Annealing

Default temperature schedule for SA:
$t = t_0 * (\frac{t_n}{t_0})^{\text{time\_passed}}$, time_passed $\in [0,1]$ Acceptance function: $RNG() < exp((\text{cur} - \text{new})/t)$ A good schedule should have a slowly decreasing acceptance rate. Priorities:

- Prototype different transitions.
- Optimize eval function (= make it dynamic).
- Find a proper temp schedule. Consider starting with lower temperatures and increasing until it starts getting worse.

Things to do if its easy to get stuck in a local optimum:

- Use a very high temperature/acceptance rate
- Complex transitions
- (rarely) kicks
- (rarely) restarting temp schedule
- (rarely) multiple runs

Things to do when there are lots of invalid states:

- Add an additional scoring component based on number of collisions
- Do many short SA runs with high temp on partial subsolutions

If your eval is very fast, make sure RNG, exp and cur_time are not bottlenecks. When SA is bad:

- Very easy to get stuck in local optima
- Simple transitions are mixed with complex ones – have a separate acceptance function for each transition
- Eval is erratic during transitions

Use a good local tester: make sure it has multithreading and relative scoring. If you are considering whether its worth optimising speed, just run with higher time limit and see if the improvements are worth it. Removing bugs is #1 priority; results are meaningless when bugs are involved. If results do not align with intuition, investigate. Merge N-dimensional coordinates into 1D. Memory allocation is slow.

RNG.h

Description: some random number generators

b87bdf, 36 lines

```
#define INLINE   inline __attribute__ ((always_inline))

struct RNG {
  unsigned int MT[624];
  int index;
  RNG(int seed = 1) {init(seed);}
  void init(int seed = 1) {MT[0] = seed; FOR(i, 1, 624) MT[i] =
       (1812433253UL * (MT[i-1] ^ (MT[i-1] >> 30)) + i); index = 0; }
  void generate() {
    const unsigned int MULT[] = {0, 2567483615UL};
    REP(i, 227) {unsigned int y = (MT[i] & 0x8000000UL) + (MT[i+1] & 0
         x7FFFFFFFUL); MT[i] = MT[i+397] ^ (y >> 1); MT[i] ^= MULT[y
         &1]; }
    FOR(i, 227, 623) {unsigned int y = (MT[i] & 0x8000000UL) + (MT[i
         +1] & 0x7FFFFFFFUL); MT[i] = MT[i-227] ^ (y >> 1); MT[i] ^=
         MULT[y&1]; }
    unsigned int y = (MT[623] & 0x8000000UL) + (MT[0] & 0x7FFFFFFFUL);
          MT[623] = MT[623-227] ^ (y >> 1); MT[623] ^= MULT[y&1];
  }
  unsigned int rand() { if (index == 0) generate(); unsigned int y =
       MT[index]; y ^= y >> 11; y ^= y << 7  & 2636928640UL; y ^= y
       << 15 & 4022730752UL; y ^= y >> 18; index = index == 623 ? 0 :
        index + 1; return y;}
  INLINE int next() {return rand(); }
  INLINE int next(int x) {return rand() % x; }
  INLINE int next(int a, int b) {return a + (rand() % (b - a)); }
  INLINE double next_double() {return (rand() + 0.5) * (1.0 /
       4294967296.0); }
  INLINE double next_double(double a, double b) {return a +
       next_double() * (b - a); }
};

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

static uint64_t x = 1234; // any value
static uint64_t splitmix64() {
  uint64_t z = (x += 0x9e3779b97f4a7c15);
  z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
  z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
  return z ^ (z >> 31);
}
```

