Massachusetts Institute of Technology

MIT $(^w^)$

Benjamin Qi, Spencer Compton, Zhezheng Luo

adapted from KACTL and MIT NULL

2021-10-16

Contest (1)

TemplateShort.cpp

b3b6dc, 53 lines

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

using ll = long long;
using db = long double; // or double if tight TL
using str = string;

using pi = pair<int,int>;
#define mp make_pair
#define f first
#define s second

#define tcT template<class T
tcT> using V = vector<T>;
tcT, size_t SZ> using AR = array<T,SZ>;
using vi = V<int>;
using vb = V<bool>;
using vpi = V<pi>;

#define sz(x) int((x).size())
#define all(x) begin(x), end(x)
#define sor(x) sort(all(x))
#define rsz resize
#define pb push_back
#define ft front()
#define bk back()

#define FOR(i,a,b) for (int i = (a); i < (b); ++i)
#define F0R(i,a) FOR(i,0,a)
#define ROF(i,a,b) for (int i = (b)-1; i >= (a); --i)
#define R0F(i,a) ROF(i,0,a)
#define rep(a) F0R(_,a)
#define each(a,x) for (auto& a: x)

const int MOD = 1e9+7;
const db PI = acos((db)-1);
mt19937 rng(0); // or mt19937_64

tcT> bool ckmin(T& a, const T& b) {
  return b < a ? a = b, 1 : 0; } // set a = min(a,b)
tcT> bool ckmax(T& a, const T& b) {
  return a < b ? a = b, 1 : 0; } // set a = max(a,b)

tcT, class U> T fstTrue(T lo, T hi, U f) {
  ++hi; assert(lo <= hi); // assuming f is increasing
  while (lo < hi) { // find first index such that f is true
    T mid = lo+(hi-lo)/2;
    f(mid) ? hi = mid : lo = mid+1;
  }
  return lo;
}

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

.bashrc

3 lines

```bash
alias clr="printf '\33c'"
co() { g++ -std=c++17 -O2 -Wall -Wextra -Wshadow -Wconversion -
  ↪o $1 $1.cpp; }
run() { co $1 && ./$1; }
```

hash.sh

3 lines

```bash
# Hash file ignoring whitespace and comments. Verifies that
# code was correctly typed. Usage: 'sh hash.sh < A.cpp'
cpp -dD -P -fpreprocessed|tr -d '[:space:]'|md5sum|cut -c-6
```

stress.sh

10 lines

```bash
# 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
```

troubleshoot.txt

75 lines

```
General:
Write down most of your thoughts, even if you're not sure
whether they're useful.
Give your variables (and files) meaningful names.
Stay organized and don't leave papers all over the place!
You should know what your code is doing ...

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?
Remove debug output.
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output as well.
Read the full problem statement again.
Have you understood the problem correctly?
Are you sure your algorithm works?
Try writing a slow (but correct) solution.
Can your algorithm handle the whole range of input?
Did you consider corner cases (ex. n=1)?
Is your output format correct? (including whitespace)
Are you clearing all data structures between test cases?
Any uninitialized variables?
Any undefined behavior (array out of bounds)?
Any overflows or NaNs (or shifting ll by >=64 bits)?
Confusing N and M, i and j, etc.?
Confusing ++i and i++?
Return vs continue vs break?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some test cases 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.
Rewrite your solution from the start or let a teammate do it.

Geometry:
Work with ints if possible.
Correctly account for numbers close to (but not) zero. Related:
for functions like acos make sure absolute val of input is not
(slightly) greater than one.
Correctly deal with vertices that are collinear, concyclic,
coplanar (in 3D), etc.
Subtracting a point from every other (but not itself)?

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's your complexity? Large TL does not mean that something
simple (like NlogN) isn't intended.
Are you copying a lot of unnecessary data? (References)
Avoid vector, map. (use arrays/unordered_map)
How big is the input and output? (consider FastIO)
What do your teammates think about your algorithm?
Calling count() on multiset?

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

Mathematics (2)

2.1 Trigonometry

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

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

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

2.2 Geometry

2.2.1 Triangles

Side lengths: a, b, c

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

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

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

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

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

Length of bisector (divides angles in two):

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

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

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

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

2.3 Derivatives/Integrals

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

Integration by parts:

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

2.4 Sums/Series

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \leq 1)$$
$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots, (-1 \leq x \leq 1)$$
$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, (-\infty < x < \infty)$$
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, (-\infty < x < \infty)$$

Data Structures (3)

3.1 STL

MapComparator.h

Description: example of function object (functor) for map or set

Usage: `set<int,cmp> s; map<int,int,cmp> m;`

5bfa6c, 1 lines

```
struct cmp{bool operator()(int l,int r)const{return l>r;}};
```

HashMap.h

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

Usage: `ht<int,int> h({},{},{},{},{1<<16});`

<ext/pb_ds/assoc_container.hpp> 05a86f, 11 lines

```
using namespace __gnu_pbds;
struct chash {
  const uint64_t C = ll(2e18*PI)+71; // large odd number
  const int RANDOM = rng();
  ll operator()(ll x) const {
    return __builtin_bswap64((x^RANDOM)*C); }
};
template<class K,class V> using um = unordered_map<K,V,chash>;
template<class K,class V> using ht = gp_hash_table<K,V,chash>;
template<class K,class V> V get(ht<K,V>& u, K x) {
  auto it = u.find(x); return it == end(u) ? 0 : it->s; }
```

IndexedSet.h

Description: A set (not multiset!) with support for finding the n'th element, and finding the index of an element. Change `null_type` for map.

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

<ext/pb_ds/assoc_container.hpp> fe2351, 12 lines

```
using namespace __gnu_pbds;
tcT> using Tree = tree<T, null_type, less<T>,
  rb_tree_tag, tree_order_statistics_node_update>;
#define ook order_of_key
#define fbo find_by_order

void treeExample() {
  Tree<int> t, t2; t.insert(8);
  auto it = t.insert(10).f; assert(it == t.lb(9));
  assert(t.ook(10) == 1 && t.ook(11) == 2 && *t.fbo(0) == 8);
  t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

LCold.h

Description: Add lines of the form $ax+b$, query maximum y-coordinate for any x.

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

df7386, 29 lines

```
using T = ll; const T INF = LLONG_MAX; // a/b rounded down
// ll fdiv(ll a, ll b) { return a/b-((a^b)<0&&a%b); }

bool _Q = 0;
struct Line {
  T a, b; mutable T lst;
  T eval(T x) const { return a*x+b; }
  bool operator<(const Line&o)const{return _Q?lst<o.lst:a<o.a;}
  T last_gre(const Line& o) const { assert(a <= o.a);
    // greatest x s.t. a*x+b >= o.a*x+o.b
    return lst=(a==o.a?(b>=o.b?INF:-INF):fdiv(b-o.b,o.a-a));}
};

struct LineContainer: multiset<Line> {
  bool isect(iterator it) { auto n_it = next(it);
    if (n_it == end()) return it->lst = INF, 0;
    return it->last_gre(*n_it) >= n_it->lst; }
  void add(T a, T b) { // remove lines after
    auto it = ins({a,b,0}); while (isect(it)) erase(next(it));
    if (it == begin()) return;
    if (isect(--it)) erase(next(it)), isect(it);
    while (it != begin()) { // remove lines before
      --it; if (it->lst < next(it)->lst) break;
      erase(next(it)); isect(it); }
  }
  T qmax(T x) { assert(!empty());
    _Q = 1; T res = lb({0,0,x})->eval(x); _Q = 0;
    return res; }
};
```

3.2 1D Range Queries

RMQ.h

Description: 1D range minimum query. Can also do queries for any associative operation in $O(1)$ with D&C

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

Time: $\mathcal{O}(1)$

a3f881, 19 lines

```
tcT> struct RMQ {
  int level(int x) { return 31-__builtin_clz(x); }
  V<T> v; V<vi> jmp;
  int cmb(int a, int b) {
    return v[a]==v[b]?min(a,b):(v[a]<v[b]?a:b); }
  void init(const V<T>& _v) {
    v = _v; jmp = {vi(sz(v))};
```

```
    iota(all(jmp[0]),0);
    for (int j = 1; 1<<j <= sz(v); ++j) {
      jmp.pb(vi(sz(v)-(1<<j)+1));
      F0R(i,sz(jmp[j])) jmp[j][i] = cmb(jmp[j-1][i],
        jmp[j-1][i+(1<<(j-1))]);
    }
  }
  int index(int l, int r) {
    assert(l <= r); int d = level(r-l+1);
    return cmb(jmp[d][l],jmp[d][r-(1<<d)+1]); }
  T query(int l, int r) { return v[index(l,r)]; }
};
```

SegTree.h

Description: 1D point update, range query where cmb is any associative operation. If $N = 2^p$ then seg[1]==query(0,N-1).

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

1630f3, 18 lines

```
tcT> struct SegTree { // cmb(ID,b) = b
  const T ID{}; T cmb(T a, T b) { return a+b; }
  int n; V<T> seg;
  void init(int _n) { // upd, query also work if n = _n
    for (n = 1; n < _n; ) n *= 2;
    seg.assign(2*n,ID); }
  void pull(int p) { seg[p] = cmb(seg[2*p],seg[2*p+1]); }
  void upd(int p, T val) { // set val at position p
    seg[p += n] = val; for (p /= 2; p; p /= 2) pull(p); }
  T query(int l, int r) { // associative op on [l, r]
    T ra = ID, rb = ID;
    for (l += n, r += n+1; l < r; l /= 2, r /= 2) {
      if (l&1) ra = cmb(ra,seg[l++]);
      if (r&1) rb = cmb(seg[--r],rb);
    }
    return cmb(ra,rb);
  }
};
```

LazySeg.h

Description: 1D range increment and sum query.