SimAnneal.h

Description: template for simulated annealing

<bits/extc++.h>, <sys/time.h> 8b82f3, 137 lines

```
#pragma GCC optimize("Ofast,omit-frame-pointer,inline,unroll-all-loops
     ")

#pragma GCC target("avx2")

using namespace std;
using ll = long long;

#define vt vector
#define f first
#define s second
#define pb push_back
#define all(x) begin(x), end(x)
#define size(x) ((int) (x).size())
#define FOR(i, a, b) for (int i = (a); i < (b); i++)
#define ROF(i, a, b) for (int i = (b) - 1; i >= (a); i--)
#define F0R(i, b) FOR (i, 0, b)
const int inf = 1e9;
const ll INF = 1e18;
using vi = vt<int>;
using db = double;

__gnu_cxx::sfmt19937 mt(random_device{}());

int gen(int l, int r) {
  return uniform_int_distribution<int>(l, r)(mt);
}
db next_double() {
  return uniform_real_distribution<db>(0, 1)(mt);
}

double get_time() {timeval tv; gettimeofday(&tv, NULL); return tv.
     tv_sec + tv.tv_usec * 1e-6;}
double start_time = get_time();
double elapsed() {return get_time() - start_time;}

const db TIME_LIMIT = 0.95;
db t0 = 1e9, tn = 0.1, time_passed = 1e-9;
const db maximise_score = -1; // -1 if minimise

// data

int n;
vector<int> a;

// end data

using T = int;
struct State {
  T value;
  vi a;

  // static transition: consider optimising to dynamic
  void to_neighbour() {
    int i, j;
    switch(gen(0, 4)) {
      case 0:
        i = gen(0, n - 1);
        j = gen(0, n - 2);
        if (i == j) j++;
        swap(a[i], a[j]);
        break;
      case 1:
        a[gen(0, n - 1)] *= -1;
        break;
      case 2:
      case 3:
        i = gen(0, n);
        j = gen(0, n);
        if (i > j) swap(i, j);
        reverse(a.begin() + i, a.begin() + j);
        break;
      case 4:
        i = gen(0, n);
        j = gen(0, n);
        if (i > j) swap(i, j);
        FOR (k, i, j) a[k] *= -1;
        break;
    }
  }

  int calc_value() {
    T ans = 0, sum = 0;
    for (int x : a) sum += x, ans += abs(sum);
    return value = ans;
  }

  void print() {
    F0R (i, n) {
      if (a[i] > 0) cout << a[i] << " a\n";
      else cout << -a[i] << " f\n";
    }
    cout << value << '\n';
  }
};

State cur, best;

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

  cin >> n;
  a.resize(n);
  for (int &x : a) cin >> x;
  sort(all(a));
  F0R (i, n) if (a[i] % 4 == 1 || a[i] % 4 == 2) a[i] *= -1;
  cur.a = a;
  cur.calc_value();
  best = cur;
  t0 = cur.value * 2;

  if (n == 1) {
    cur.print();
    return 0;
  }

  int its = 0;
  db t;
  while (true) {
    if ((its & 511) == 0) {
      time_passed = elapsed() / TIME_LIMIT;
      if (time_passed > 1.0) break;
      t = t0 * pow(tn / t0, time_passed);
    }
    its++;
```

```
    State neighbour = cur;
    neighbour.to_neighbour();
    if ((neighbour.calc_value() - cur.value) * maximise_score >= 0
      || next_double() < exp(((neighbour.value - cur.value) *
          maximise_score) / t)) {
      cur = neighbour;
      if ((cur.value - best.value) * maximise_score > 0) {
        best = cur;
      }
    }
  }
  best.print();
}
```