8946fc, 26 lines

```
tcT, int SZ> struct LazySeg {
  static_assert(pct(SZ) == 1); // SZ must be power of 2
  const T ID = 0; T cmb(T a, T b) { return a+b; }
  T seg[2*SZ], lazy[2*SZ];
  LazySeg() { F0R(i,2*SZ) seg[i] = lazy[i] = ID; }
  void push(int ind, int L, int R) {
    seg[ind] += (R-L+1)*lazy[ind]; // dependent on operation
    if (L != R) F0R(i,2) lazy[2*ind+i] += lazy[ind];
    lazy[ind] = 0;
  } // recalc values for current node
  void pull(int ind){seg[ind]=cmb(seg[2*ind],seg[2*ind+1]);}
  void build() { ROF(i,1,SZ) pull(i); }
  void upd(int lo,int hi,T inc,int ind=1,int L=0, int R=SZ-1) {
    push(ind,L,R); if (hi < L || R < lo) return;
    if (lo <= L && R <= hi) {
      lazy[ind] = inc; push(ind,L,R); return; }
    int M = (L+R)/2; upd(lo,hi,inc,2*ind,L,M);
    upd(lo,hi,inc,2*ind+1,M+1,R); pull(ind);
  }
  T query(int lo, int hi, int ind=1, int L=0, int R=SZ-1) {
    push(ind,L,R); if (lo > R || L > hi) return ID;
    if (lo <= L && R <= hi) return seg[ind];
    int M = (L+R)/2; return cmb(query(lo,hi,2*ind,L,M),
      query(lo,hi,2*ind+1,M+1,R));
  }
};
```

PSeg.h

Description: Persistent min segtree with lazy updates, no propagation. If making d a vector then save the results of upd and build in local variables first to avoid issues when vector resizes in C++14 or lower.

Memory: $\mathcal{O}(N + Q \log N)$

8f37fa, 46 lines

```
tcT, int SZ> struct pseg {
  static const int LIM = 2e7;
  struct node {
    int l, r; T val = 0, lazy = 0;
    void inc(T x) { lazy += x; }
    T get() { return val+lazy; }
  };
  node d[LIM]; int nex = 0;
  int copy(int c) { d[nex] = d[c]; return nex++; }
  T cmb(T a, T b) { return min(a,b); }
  void pull(int c) { d[c].val =
    cmb(d[d[c].l].get(), d[d[c].r].get()); }
  //// MAIN FUNCTIONS
  T query(int c, int lo, int hi, int L, int R) {
    if (lo <= L && R <= hi) return d[c].get();
    if (R < lo || hi < L) return MOD;
    int M = (L+R)/2;
    return d[c].lazy+cmb(query(d[c].l,lo,hi,L,M),
              query(d[c].r,lo,hi,M+1,R));
  }
  int upd(int c, int lo, int hi, T v, int L, int R) {
    if (R < lo || hi < L) return c;
    int x = copy(c);
    if (lo <= L && R <= hi) { d[x].inc(v); return x; }
    int M = (L+R)/2;
    d[x].l = upd(d[x].l,lo,hi,v,L,M);
    d[x].r = upd(d[x].r,lo,hi,v,M+1,R);
    pull(x); return x;
  }
  int build(const V<T>& arr, int L, int R) {
    int c = nex++;
    if (L == R) {
      if (L < sz(arr)) d[c].val = arr[L];
      return c;
    }
    int M = (L+R)/2;
    d[c].l = build(arr,L,M), d[c].r = build(arr,M+1,R);
    pull(c); return c;
  }
  vi loc; //// PUBLIC
  void upd(int lo, int hi, T v) {
    loc.pb(upd(loc.bk,lo,hi,v,0,SZ-1)); }
  T query(int ti, int lo, int hi) {
    return query(loc[ti],lo,hi,0,SZ-1); }
  void build(const V<T>&arr) {loc.pb(build(arr,0,SZ-1));}
};
```

Treap.h

Description: Easy BBST. Use split and merge to implement insert and delete.

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

bdb758, 65 lines

```
using pt = struct tnode*;
struct tnode {
  int pri, val; pt c[2]; // essential
  int sz; ll sum; // for range queries
  bool flip = 0; // lazy update
  tnode (int _val) {
    pri = rng(); sum = val = _val;
    sz = 1; c[0] = c[1] = nullptr;
  }
  ~tnode() { F0R(i,2) delete c[i]; }
};
int getsz(pt x) { return x?x->sz:0; }
ll getsum(pt x) { return x?x->sum:0; }
pt prop(pt x) { // lazy propagation
  if (!x || !x->flip) return x;
  swap(x->c[0],x->c[1]);
  x->flip = 0; F0R(i,2) if (x->c[i]) x->c[i]->flip ^= 1;
  return x;
}
pt calc(pt x) {
  pt a = x->c[0], b = x->c[1];
  assert(!x->flip); prop(a), prop(b);
  x->sz = 1+getsz(a)+getsz(b);
  x->sum = x->val+getsum(a)+getsum(b);
  return x;
}
void tour(pt x, vi& v) { // print values of nodes,
  if (!x) return; // inorder traversal
  prop(x); tour(x->c[0],v); v.pb(x->val); tour(x->c[1],v);
}
pair<pt,pt> split(pt t, int v) { // >= v goes to the right
  if (!t) return {t,t};
  prop(t);
  if (t->val >= v) {
    auto p = split(t->c[0], v); t->c[0] = p.s;
    return {p.f,calc(t)};
  } else {
    auto p = split(t->c[1], v); t->c[1] = p.f;
    return {calc(t),p.s};
  }
}
pair<pt,pt> splitsz(pt t, int sz) { // sz nodes go to left
  if (!t) return {t,t};
  prop(t);
  if (getsz(t->c[0]) >= sz) {
    auto p = splitsz(t->c[0],sz); t->c[0] = p.s;
    return {p.f,calc(t)};
  } else {
    auto p=splitsz(t->c[1],sz-getsz(t->c[0])-1); t->c[1]=p.f;
    return {calc(t),p.s};
  }
}
pt merge(pt l, pt r) { //  keys in l < keys in r
  if (!l || !r) return l?:r;
  prop(l), prop(r); pt t;
  if (l->pri > r->pri) l->c[1] = merge(l->c[1],r), t = l;
  else r->c[0] = merge(l,r->c[0]), t = r;
  return calc(t);
}
pt ins(pt x, int v) { // insert v
  auto a = split(x,v), b = split(a.s,v+1);
  return merge(a.f,merge(new tnode(v),b.s)); }
pt del(pt x, int v) { // delete v
  auto a = split(x,v), b = split(a.s,v+1);
  return merge(a.f,b.s); }
```

Number Theory (4)

4.1 Modular Arithmetic

ModIntShort.h

Description: Modular arithmetic.

Usage: mi a = MOD+5; cout << (int)inv(a); // 400000003

63d209, 20 lines

```
struct mi { // WARNING: needs some adjustment to work with FFT
  int v; explicit operator int() const { return v; }
  mi():v(0) {}
  mi(ll _v):v(int(_v%MOD)) { v += (v<0)*MOD; }
```

```
};
mi& operator+=(mi& a, mi b) {
  if ((a.v += b.v) >= MOD) a.v -= MOD;
  return a; }
mi& operator-=(mi& a, mi b) {
  if ((a.v -= b.v) < 0) a.v += MOD;
  return a; }
mi operator+(mi a, mi b) { return a += b; }
mi operator-(mi a, mi b) { return a -= b; }
mi operator*(mi a, mi b) { return mi((ll)a.v*b.v); }
mi& operator*=(mi& a, mi b) { return a = a*b; }
mi pow(mi a, ll p) { assert(p >= 0); // won't work for negative
   ↪ p
  return p==0?1:pow(a*a,p/2)*(p&1?a:1); }
mi inv(mi a) { assert(a.v != 0); return pow(a,MOD-2); }
mi operator/(mi a, mi b) { return a*inv(b); }
bool operator==(mi a, mi b) { return a.v == b.v; }
```

ModFact.h

Description: pre-compute factorial mod inverses, assumes MOD is prime and $SZ < MOD$.
Time: $\mathcal{O}(SZ)$ 6ce1a9, 10 lines

```
vmi invs, fac, ifac;
void genFac(int SZ) {
  invs.rsz(SZ), fac.rsz(SZ), ifac.rsz(SZ);
  invs[1] = fac[0] = ifac[0] = 1;
  FOR(i,2,SZ) invs[i] = mi(-(ll)MOD/i*(int)invs[MOD%i]);
  FOR(i,1,SZ) fac[i] = fac[i-1]*i, ifac[i] = ifac[i-1]*invs[i];
}
mi comb(int a, int b) {
  if (a < b || b < 0) return 0;
  return fac[a]*ifac[b]*ifac[a-b]; }
```

ModMulLL.h

Description: Multiply two 64-bit integers mod another if 128-bit is not available. modMul is equivalent to `(ul)(__int128(a)*b%mod)`. Works for $0 \le a, b < mod < 2^{63}$.
530181, 9 lines

```
using ul = uint64_t;
ul modMul(ul a, ul b, const ul mod) {
  ll ret = a*b-mod*(ul)((db)a*b/mod);
  return ret+((ret<0)-(ret>=(ll)mod))*mod; }
ul modPow(ul a, ul b, const ul mod) {
  if (b == 0) return 1;
  ul res = modPow(a,b/2,mod); res = modMul(res,res,mod);
  return b&1 ? modMul(res,a,mod) : res;
}
```

FastMod.h

Description: Barrett reduction computes $a\%b$ about 4 times faster than usual where $b > 1$ is constant but not known at compile time. Division by b is replaced by multiplication by m and shifting right 64 bits. aa19c9, 7 lines

```
using ul = uint64_t; using L = __uint128_t;
struct FastMod {
  ul b, m; FastMod(ul b) : b(b), m(-1ULL / b) {}
  ul reduce(ul a) {
    ul q = (ul)((__uint128_t(m) * a) >> 64), r = a - q * b;
    return r - (r >= b) * b; }
};
```

ModSqrt.h

Description: square root of integer mod a prime, -1 if doesn't exist.
Time: $\mathcal{O}(\log^2(MOD))$
"ModInt.h" 4ece48, 14 lines

```
using T = int;
T sqrt(mi a) {
  mi p = pow(a,(MOD-1)/2);
  if (p.v != 1) return p.v == 0 ? 0 : -1;
  T s = MOD-1; int r = 0; while (s%2 == 0) s /= 2, ++r;
  mi n = 1; while (pow(n,(MOD-1)/2).v != 1) n = T(n)+1;
  // n non-square, ord(g)=2^r, ord(b)=2^m, ord(g)=2^r, m<r
  for (mi x = pow(a,(s+1)/2), b = pow(a,s), g = pow(n,s);;) {
    if (b.v == 1) return min(x.v,MOD-x.v); // x^2=ab
    int m = 0; for (mi t = b; t.v != 1; t *= t) ++m;
    rep(r-m-1) g *= g; // ord(g)=2^{m+1}
    x *= g, g *= g, b *= g, r = m; // ord(g)=2^m, ord(b)<2^m
  }
}
```

ModSum.h

Description: Counts # of lattice points (x, y) s.t. $1 \le x, 1 \le y, ax+by \le S$ and related quantities.
Time: $\mathcal{O}(\log ab)$ 8965e0, 18 lines

```
using ul = uint64_t;
ul sum2(ul n) { return n/2*((n-1)|1); } // sum(0..n-1)
// \return |{(x,y) | 1 <= x, 1 <= y, a*x+b*y <= S}|
//        = sum_{i=1}^{qs} (S-a*i)/b
ul triSum(ul a, ul b, ul s) { assert(a > 0 && b > 0);
    ul qs = s/a, rs = s%a; // ans = sum_{i=0}^{qs-1}(i*a+rs)/b
    ul ad = a/b*sum2(qs)+rs/b*qs; a %= b, rs %= b;
    return ad+(a?triSum(b,a,a*qs+rs):0); // reduce if a >= b
} // then swap x and y axes and recurse

// \return sum_{x=0}^{n-1} (a*x+b)/m
//        = |{(x,y) | 0 < m*y <= a*x+b < a*n+b}|
ul divSum(ul n, ul a, ul b, ul m) { assert(m > 0);
    return a == 0 ? b/m*n : triSum(m,a,a*n+b); }
// \return sum_{x=0}^{n-1} (a*x+b)%m
ll modSum(ul n, ll a, ll b, ul m) { assert(m > 0);
    a = (a%m+m)%m, b = (b%m+m)%m;
    return a*sum2(n)+b*n-m*divSum(n,a,b,m); }
```

4.2 Primality

4.2.1 Primes

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

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

4.2.2 Divisors

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

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

Dirichlet Convolution: Given a function $f(x)$, let $(f * g)(x) = \sum_{d|x} g(d)f(x/d)$.

If the partial sums $s_{f*g}(n), s_g(n)$ can be computed in $O(1)$ and $s_f(1 \dots n^{2/3})$ can be computed in $O\left(n^{2/3}\right)$ then all $s_f\left(\frac{n}{d}\right)$ can as well. Use

$$s_{f*g}(n) = \sum_{d=1}^{n} g(d)s_f(n/d).$$

If $f(x) = \mu(x)$ then $g(x) = 1, (f * g)(x) = (x == 1)$, and $s_f(n) = 1 - \sum_{i=2}^{n} s_f(n/i)$.

If $f(x) = \phi(x)$ then $g(x) = 1, (f * g)(x) = x$, and $s_f(n) = \frac{n(n+1)}{2} - \sum_{i=2}^{n} s_f(n/i)$.

Sieve.h

Description: Tests primality up to SZ. Runs faster if only odd indices are stored.
Time: $\mathcal{O}(SZ \log\log SZ)$ or $\mathcal{O}(SZ)$ 41c6ed, 20 lines

```
template<int SZ> struct Sieve {
  bitset<SZ> is_prime; vi primes;
  Sieve() {
    is_prime.set(); is_prime[0] = is_prime[1] = 0;
    for (int i = 4; i < SZ; i += 2) is_prime[i] = 0;
    for (int i = 3; i*i < SZ; i += 2) if (is_prime[i])
      for (int j = i*i; j < SZ; j += i*2) is_prime[j] = 0;
    F0R(i,SZ) if (is_prime[i]) primes.pb(i);
  }
  // int sp[SZ]{}; // smallest prime that divides
  // Sieve() { // above is faster
  //   FOR(i,2,SZ) {
  //     if (sp[i] == 0) sp[i] = i, primes.pb(i);
  //     for (int p: primes) {
  //       if (p > sp[i] || i*p >= SZ) break;
  //       sp[i*p] = p;
  //     }
  //   }
  // }
};
```

MultiplicativePrefix.h

Description: Prefix sums of some multiplicative functions. Solve for all square-free numbers in range, then fix.
Time: faster than $\mathcal{O}\left(N^{2/3}\right)$?
"Sieve.h" 2c8704, 22 lines

```
ll pre0(ll n, int i) { // prod(primes) in factorization of x
  ll res = n*(n+1)/2; // assume all square-free
  for (;;++i) {
    ll p = S.pr[i], nn = n/p/p; if (!nn) break;
    for(ll coef=p*(p-1);nn;nn/=p)res-=coef*pre0(nn,i+1);
  }
  return res;
}
ll pre1(ll n, int i) { // gcd of x and arithmetic derivative
  // p^e contributes p^e if e%p == 0 and p^{e-1} otherwise
  ll res = n; // assume all square-free
  for (;;++i) {
    ll p = S.pr[i], nn = n/p/p; if (!nn) break;
    ll lst = 1, mul = p*p;
    for (int e = 2; nn; mul *= p, nn /= p, ++e) {
      ll nex = mul; if (e%p) nex /= p;
      if (lst != nex) res += (nex-lst)*pre1(nn,i+1);
      lst = nex;
```

```
    }
  }
  return res;
}
```

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;
}
```

MillerRabin.h

Description: Deterministic primality test, works up to 2^{64}. For larger numbers, extend A randomly.

"ModMulLL.h" 89df33, 11 lines

```
bool prime(ul n) { // not ll!
  if (n < 2 || n % 6 % 4 != 1) return n-2 < 2;
  ul A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
      s = __builtin_ctzll(n-1), d = n>>s;
  each(a,A) {   // ^ count trailing zeroes
    ul p = modPow(a,d,n), i = s;
    while (p != 1 && p != n-1 && a%n && i--) p = modMul(p,p,n);
    if (p != n-1 && i != s) return 0;
  }
  return 1;
}
```

FactorFast.h

Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).

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

"MillerRabin.h", "ModMulLL.h" 99cf33, 16 lines

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

4.3 Euclidean Algorithm

FracInterval.h

Description: Given fractions $a < b$ with non-negative numerators and denominators, finds fraction f with lowest denominator such that $a < f < b$. Should work with all numbers less than 2^{62}.

1860f3, 6 lines

```
pl bet(pl a, pl b) {
  ll num = a.f/a.s; a.f -= num*a.s, b.f -= num*b.s;
  if (b.f > b.s) return {1+num,1};
  auto x = bet({b.s,b.f},{a.s,a.f});
  return {x.s+num*x.f,x.f};
}
```

Euclid.h

Description: Generalized Euclidean algorithm. euclid and invGeneral work for $A, B < 2^{62}$. minBetween assumes that $0 \le L \le R < B$, works for $AB < 2^{62}$ (same for min_rem)

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

5ddb26, 49 lines

```
// ceil(a/b)
// ll cdiv(ll a, ll b) { return a/b+((a^b)>0&&a%b); }
pl euclid(ll A, ll B) { // For A,B>=0, finds (x,y) s.t.
  // Ax+By=gcd(A,B), |Ax|,|By|<=AB/gcd(A,B)
  if (!B) return {1,0};
  pl p = euclid(B,A%B); return {p.s,p.f-A/B*p.s}; }
ll invGeneral(ll A, ll B) { // find x in [0,B) such that Ax=1
  ↪mod B
  pl p = euclid(A,B); assert(p.f*A+p.s*B == 1);
  return p.f+(p.f<0)*B; } // must have gcd(A,B)=1
ll minBetween(ll A, ll B, ll L, ll R) {
  // min x s.t. exists y s.t. L <= A*x-B*y <= R
  A %= B;
  if (L == 0) return 0;
  if (A == 0) return -1;
  ll k = cdiv(L,A); if (A*k <= R) return k;
  ll x = minBetween(B,A,A-R%A,A-L%A); // min x s.t. exists y
  // s.t. -R <= Bx-Ay <= -L
  return x == -1 ? x : cdiv(B*x+L,A); // solve for y
}

// find min((Ax+C)%B) for 0 <= x <= M
// aka find minimum non-negative value of A*x-B*y+C
// where 0 <= x <= M, 0 <= y
ll minRemainder(ll A, ll B, ll C, ll M) {
  assert(A >= 0 && B > 0 && C >= 0 && M >= 0);
  A %= B, C %= B; ckmin(M,B-1);
  if (A == 0) return C;
  if (C >= A) { // make sure C<A
    ll ad = cdiv(B-C,A);
    M -= ad; if (M < 0) return C;
    C += ad*A-B;
  }
  ll q = B/A, new_B = B%A; // new_B < A
  if (new_B == 0) return C; // B-q*A

  // now minimize A*x-new_B*y+C
  // where 0 <= x,y and x+q*y <= M, 0 <= C < new_B < A
  // q*y -> C-new_B*y
  if (C/new_B > M/q) return C-M/q*new_B;
  M -= C/new_B*q; C %= new_B; // now C < new_B