BeamSearch.h

Description: example beam search solution

<bits/extc++.h>, <sys/time.h> 42c1f1, 187 lines

```
#pragma GCC optimize("Ofast,omit-frame-pointer,inline,unroll-all-loops
    ")

#pragma GCC target("avx2")

using namespace std;
using ll = long long;

#define vt vector
#define f first
#define s second
#define pb push_back
#define all(x) begin(x), end(x)
#define size(x) ((int) (x).size())
#define FOR(i, a, b) for (int i = (a); i < (b); i++)
#define ROF(i, a, b) for (int i = (b) - 1; i >= (a); i--)
#define F0R(i, b) FOR (i, 0, b)
const int inf = 1e9;
const ll INF = 1e18;
using vi = vt<int>;
using db = long double;

__gnu_cxx::sfmt19937 mt(random_device{}());

int gen(int l, int r) {
  return uniform_int_distribution<int>(l, r)(mt);
}

double get_time() {timeval tv; gettimeofday(&tv, NULL); return tv.
    tv_sec + tv.tv_usec * 1e-6;}
double start_time = get_time();
double elapsed() {return get_time() - start_time;}

const db TIME_LIMIT = 5;

// data

int n, k;
vector<int> a;
db ik;

// end data

using T = db;

struct State {
  T value{};
  vt<vi> buckets;
  vt<db> sums;

  void print() {
    F0R (i, k) {
      for (int x : buckets[i]) cout << x << ' ';
      cout << '\n';
    }
    cerr << "value: " << fixed << setprecision(24) << value << '\n';
  }
};

class Transition {
public:
  T new_val{};
  virtual ~Transition() = default;
  virtual State accept() = 0;
  bool operator<(const Transition& oth) const { return new_val < oth.
      new_val; }
};

class Swap : public Transition {
  State& cur;
  int i{}, j{}, el{};
public:
  explicit Swap(State& cur_) : cur(cur_) {
    do { i = gen(0, k - 1); } while (size(cur.buckets[i]) <= 1);

    j = gen(0, k - 2);
    if (j >= i) ++j;

    el = gen(0, size(cur.buckets[i]) - 1);
    int x = cur.buckets[i][el];

    new_val = cur.value
      / powl(cur.sums[i], ik) / powl(cur.sums[j], ik)
      * powl(cur.sums[i] - x, ik) * powl(cur.sums[j] + x, ik);
  }

  State accept() override {
    State nxt = cur;
    int &x = nxt.buckets[i][el];
    nxt.buckets[j].pb(x);
    nxt.sums[i] -= x;
    nxt.sums[j] += x;
    swap(x, nxt.buckets[i].back());
    nxt.buckets[i].pop_back();
    nxt.value = new_val;
    return nxt;
  }
};

class Move : public Transition {
  State& cur;
  int i{}, j{}, xi{}, yi{};
public:
  explicit Move(State& cur_) : cur(cur_) {
    i = gen(0, k - 1);
    j = gen(0, k - 2);
    if (j >= i) ++j;

    xi = gen(0, size(cur.buckets[i]) - 1);
    yi = gen(0, size(cur.buckets[j]) - 1);

    int x = cur.buckets[i][xi];
    int y = cur.buckets[j][yi];

    new_val = cur.value
        / powl(cur.sums[i], ik) / powl(cur.sums[j], ik)
        * powl(cur.sums[i] - x + y, ik) * powl(cur.sums[j] - y + x, ik
            );
  }

  State accept() override {
    State nxt = cur;
    int &x = nxt.buckets[i][xi];
    int &y = nxt.buckets[j][yi];
    nxt.sums[j] += x - y;
    nxt.sums[i] += y - x;
    swap(x, y);
    nxt.value = new_val;
    return nxt;
  }
};

db calc_score(State& state) {
  db val = 1;
  F0R (i, k) val *= powl(state.sums[i], ik);
  return val;
}

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

  cin >> n >> k;
  a.resize(n);
  for (int &x : a) cin >> x;

  State cur;
  cur.buckets.resize(k);
  cur.sums.assign(k, 0);
  F0R (i, n) {
    int j = int(min_element(all(cur.sums)) - cur.sums.begin());
    cur.sums[j] += a[i];
    cur.buckets[j].push_back(a[i]);
  }
  ik = 1.0l / k;
  cur.value = calc_score(cur);

  const int beam_width = 100;
  const int neighbours = 100;

  vector<State> states{cur};
  State best = cur;

  int its = 0;
  while (elapsed() < TIME_LIMIT) {
    its++;
    vector<unique_ptr<Transition>> transitions;
    transitions.reserve(size(states) * neighbours * 2);

    for (auto &st : states) {
      if (st.value > best.value) best = st;
      F0R (_, neighbours) {
        if (k > 1) transitions.emplace_back(make_unique<Swap>(st));
        transitions.emplace_back(make_unique<Move>(st));
      }
    }

    if (transitions.empty()) break;

    sort(all(transitions), [](const auto& a, const auto& b){ return a
        ->new_val > b->new_val; });

    vector<State> new_states;
    int take = min<int>(beam_width, size(transitions));
    new_states.reserve(take);
    F0R (i, take) new_states.push_back(transitions[i]->accept());

    states = std::move(new_states);
  }
```

```
  best.print();
}
```