  // given y, we can compute x = ceil[((B-q*A)*y-C)/A]
  // so x+q*y = ceil((B*y-C)/A) <= M
  ll max_Y = (M*A+C)/B; // must have y <= max_Y
  ll max_X = cdiv(new_B*max_Y-C,A); // must have x <= max_X
  if (max_X*A-new_B*max_Y+C >= new_B) --max_X;
  // now we can remove upper bound on y
  return minRemainder(A,new_B,C,max_X);
}
```

CRT.h

Description: Chinese Remainder Theorem. $a.f \pmod{a.s}, b.f \pmod{b.s}$ $\implies$? $\pmod{\text{lcm}(a.s, b.s)}$. Should work for $ab < 2^{62}$.

"Euclid.h" 574c0e, 10 lines

```
pl CRT(pl a, pl b) {
  if (a.s < b.s) swap(a,b);
  ll x,y; tie(x,y) = euclid(a.s,b.s);
  ll g = a.s*x+b.s*y, l = a.s/g*b.s;
  if ((b.f-a.f)%g) return {-1,-1}; // no solution
  // ?*a.s+a.f \equiv b.f \pmod{b.s}
  // ?=(b.f-a.f)/g*(a.s/g)^{-1} \pmod{b.s/g}
  x = (b.f-a.f)%b.s*x%b.s/g*a.s+a.f;
  return {x+(x<0)*l,l};
}
```

4.4 Pythagorean Triples

The Pythagorean triples are uniquely generated by

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

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

4.5 Lifting the Exponent

For $n > 0$, p prime, and ints x, y s.t. $p \nmid x, y$ and $p | x - y$:

- $p \neq 2$ or $p = 2, 4 | x - y$ $\implies v_p(x^n - y^n) = v_p(x - y) + v_p(n)$.
- $p = 2, 2 | n \implies v_2(x^n - y^n) = v_2((x^2)^{n/2} - (y^2)^{n/2})$.

Combinatorial (5)

5.1 Permutations

5.1.1 Cycles

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

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

5.1.2 Burnside's lemma

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

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

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

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

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

5.2 Partitions and subsets

5.2.1 Partition function

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

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

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

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

5.2.2 Lucas' Theorem

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

5.3 General purpose numbers

5.3.1 Bernoulli numbers

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

Sums of powers:

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

Euler-Maclaurin formula for infinite sums:

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

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

5.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

5.3.3 Eulerian numbers

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

$$E(n,k) = (n-k)E(n-1,k-1) + (k+1)E(n-1,k)$$

$$E(n,0) = E(n,n-1) = 1$$

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

5.3.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

$$S(n,k) = S(n-1,k-1) + kS(n-1,k)$$

$$S(n,1) = S(n,n) = 1$$

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

5.3.5 Bell numbers

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

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

5.3.6 Labeled unrooted trees

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

5.3.7 Catalan numbers

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

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

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

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

5.4 Young Tableaux

Let a **Young diagram** have shape $\lambda = (\lambda_1 \geq \cdots \geq \lambda_k)$, where λ_i equals the number of cells in the i-th (left-justified) row from the top. A **Young tableau** of shape λ is a filling of the $n = \sum \lambda_i$ cells with a permutation of $1 \ldots n$ such that each row and column is increasing.

Hook-Length Formula: For the cell in position (i,j), let $h_\lambda(i,j) = |\{(I,J) | i \leq I, j \leq J, (I = i \text{ or } J = j)\}|$. The number of Young tableaux of shape λ is equal to $f^\lambda = \frac{n!}{\prod h_\lambda(i,j)}$.

Schensted's Algorithm: converts a permutation σ of length n into a pair of Young Tableaux $(S(\sigma), T(\sigma))$ of the same shape. When inserting $x = \sigma_i$,

1. Add x to the first row of S by inserting x in place of the largest y with $x < y$. If y doesn't exist, push x to the end of the row, set the value of T at that position to be i, and stop.

2. Add y to the second row using the same rule, keep repeating as necessary.

All pairs $(S(\sigma), T(\sigma))$ of the same shape correspond to a unique σ, so $n! = \sum (f^\lambda)^2$. Also, $S(\sigma^R) = S(\sigma)^T$.

Let $d_k(\sigma), a_k(\sigma)$ be the lengths of the longest subseqs which are a union of k decreasing/ascending subseqs, respectively. Then $a_k(\sigma) = \sum_{i=1}^{k} \lambda_i, d_k(\sigma) = \sum_{i=1}^{k} \lambda_i^*$, where λ_i^* is size of the i-th column.

5.5 Other

DeBruijnSeq.h

Description: Given alphabet $[0, k)$ constructs a cyclic string of length k^n that contains every length n string as substr.

a6961b, 13 lines

```
vi deBruijnSeq(int k, int n) {
  if (k == 1) return {0};
  vi seq, aux(n+1);
  function<void(int,int)> gen = [&](int t, int p) {
    if (t > n) { // +lyndon word of len p
      if (n%p == 0) FOR(i,1,p+1) seq.pb(aux[i]);
    } else {
      aux[t] = aux[t-p]; gen(t+1,p);
      while (++aux[t] < k) gen(t+1,t);
    }
  };
  gen(1,1); return seq;
}
```

NimProduct.h

Description: Product of nimbers is associative, commutative, and distributive over addition (xor). Forms finite field of size 2^{2^k}. Defined by $ab = \text{mex}(\{a'b + ab' + a'b' : a' < a, b' < b\})$. Application: Given 1D coin turning games G_1, G_2 $G_1 \times G_2$ is the 2D coin turning game defined as follows. If turning coins at $x_1, x_2, \ldots, x_m$ is legal in G_1 and $y_1, y_2, \ldots, y_n$ is legal in G_2, then turning coins at all positions (x_i, y_j) is legal assuming that the coin at (x_m, y_n) goes from heads to tails. Then the grundy function $g(x, y)$ of $G_1 \times G_2$ is $g_1(x) \times g_2(y)$.
Time: 64^2 xors per multiplication, memorize to speed up.

5afe17, 46 lines

```
using ul = uint64_t;
struct Precalc {
  ul tmp[64][64], y[8][8][256];
  unsigned char x[256][256];
  Precalc() { // small nim products, all < 256
    F0R(i,256) F0R(j,256) x[i][j] = mult<8>(i,j);
    F0R(i,8) F0R(j,i+1) F0R(k,256)
      y[i][j][k] = mult<64>(prod2(8*i,8*j),k);
  }
  ul prod2(int i, int j) { // nim prod of 2^i, 2^j
    ul& u = tmp[i][j]; if (u) return u;
    if (!(i&j)) return u = 1ULL<<(i|j);
    int a = (i&j)&-(i&j); // a=2^k, consider 2^{2^k}
    return u=prod2(i^a,j)^prod2((i^a)|(a-1),(j^a)|(i&(a-1)));
    // 2^{2^k}*2^{2^k} = 2^{2^k}+2^{2^k-1}
  } // 2^{2^i}*2^{2^j} = 2^{2^i+2^j} if i<j
  template<int L> ul mult(ul a, ul b) {
    ul c = 0; F0R(i,L) if (a>>i&1)
      F0R(j,L) if (b>>j&1) c ^= prod2(i,j);
    return c;
  }
  // 2^{8*i}*(a>>(8*i)&255) * 2^{8*j}*(b>>(8*j)&255)
  // -> (2^{8*i}*2^{8*j})*((a>>(8*i)&255)*(b>>(8*j)&255))
  ul multFast(ul a, ul b) const { // faster nim product
    ul res = 0; auto f=[](ul c,int d) {return c>>(8*d)&255;};
    F0R(i,8) {
      F0R(j,i) res ^= y[i][j][x[f(a,i)][f(b,j)]
              ^x[f(a,j)][f(b,i)]];
      res ^= y[i][i][x[f(a,i)][f(b,i)]];
    }
    return res;
  }
};
const Precalc P;

struct nb { // nimber
  ul x; nb() { x = 0; }
  nb(ul _x): x(_x) {}
  explicit operator ul() { return x; }
  nb operator+(nb y) { return nb(x^y.x); }
  nb operator*(nb y) { return nb(P.multFast(x,y.x)); }
  friend nb pow(nb b, ul p) {
    nb res = 1; for (;p;p/=2,b=b*b) if (p&1) res = res*b;
    return res; } // b^{2^{2^A}-1}=1 where 2^{2^A} > b
  friend nb inv(nb b) { return pow(b,-2); }
};
```

MatroidIsect.h

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

"DSU.h" d0051c, 51 lines

```
struct Gmat { // graphic matroid
  int V = 0; vpi ed; DSU D;
  Gmat(vpi _ed):ed(_ed) {
    map<int,int> m; each(t,ed) m[t.f] = m[t.s] = 0;
    each(t,m) t.s = V++;
    each(t,ed) t.f = m[t.f], t.s = m[t.s];
  }
  void clear() { D.init(V); }
  void ins(int i) { assert(D.unite(ed[i].f,ed[i].s)); }
  bool indep(int i) { return !D.sameSet(ed[i].f,ed[i].s); }
};
struct Cmat { // colorful matroid
  int C = 0; vi col; V<bool> used;
  Cmat(vi col):col(col) {each(t,col) ckmax(C,t+1); }
  void clear() { used.assign(C,0); }
  void ins(int i) { used[col[i]] = 1; }
  bool indep(int i) { return !used[col[i]]; }
};
template<class M1, class M2> struct MatroidIsect {
  int n; V<bool> iset; M1 m1; M2 m2;
  bool augment() {
    vi pre(n+1,-1); queue<int> q({n});
    while (sz(q)) {
      int x = q.ft; q.pop();
      if (iset[x]) {
        m1.clear(); F0R(i,n) if (iset[i] && i != x) m1.ins(i);
        F0R(i,n) if (!iset[i] && pre[i] == -1 && m1.indep(i))
          pre[i] = x, q.push(i);
      } else {
        auto backE = [&]() { // back edge
          m2.clear();
          F0R(c,2)F0R(i,n)if((x==i||iset[i])&&(pre[i]==-1)==c){
            if (!m2.indep(i))return c?pre[i]=x,q.push(i),i:-1;
            m2.ins(i); }
          return n;
        };
        for (int y; (y = backE()) != -1;) if (y == n) {
          for(; x != n; x = pre[x]) iset[x] = !iset[x];
          return 1; }
      }
    }
    return 0;
  }
  MatroidIsect(int n, M1 m1, M2 m2):n(n), m1(m1), m2(m2) {
    iset.assign(n+1,0); iset[n] = 1;
    m1.clear(); m2.clear(); // greedily add to basis
    R0F(i,n) if (m1.indep(i) && m2.indep(i))
      iset[i] = 1, m1.ins(i), m2.ins(i);
    while (augment());
  }
};
```

Numerical (6)

6.1 Matrix

Matrix.h

Description: 2D matrix operations.

"ModInt.h" c2e27f, 34 lines

```
using T = mi;
using Mat = V<V<T>>; // use array instead if tight TL

Mat makeMat(int r, int c) { return Mat(r,V<T>(c)); }
Mat makeId(int n) {
  Mat m = makeMat(n,n);
  F0R(i,n) m[i][i] = 1;
  return m;
}
Mat& operator+=(Mat& a, const Mat& b) {
  assert(sz(a) == sz(b) && sz(a[0]) == sz(b[0]));
  F0R(i,sz(a)) F0R(j,sz(a[0])) a[i][j] += b[i][j];
  return a;
}
Mat& operator-=(Mat& a, const Mat& b) {
  assert(sz(a) == sz(b) && sz(a[0]) == sz(b[0]));
  F0R(i,sz(a)) F0R(j,sz(a[0])) a[i][j] -= b[i][j];
  return a;
}
Mat operator*(const Mat& a, const Mat& b) {
  int x = sz(a), y = sz(a[0]), z = sz(b[0]);
  assert(y == sz(b)); Mat c = makeMat(x,z);
  F0R(i,x) F0R(j,y) F0R(k,z) c[i][k] += a[i][j]*b[j][k];
  return c;
}
Mat operator+(Mat a, const Mat& b) { return a += b; }
Mat operator-(Mat a, const Mat& b) { return a -= b; }
Mat& operator*=(Mat& a, const Mat& b) { return a = a*b; }
Mat pow(Mat m, ll p) {
  int n = sz(m); assert(n == sz(m[0]) && p >= 0);
  Mat res = makeId(n);
  for (; p; p /= 2, m *= m) if (p&1) res *= m;
  return res;
}
```

MatrixInv.h

Description: Uses gaussian elimination to convert into reduced row echelon form and calculates determinant. For determinant via arbitrary modulos, use a modified form of the Euclidean algorithm because modular inverse may not exist. If you have computed $A^{-1} \pmod{p^k}$, then the inverse $\pmod{p^{2k}}$ is $A^{-1}(2I - AA^{-1})$.
Time: $\mathcal{O}\left(N^3\right)$, determinant of 1000×1000 matrix of modints in 1 second if you reduce # of operations by half

73ec43, 38 lines

```
const db EPS = 1e-9; // adjust?
int getRow(V<V<db>>& m, int R, int i, int nex) {
    pair<db,int> bes{0,-1}; // find row with max abs value
    FOR(j,nex,R) ckmax(bes,{abs(m[j][i]),j});
    return bes.f < EPS ? -1 : bes.s; }
```

```
int getRow(V<vmi>& m, int R, int i, int nex) {
    FOR(j,nex,R) if (m[j][i] != 0) return j;
    return -1; }
pair<T,int> gauss(Mat& m) { // convert to reduced row echelon
   ↪form
    if (!sz(m)) return {1,0};
    int R = sz(m), C = sz(m[0]), rank = 0, nex = 0;
    T prod = 1; // determinant
    F0R(i,C) {
        int row = getRow(m,R,i,nex);
        if (row == -1) { prod = 0; continue; }
        if (row != nex) prod *= -1, swap(m[row],m[nex]);
        prod *= m[nex][i]; rank++;
        T x = 1/m[nex][i]; FOR(k,i,C) m[nex][k] *= x;
        F0R(j,R) if (j != nex) {
            T v = m[j][i]; if (v == 0) continue;
            FOR(k,i,C) m[j][k] -= v*m[nex][k];
        }
        nex++;
    }
    return {prod,rank};
}
Mat inv(Mat m) {
    int R = sz(m); assert(R == sz(m[0]));
    Mat x = makeMat(R,2*R);
    F0R(i,R) {
        x[i][i+R] = 1;
        F0R(j,R) x[i][j] = m[i][j];
    }
    if (gauss(x).s != R) return Mat();
    Mat res = makeMat(R,R);
    F0R(i,R) F0R(j,R) res[i][j] = x[i][j+R];
    return res;
}
```

MatrixTree.h

Description: Kirchhoff's Matrix Tree Theorem. Given adjacency matrix, calculates # of spanning trees.

"MatrixInv.h" 066e59, 11 lines

```
T numSpan(Mat m) {
  int n = sz(m); Mat res(n-1,n-1);
  F0R(i,n) FOR(j,i+1,n) {
    mi ed = m[i][j]; res[i][i] += ed;
    if (j != n-1) {
      res[j][j] += ed;
      res[i][j] -= ed, res[j][i] -= ed;
    }
  }
  return gauss(res).f;
}
```

ShermanMorrison.h

Description: Calculates $(A+uv^T)^{-1}$ given $B=A^{-1}$. Not invertible if sum=0.

"MatrixInv.h" 3a3f34, 7 lines

```
void ad(Mat& B, const V<T>& u, const V<T>& v) {
  int n = sz(A); V<T> x(n), y(n);
  F0R(i,n) F0R(j,n)
    x[i] += B[i][j]*u[j], y[j] += v[i]*B[i][j];
  T sum = 1; F0R(i,n) F0R(j,n) sum += v[i]*B[i][j]*u[j];
  F0R(i,n) F0R(j,n) B[i][j] -= x[i]*y[j]/sum;
}
```

6.2 Polynomials

Poly.h

Description: Basic poly ops including division. Can replace `T` with double, complex.

"ModInt.h" 44d0dd, 73 lines

```
using T = mi; using poly = V<T>;
void remz(poly& p) { while (sz(p)&&p.bk==T(0)) p.pop_back(); }
poly REMZ(poly p) { remz(p); return p; }
poly rev(poly p) { reverse(all(p)); return p; }
poly shift(poly p, int x) {
  if (x >= 0) p.insert(begin(p),x,0);
  else assert(sz(p)+x >= 0), p.erase(begin(p),begin(p)-x);
  return p;
}
poly RSZ(const poly& p, int x) {
  if (x <= sz(p)) return poly(begin(p),begin(p)+x);
  poly q = p; q.rsz(x); return q;  }
T eval(const poly& p, T x) { // evaluate at point x
  T res = 0; R0F(i,sz(p)) res = x*res+p[i];
  return res; }
poly dif(const poly& p) { // differentiate
  poly res; FOR(i,1,sz(p)) res.pb(T(i)*p[i]);
  return res; }
poly integ(const poly& p) { // integrate
  static poly invs{0,1};
  for (int i = sz(invs); i <= sz(p); ++i)
    invs.pb(-MOD/i*invs[MOD%i]);
  poly res(sz(p)+1); F0R(i,sz(p)) res[i+1] = p[i]*invs[i+1];
  return res;
}

poly& operator+=(poly& l, const poly& r) {
  l.rsz(max(sz(l),sz(r))); F0R(i,sz(r)) l[i] += r[i];
  return l; }
poly& operator-=(poly& l, const poly& r) {
  l.rsz(max(sz(l),sz(r))); F0R(i,sz(r)) l[i] -= r[i];
  return l; }
poly& operator*=(poly& l, const T& r) { each(t,l) t *= r;
  return l;  }
poly& operator/=(poly& l, const T& r) { each(t,l) t /= r;
  return l; }
poly operator+(poly l, const poly& r) { return l += r; }
poly operator-(poly l, const poly& r) { return l -= r; }
poly operator-(poly l) { each(t,l) t *= -1; return l; }
poly operator*(poly l, const T& r) { return l *= r; }
poly operator*(const T& r, const poly& l) { return l*r; }
poly operator/(poly l, const T& r) { return l /= r;  }
poly operator*(const poly& l, const poly& r) {
  if (!min(sz(l),sz(r))) return {};
  poly x(sz(l)+sz(r)-1);
  F0R(i,sz(l)) F0R(j,sz(r)) x[i+j] += l[i]*r[j];
  return x;
}
poly& operator*=(poly& l, const poly& r) { return l = l*r; }

pair<poly,poly> quoRem(poly a, poly b) {
  remz(a); remz(b); assert(sz(b));
  T lst = b.bk, B = T(1)/lst; each(t,a) t *= B;
  each(t,b) t *= B;
  poly q(max(sz(a)-sz(b)+1,0));
  for (int dif; (dif=sz(a)-sz(b)) >= 0; remz(a)) {
    q[dif] = a.bk; F0R(i,sz(b)) a[i+dif] -= q[dif]*b[i]; }
  each(t,a) t *= lst;
  return {q,a}; // quotient, remainder
}
poly operator%(const poly& a, const poly& b) {
  return quoRem(a,b).s; }
T resultant(poly a, poly b) { // R(A,B)
  // =b_m^n*prod_{j=1}^mA(mu_j)
  // =b_m^na_n^m*prod_{i=1}^nprod_{j=1}^m(mu_j-lambda_i)
  // =(-1)^{mn}a_n^m*prod_{i=1}^nB(lambda_i)
  // =(-1)^{nm}R(B,A)
  // Also, R(A,B)=b_m^{deg(A)-deg(A-CB)}R(A-CB,B)
  int ad = sz(a)-1, bd = sz(b)-1;
  if (bd <= 0) return bd < 0 ? 0 : pow(b.bk,ad);
  int pw = ad; a = a%b; pw -= (ad = sz(a)-1);
  return resultant(b,a)*pow(b.bk,pw)*T((bd&ad&1)?-1:1);
}
```

PolyInterpolate.h

Description: n points determine unique polynomial of degree $\leq n-1$. For numerical precision pick $v[k].f = c * \cos(k/(n-1)*\pi), k = 0 \ldots n-1$.
Time: $\mathcal{O}\left(n^2\right)$

"Poly.h" aada3a, 8 lines