Various (11)

11.1 Intervals

IntervalContainer.h

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

edce47, 23 lines

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

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

IntervalCover.h

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

9e9d8d, 19 lines

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

ConstantIntervals.h

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

753a4c, 19 lines

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

11.2 Misc. algorithms

TernarySearch.h

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

9155b4, 11 lines

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

LIS.h

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

2932a0, 17 lines

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

FastKnapsack.h

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

b20ccc, 16 lines

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

11.3 Dynamic programming

KnuthDP.h

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

DivideAndConquerDP.h

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

d38d2b, 18 lines

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

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

11.4 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).

11.5 Optimization tricks

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

11.5.1 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.
- `F0R (b, K) F0R (i, (1 << K))`
  `if (i & 1 << b) D[i] += D[i ^ (1 << b)];`
  computes all sums of subsets.

11.5.2 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 ("O0", "trapv")` kills the program on integer overflows (but is really slow).

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;
  }
};
```

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.

7b3c70, 17 lines

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

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

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.

745db2, 8 lines

```
// Either globally or in a single class:
static char buf[450 << 20];
void* operator new(size_t s) {
  static size_t i = sizeof buf;
  assert(s < i);
  return (void*)&buf[i -= s];
}
void operator delete(void*) {}
```

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<int, small<int>>> 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) {}
};
```

Unrolling.h

520e76, 5 lines

```
#define F {...; ++i;}
int i = from;
while (i&3 && i < to) F // for alignment, if needed
while (i + 4 <= to) { F F F F }
while (i < to) F
```

11.6 Other

GrayCode.h

Description: Gray codes

e87165, 10 lines

```
int g(int n) {
  return n ^ (n >> 1);
}
int rev_g (int g) {
  int n = 0;
  for (; g; g >>= 1)
  n ^= g;
  return n;
}
```

Timer.h

Description: Timer object.

<chrono> c65b97, 14 lines

```
using namespace std;

struct Timer {
  chrono::high_resolution_clock::time_point st;

  Timer() { reset(); }
  void reset() { st = chrono::high_resolution_clock::now(); }

  long long elapsed() {
  auto ed = chrono::high_resolution_clock::now();
  return chrono::duration_cast<chrono::milliseconds>(ed - st).count();
  }
  long long operator()() { return elapsed(); }
};
```

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
```

Test session (A)

A.1 To test during test session

- Keyboard layouts
- bashrc
- vimrc
- template.cpp
- Java
- Python
- Printing
- Sending clarifications
- Documentation
- Submit script
- Whitespace/case insensitivity in output
- Return values from main
- Printing to stderr
- Source code limit
- Is it possible to submit and read from non-source files?
- SIMD support on machine resp. judge
- How long do array accesses take on machine resp. judge?
- How long do small operations take on machine resp. judge?
- Does `clock()` work?
- All judge errors:
  - Accepted
  - WA
  - Presentation Error
  - RTE (what results in RTE?)
  - TLE
  - Memory limit, both stack and heap
  - Output limit
  - Illegal function
  - Compile error
  - Compile time limit exceeded
  - Compile memory limit exceeded
  - Too late
- Listing all available binaries

```
echo $PATH | tr ':' ' ' | xargs ls | grep -v /
| sort | uniq | tr '\n' ' ' > path.txt
```