```
poly interpolate(V<pair<T,T>> v) {
  poly res, tmp{1};
  F0R(i,sz(v)) { T prod = 1; // add one point at a time
    F0R(j,i) v[i].s -= prod*v[j].s, prod *= v[i].f-v[j].f;
    v[i].s /= prod; res += v[i].s*tmp; tmp *= poly{-v[i].f,1};
  } // add multiple of (x-v[0].f)*(x-v[1].f)*...*(x-v[i-1].f)
  return res;
}
```

FFT.h

Description: Multiply polynomials of ints for any modulus $< 2^{31}$. For XOR convolution ignore `m` within `fft`.
Time: $\mathcal{O}(N \log N)$

"ModInt.h" 9d4a1a, 41 lines

```
// const int MOD = 998244353;
tcT> void fft(V<T>& A, bool inv = 0) { // NTT
  int n = sz(A); assert((T::mod-1)%n == 0); V<T> B(n);
  for(int b = n/2; b; b /= 2, swap(A,B)) { // w = n/b'th root
    T w = pow(T::rt(),(T::mod-1)/n*b), m = 1;
    for(int i = 0; i < n; i += b*2, m *= w) F0R(j,b) {
      T u = A[i+j], v = A[i+j+b]*m;
      B[i/2+j] = u+v; B[i/2+j+n/2] = u-v;
    }
  }
  if (inv) { reverse(1+all(A));
    T z = T(1)/T(n); each(t,A) t *= z; }
} // for NTT-able moduli
tcT> V<T> mul(V<T> A, V<T> B) {
  if (!min(sz(A),sz(B))) return {};
  int s = sz(A)+sz(B)-1, n = 1; for (; n < s; n *= 2);
  bool eq = A == B; A.rsz(n), fft(A);
  if (eq) B = A; // squaring A, reuse result
  else B.rsz(n), fft(B);
  F0R(i,n) A[i] *= B[i];
  fft(A,1); A.rsz(s); return A;
}
template<class M, class T> V<M> mulMod(V<T> x, V<T> y) {
  auto con = [](const V<T>& v) {
    V<M> w(sz(v)); F0R(i,sz(v)) w[i] = (int)v[i];
    return w; };
  return mul(con(x),con(y));
} // arbitrary moduli
tcT> V<T> MUL(const V<T>& A, const V<T>& B) {
  using m0 = mint<(119<<23)+1,62>; auto c0 = mulMod<m0>(A,B);
  using m1 = mint<(5<<25)+1,  62>; auto c1 = mulMod<m1>(A,B);
  using m2 = mint<(7<<26)+1,  62>; auto c2 = mulMod<m2>(A,B);
  int n = sz(c0); V<T> res(n); m1 r01 = 1/m1(m0::mod);
  m2 r02 = 1/m2(m0::mod), r12 = 1/m2(m1::mod);
  F0R(i,n) { // a=remainder mod m0::mod, b fixes it mod m1::mod
    int a = c0[i].v, b = ((c1[i]-a)*r01).v,
```

```
    c = (((c2[i]-a)*r02-b)*r12).v;
    res[i] = (T(c)*m1::mod+b)*m0::mod+a; // c fixes m2::mod
  }
  return res;
}
```

PolyInvSimpler.h

Description: computes A^{-1} such that $AA^{-1} \equiv 1 \pmod{x^n}$. Newton's method: If you want $F(x) = 0$ and $F(Q_k) \equiv 0 \pmod{x^a}$ then $Q_{k+1} = Q_k - \frac{F(Q_k)}{F'(Q_k)} \pmod{x^{2a}}$ satisfies $F(Q_{k+1}) \equiv 0 \pmod{x^{2a}}$. Application: if $f(n), g(n)$ are the #s of forests and trees on n nodes then $\sum_{n=0}^{\infty} f(n)x^n = \exp\left(\sum_{n=1}^{\infty} \frac{g(n)}{n!}\right)$.
Usage: `vmi v={1,5,2,3,4}; ps(exp(2*log(v,9),9)); // squares v`
Time: $\mathcal{O}(N \log N)$

"PolyConv.h" 2b368f, 37 lines

```
poly inv(poly A, int n) { // Q-(1/Q-A)/(-Q^{-2})
  poly B{1/A[0]};
  for (int x = 2; x/2 < n; x *= 2)
    B = 2*B-RSZ(conv(RSZ(A,x),conv(B,B)),x);
  return RSZ(B,n);
}
poly sqrt(const poly& A, int n) {  // Q-(Q^2-A)/(2Q)
  assert(A[0] == 1); poly B{1};
  for (int x = 2; x/2 < n; x *= 2)
    B = T(1)/T(2)*RSZ(B+conv(RSZ(A,x),inv(B,x)),x);
  return RSZ(B,n);
}
// return {quotient, remainder}
pair<poly,poly> divi(const poly& f, const poly& g) {
  if (sz(f) < sz(g)) return {{},f};
  poly q = conv(inv(rev(g),sz(f)-sz(g)+1),rev(f));
  q = rev(RSZ(q,sz(f)-sz(g)+1));
  poly r = RSZ(f-conv(q,g),sz(g)-1); return {q,r};
}
poly log(poly A, int n) { assert(A[0] == 1); // (ln A)' = A'/A
  A.rsz(n); return integ(RSZ(conv(dif(A),inv(A,n-1)),n-1)); }
poly exp(poly A, int n) { assert(A[0] == 0);
  poly B{1}, IB{1};
  for (int x = 1; x < n; x *= 2) {
    IB = 2*IB-RSZ(conv(B,conv(IB,IB)),x); // inverse of B to x
      ↪places
    poly Q = dif(RSZ(A,x)); Q += RSZ(conv(IB,dif(B)-conv(B,Q))
      ↪,2*x-1);
    // first x-1 terms of dif(B)-conv(B,Q) are zero
    B = B+RSZ(conv(B,RSZ(A,2*x)-integ(Q)),2*x);
  } // We know that Q=A' is B'/B to x-1 places, we want to find
    ↪ B'/B to 2x-1 places
  return RSZ(B,n);
}
// poly expOld(poly A, int n) { // Q-(lnQ-A)/(1/Q)
//  assert(A[0] == 0); poly B = {1};
//  while (sz(B) < n) { int x = 2*sz(B);
//    B = RSZ(B+conv(B,RSZ(A,x)-log(B,x)),x); }
//  return RSZ(B,n);
// }
```

6.3 Misc

LinRec.h

Description: Berlekamp-Massey, computes linear recurrence `C` of order N for sequence `s` of $2N$ terms.
Usage: `LinRec L; L.init({0,1,1,2,3,5,8}); // Fibonacci`
Time: init ⇒ $\mathcal{O}(N|C|)$, eval ⇒ $\mathcal{O}\left(|C|^2 \log p\right)$.

"Poly.h" e63980, 32 lines

```
struct LinRec {
  poly s, C, rC;
  void BM() { // find smallest C such that C[0]=1 and
    // for all i >= sz(C)-1, sum_{j=0}^{sz(C)-1}C[j]*s[i-j]=0
    // If we treat C and s as polynomials in D, then
    // for all i >= sz(C)-1, [D^i]C*s=0
    int x = 0; T b = 1;
    poly B; B = C = {1}; // B is fail vector
    F0R(i,sz(s)) { // update C after adding a term of s
      ++x; int L = sz(C), M = i+3-L;
      T d = 0; F0R(j,L) d += C[j]*s[i-j]; // [D^i]C*s
      if (d == 0) continue; // [D^i]C*s=0
      poly _C = C; T coef = d/b;
      C.rsz(max(L,M)); F0R(j,sz(B)) C[j+x] -= coef*B[j];
      if (L < M) B = _C, b = d, x = 0;
    }
  }
  void init(const poly& _s) {
    s = _s; BM();
    rC = C; reverse(all(rC)); // poly for getPow
    C.erase(begin(C)); each(t,C) t *= -1;
  } // now s[i]=sum_{j=0}^{sz(C)-1}C[j]*s[i-j-1]
  poly getPow(ll p) { // get x^p mod rC
    if (p == 0) return {1};
    poly r = getPow(p/2); r = (r*r)%rC;
    return p&1?(r*poly{0,1})%rC:r;
  }
  T dot(poly v) { // dot product with seq
    T ans = 0; F0R(i,sz(v)) ans += v[i]*s[i];
    return ans; } // get p-th term of rec
  T eval(ll p) { assert(p >= 0); return dot(getPow(p)); }
};
```

Integrate.h

Description: Integration of a function over an interval using Simpson's rule, exact for polynomials of degree up to 3. The error should be proportional to dif^4, although in practice you will want to verify that the result is stable to desired precision when epsilon changes.

3ebaab, 7 lines

```
// db f(db x) { return x*x+3*x+1; }
template<class F> db quad(F f, db a, db b) {
  const int n = 1000;
  db dif = (b-a)/2/n, tot = f(a)+f(b);
  FOR(i,1,2*n) tot += f(a+i*dif)*(i&1?4:2);
  return tot*dif/3;
}
```

IntegrateAdaptive.h

Description: Unused. Fast integration using adaptive Simpson's rule, exact for polynomials of degree up to 5.
Usage:
```
db z, y;
db h(db x) { return x*x + y*y + z*z <= 1; }
db g(db y) { ::y = y; return quad(h, -1, 1); }
db f(db z) { ::z = z; return quad(g, -1, 1); }
db sphereVol = quad(f,-1,1), pi = sphereVol*3/4;
```

3b316e, 10 lines

```
template<class F> db simpson(F f, db a, db b) {
  db c = (a+b)/2; return (f(a)+4*f(c)+f(b))*(b-a)/6; }
template<class F> db rec(F& f, db a, db b, db eps, db S) {
  db c = (a+b)/2;
  db S1 = simpson(f,a,c), S2 = simpson(f,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> db quad(F f, db a, db b, db eps = 1e-8) {
  return rec(f,a,b,eps,simpson(f,a,b)); }
```

Simplex.h

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

c99f9c, 67 lines

```
using T = db; // double probably suffices
using vd = V<T>; using vvd = V<vd>;
const T eps = 1e-8, inf = 1/.0;

#define ltj(X) if (s==-1 || mp(X[j],N[j])<mp(X[s],N[s])) s=j
struct LPSolver {
  int m, n; // # m = contraints, # n = variables
  vi N, B; // N[j] = non-basic variable (j-th column), = 0
  vvd D; // B[j] = basic variable (j-th row)
  LPSolver(const vvd& A, const vd& b, const vd& c) :
    m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)) {
    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];
    // B[i]: basic variable for each constraint
    // D[i][n]: artificial variable for testing feasibility
    F0R(j,n) N[j] = j, D[m][j] = -c[j];
    // D[m] stores negation of objective,
    // which we want to minimize
    N[n] = -1; D[m+1][n] = 1; // to find initial feasible
  } // solution, minimize artificial variable
  void pivot(int r, int s) { // swap B[r] (row)
    T inv = 1/D[r][s]; // with N[r] (column)
    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) { // if phase=1, ignore artificial variable
      int s = -1; F0R(j,n+1) if (N[j] != -phase) ltj(D[x]);
      // find most negative col for nonbasic (NB) variable
      if (D[x][s] >= -eps) return 1;
      // can't get better sol by increasing NB variable
      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;
        // find smallest positive ratio
      } // -> max increase in NB variable
      if (r == -1) return 0; // objective is unbounded
      pivot(r,s);
    }
  }
  T solve(vd& x) { // 1. check if x=0 feasible
    int r = 0; FOR(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] < -eps) { // if not, find feasible start
      pivot(r,n); // make artificial variable basic
      assert(simplex(2)); // I think this will always be true??
      if (D[m+1][n+1] < -eps) return -inf;
      // D[m+1][n+1] is max possible value of the negation of
      // artificial variable, optimal value should be zero
```

```
    // if exists feasible solution
    F0R(i,m) if (B[i] == -1) { // artificial var basic
      int s = 0; FOR(j,1,n+1) ltj(D[i]); // -> nonbasic
      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;
 }
};
```

Graphs (7)

Erdos-Gallai: $d_1 \geq \cdots \geq d_n$ can be degree sequence of simple graph on n vertices iff their sum is even and $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k), \forall 1 \leq k \leq n$.

7.1 Cycles

DirectedCycle.h

Description: skack

3504a5, 17 lines

```
template<int SZ> struct DirCyc {
  vi adj[SZ], stk, cyc; vb inStk, vis;
  void dfs(int x) {
    stk.pb(x); inStk[x] = vis[x] = 1;
    each(i,adj[x]) {
      if (inStk[i]) cyc = {find(all(stk),i),end(stk)};
      else if (!vis[i]) dfs(i);
      if (sz(cyc)) return;
    }
    stk.pop_back(); inStk[x] = 0;
  }
  vi init(int N) {
    inStk.rsz(N), vis.rsz(N);
    F0R(i,N) if (!vis[i] && !sz(cyc)) dfs(i);
    return cyc;
  }
};
```

NegativeCycle.h

Description: use Bellman-Ford (make sure no underflow)

688ec8, 11 lines

```
vi negCyc(int N, V<pair<pi,int>> ed) {
  vl d(N); vi p(N); int x = -1;
  rep(N) {
    x = -1; each(t,ed) if (ckmin(d[t.f.s],d[t.f.f]+t.s))
      p[t.f.s] = t.f.f, x = t.f.s;
    if (x == -1) return {};
  }
  rep(N) x = p[x]; // enter cycle
  vi cyc{x}; while (p[cyc.bk] != x) cyc.pb(p[cyc.bk]);
  reverse(all(cyc)); return cyc;
}
```

7.2 DSU

DSU.h

Description: Disjoint Set Union with path compression and union by size. Add edges and test connectivity. Use for Kruskal's or Boruvka's minimum spanning tree.
Time: $\mathcal{O}(\alpha(N))$

e42a83, 11 lines

```
struct DSU {
  vi e; void init(int N) { e = vi(N,-1); }
  int get(int x) { return e[x] < 0 ? x : e[x] = get(e[x]); }
  bool sameSet(int a, int b) { return get(a) == get(b); }
  int size(int x) { return -e[get(x)]; }
  bool unite(int x, int y) { // union by size
    x = get(x), y = get(y); if (x == y) return 0;
    if (e[x] > e[y]) swap(x,y);
    e[x] += e[y]; e[y] = x; return 1;
  }
};
```

7.3 Trees

LCAjump.h

Description: Calculates least common ancestor in tree with verts $0 \ldots N-1$ and root R using binary jumping.
Memory: $\mathcal{O}(N \log N)$
Time: $\mathcal{O}(N \log N)$ build, $\mathcal{O}(\log N)$ query

6b0ee9, 28 lines

```
struct LCA {
  int N; V<vi> par, adj; vi depth;
  void init(int _N) {  N = _N;
    int d = 1; while ((1<<d) < N) ++d;
    par.assign(d,vi(N)); adj.rsz(N); depth.rsz(N);
  }
  void ae(int x, int y) { adj[x].pb(y), adj[y].pb(x); }
  void gen(int R = 0) { par[0][R] = R; dfs(R); }
  void dfs(int x = 0) {
    FOR(i,1,sz(par)) par[i][x] = par[i-1][par[i-1][x]];
    each(y,adj[x]) if (y != par[0][x])
      depth[y] = depth[par[0][y]=x]+1, dfs(y);
  }
  int jmp(int x, int d) {
    F0R(i,sz(par)) if ((d>>i)&1) x = par[i][x];
    return x; }
  int lca(int x, int y) {
    if (depth[x] < depth[y]) swap(x,y);
    x = jmp(x,depth[x]-depth[y]); if (x == y) return x;
    R0F(i,sz(par)) {
      int X = par[i][x], Y = par[i][y];
      if (X != Y) x = X, y = Y;
    }
    return par[0][x];
  }
  int dist(int x, int y) { // # edges on path
    return depth[x]+depth[y]-2*depth[lca(x,y)]; }
};
```

LCArmq.h

Description: Euler Tour LCA. Compress takes a subset S of nodes and computes the minimal subtree that contains all the nodes pairwise LCAs and compressing edges. Returns a list of `(par, orig_index)` representing a tree rooted at 0. The root points to itself.
Time: $\mathcal{O}(N \log N)$ build, $\mathcal{O}(1)$ LCA, $\mathcal{O}(|S| \log |S|)$ compress

"RMQ.h" e5a035, 28 lines

```
struct LCA {
  int N; V<vi> adj;
  vi depth, pos, par, rev; // rev is for compress
  vpi tmp; RMQ<pi> r;
  void init(int _N) { N = _N; adj.rsz(N);
    depth = pos = par = rev = vi(N); }
  void ae(int x, int y) { adj[x].pb(y), adj[y].pb(x); }
  void dfs(int x) {
    pos[x] = sz(tmp); tmp.eb(depth[x],x);
    each(y,adj[x]) if (y != par[x]) {
      depth[y] = depth[par[y]=x]+1, dfs(y);
      tmp.eb(depth[x],x); }
  }
  void gen(int R = 0) { par[R] = R; dfs(R); r.init(tmp); }
  int lca(int u, int v){
    u = pos[u], v = pos[v]; if (u > v) swap(u,v);
    return r.query(u,v).s; }
  int dist(int u, int v) {
    return depth[u]+depth[v]-2*depth[lca(u,v)]; }
  vpi compress(vi S) {
    auto cmp = [&](int a, int b) { return pos[a] < pos[b]; };
    sort(all(S),cmp); R0F(i,sz(S)-1) S.pb(lca(S[i],S[i+1]));
    sort(all(S),cmp); S.erase(unique(all(S)),end(S));
    vpi ret{{0,S[0]}}; F0R(i,sz(S)) rev[S[i]] = i;
    FOR(i,1,sz(S)) ret.eb(rev[lca(S[i-1],S[i])],S[i]);
    return ret;
  }
};
```

HLD.h

Description: Heavy-Light Decomposition, add val to verts and query sum in path/subtree.
Time: any tree path is split into $\mathcal{O}(\log N)$ parts

"LazySeg.h" 1802e2, 48 lines

```
template<int SZ, bool VALS_IN_EDGES> struct HLD {
  int N; vi adj[SZ];
  int par[SZ], root[SZ], depth[SZ], sz[SZ], ti;
  int pos[SZ]; vi rpos; // rpos not used but could be useful
  void ae(int x, int y) { adj[x].pb(y), adj[y].pb(x); }
  void dfsSz(int x) {
    sz[x] = 1;
    each(y,adj[x]) {
      par[y] = x; depth[y] = depth[x]+1;
      adj[y].erase(find(all(adj[y]),x));
      dfsSz(y); sz[x] += sz[y];
      if (sz[y] > sz[adj[x][0]]) swap(y,adj[x][0]);
    }
  }
  void dfsHld(int x) {
    pos[x] = ti++; rpos.pb(x);
    each(y,adj[x]) {
      root[y] = (y == adj[x][0] ? root[x] : y);
      dfsHld(y); }
  }
  void init(int _N, int R = 0) { N = _N;
    par[R] = depth[R] = ti = 0; dfsSz(R);
    root[R] = R; dfsHld(R);
  }
  int lca(int x, int y) {
    for (; root[x] != root[y]; y = par[root[y]])
      if (depth[root[x]] > depth[root[y]]) swap(x,y);
    return depth[x] < depth[y] ? x : y;
  }
  LazySeg<ll,SZ> tree; // segtree for sum
  template <class BinaryOp>
  void processPath(int x, int y, BinaryOp op) {
    for (; root[x] != root[y]; y = par[root[y]]) {
      if (depth[root[x]] > depth[root[y]]) swap(x,y);
      op(pos[root[y]],pos[y]); }
    if (depth[x] > depth[y]) swap(x,y);
    op(pos[x]+VALS_IN_EDGES,pos[y]);
  }
  void modifyPath(int x, int y, int v) {
    processPath(x,y,[this,&v](int l, int r) {
      tree.upd(l,r,v); }); }
  ll queryPath(int x, int y) {
    ll res = 0; processPath(x,y,[this,&res](int l, int r) {
      res += tree.query(l,r); });
    return res; }
```

```
  void modifySubtree(int x, int v) {
    tree.upd(pos[x]+VALS_IN_EDGES,pos[x]+sz[x]-1,v); }
};
```

Centroid.h

Description: The centroid of a tree of size N is a vertex such that after removing it, all resulting subtrees have size at most $\frac{N}{2}$. Supports updates in the form "add 1 to all verts v such that $dist(x, v) \leq y$."
Memory: $\mathcal{O}(N \log N)$
Time: $\mathcal{O}(N \log N)$ build, $\mathcal{O}(\log N)$ update and query

907e21, 54 lines

```
void ad(vi& a, int b) { ckmin(b,sz(a)-1); if (b>=0) a[b]++; }
void prop(vi& a) { R0F(i,sz(a)-1) a[i] += a[i+1]; }
template<int SZ> struct Centroid {
  vi adj[SZ]; void ae(int a,int b){adj[a].pb(b),adj[b].pb(a);}
  bool done[SZ]; // processed as centroid yet
  int N,sub[SZ],cen[SZ],lev[SZ]; // subtree size, centroid anc
  int dist[32-__builtin_clz(SZ)][SZ]; // dists to all ancs
  vi stor[SZ], STOR[SZ];
  void dfs(int x, int p) { sub[x] = 1;
    each(y,adj[x]) if (!done[y] && y != p)
      dfs(y,x), sub[x] += sub[y];
  }
  int centroid(int x) {
    dfs(x,-1);
    for (int sz = sub[x];;) {
      pi mx = {0,0};
      each(y,adj[x]) if (!done[y] && sub[y] < sub[x])
        ckmax(mx,{sub[y],y});
      if (mx.f*2 <= sz) return x;
      x = mx.s;
    }
  }
  void genDist(int x, int p, int lev) {
    dist[lev][x] = dist[lev][p]+1;
    each(y,adj[x]) if (!done[y] && y != p) genDist(y,x,lev); }
  void gen(int CEN, int _x) { // CEN = centroid above x
    int x = centroid(_x); done[x] = 1; cen[x] = CEN;
    sub[x] = sub[_x]; lev[x] = (CEN == -1 ? 0 : lev[CEN]+1);
    dist[lev[x]][x] = 0;
    stor[x].rsz(sub[x]),STOR[x].rsz(sub[x]+1);
    each(y,adj[x]) if (!done[y]) genDist(y,x,lev[x]);
    each(y,adj[x]) if (!done[y]) gen(x,y);
  }
  void init(int _N) { N = _N; FOR(i,1,N+1) done[i] = 0;
    gen(-1,1); } // start at vert 1
  void upd(int x, int y) {
    int cur = x, pre = -1;
    R0F(i,lev[x]+1) {
      ad(stor[cur],y-dist[i][x]);
      if (pre != -1) ad(STOR[pre],y-dist[i][x]);
      if (i > 0) pre = cur, cur = cen[cur];
    }
  } // call propAll() after all updates
  void propAll() { FOR(i,1,N+1) prop(stor[i]), prop(STOR[i]); }
  int query(int x) { // get value at vertex x
    int cur = x, pre = -1, ans = 0;
    R0F(i,lev[x]+1) { // if pre != -1, subtract those from
      ans += stor[cur][dist[i][x]]; // same subtree
      if (pre != -1) ans -= STOR[pre][dist[i][x]];
      if (i > 0) pre = cur, cur = cen[cur];
    }
    return ans;
  }
};
```

7.3.1 SqrtDecompton

HLD generally suffices. If not, here are some common strategies:

- Rebuild the tree after every $\sqrt{N}$ queries.
- Consider vertices with $>$ or $<$ $\sqrt{N}$ degree separately.
- For subtree updates, note that there are $O(\sqrt{N})$ distinct sizes among child subtrees of any node.

Block Tree: Use a DFS to split edges into contiguous groups of size $\sqrt{N}$ to $2\sqrt{N}$.

Mo's Algorithm for Tree Paths: Maintain an array of vertices where each one appears twice, once when a DFS enters the vertex (`st`) and one when the DFS exists (`en`). For a tree path $u \leftrightarrow v$ such that `st[u]<st[v]`,

- If u is an ancestor of v, query `[st[u],st[v]]`.
- Otherwise, query `[en[u],st[v]]` and consider $LCA(u, v)$ separately.

Solutions with worse complexities can be faster if you optimize the operations that are performed most frequently. Use arrays instead of vectors whenever possible. Iterating over an array in order is faster than iterating through the same array in some other order (ex. one given by a random permutation) or DFSing on a tree of the same size. Also, the difference between $\sqrt{N}$ and the optimal block (or buffer) size can be quite large. Try up to 5x smaller or larger (at least).

7.4 DFS Algorithms

EulerPath.h

Description: Eulerian path starting at `src` if it exists, visits all edges exactly once. Works for both directed and undirected. Returns vector of {vertex,label of edge to vertex}. Second element of first pair is always -1.
Time: $\mathcal{O}(N + M)$

9c222d, 23 lines

```
template<bool directed> struct Euler {
    int N; V<vpi> adj; V<vpi::iterator> its; vb used;
    void init(int _N) { N = _N; adj.rsz(N); }
    void ae(int a, int b) {
        int M = sz(used); used.pb(0);
        adj[a].eb(b,M); if (!directed) adj[b].eb(a,M); }
    vpi solve(int src = 0) {
        its.rsz(N); F0R(i,N) its[i] = begin(adj[i]);
        vpi ans, s{{src,-1}}; // {{vert,prev vert},edge label}
        int lst = -1; // ans generated in reverse order
        while (sz(s)) {
            int x = s.bk.f; auto& it=its[x], en=end(adj[x]);
            while (it != en && used[it->s]) ++it;
            if (it == en) { // no more edges out of vertex
                if (lst != -1 && lst != x) return {};
                // not a path, no tour exists
                ans.pb(s.bk); s.pop_back(); if (sz(s)) lst=s.bk
                    ↪.f;
            } else s.pb(*it), used[it->s] = 1;
        } // must use all edges
        if (sz(ans) != sz(used)+1) return {};
        reverse(all(ans)); return ans;
    }
};
```

SCCT.h

Description: Tarjan's, DFS once to generate strongly connected components in topological order. a, b in same component if both $a \to b$ and $b \to a$ exist. Uses less memory than Kosaraju b/c doesn't store reverse edges.
Time: $\mathcal{O}(N + M)$

a36e0c, 22 lines

```
struct SCC {
  int N, ti = 0; V<vi> adj;
  vi disc, comp, stk, comps;
  void init(int _N){ N = _N, adj.rsz(N);
    disc.rsz(N), comp.rsz(N,-1);}
  void ae(int x, int y) { adj[x].pb(y); }
  int dfs(int x) {
    int low = disc[x] = ++ti; stk.pb(x);
    each(y,adj[x]) if (comp[y] == -1) // comp[y] == -1,
      ckmin(low,disc[y]?:dfs(y)); // disc[y] != 0 -> in stack
    if (low == disc[x]) { // make new SCC
      // pop off stack until you find x
      comps.pb(x); for (int y = -1; y != x;)
        comp[y = stk.bk] = x, stk.pop_back();
    }
    return low;
  }
  void gen() {
    F0R(i,N) if (!disc[i]) dfs(i);
    reverse(all(comps));
  }
};
```

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

"SCC.h" 805e1c, 32 lines

```
struct TwoSAT {
  int N = 0; vpi edges;
  void init(int _N) { N = _N; }
  int addVar() { return N++; } // for atMostOne
  void either(int x, int y) {
    x = max(2*x,-1-2*x), y = max(2*y,-1-2*y);
    edges.eb(x,y); }
  void implies(int x, int y) { either(~x,y); }
  void must(int x) { either(x,x); }
  void atMostOne(const vi& li) {
    if (sz(li) <= 1) return;
    int cur = ~li[0];
    FOR(i,2,sz(li)) {
      int next = addVar();
      either(cur,~li[i]); either(cur,next);
      either(~li[i],next); cur = ~next;
    }
```

```
    either(cur,~li[1]);
  }
  vb solve(int _N = -1) {
    if (_N != -1) N = _N;
    SCC S; S.init(2*N);
    each(e,edges) S.ae(e.f^1,e.s), S.ae(e.s^1,e.f);
    S.gen(); reverse(all(S.comps)); // reverse topo order
    for (int i = 0; i < 2*N; i += 2)
      if (S.comp[i] == S.comp[i^1]) return {};
    vi tmp(2*N); each(i,S.comps) if (!tmp[i])
      tmp[i] = 1, tmp[S.comp[i^1]] = -1;
    vb ans(N); F0R(i,N) ans[i] = tmp[S.comp[2*i]] == 1;
    return ans;
  }
};
```

BCC.h

Description: Biconnected components of edges. Removing any vertex in BCC doesn't disconnect it. To get block-cut tree, create a bipartite graph with the original vertices on the left and a vertex for each BCC on the right. Draw edge $u \leftrightarrow v$ if u is contained within the BCC for v. Self-loops are not included in any BCC while BCCS of size 1 represent bridges.
Time: $\mathcal{O}(N+M)$

0625a6, 35 lines

```
struct BCC {
  V<vpi> adj; vpi ed;
  V<vi> edgeSets, vertSets; // edges for each bcc
  int N, ti = 0; vi disc, stk;
  void init(int _N) { N = _N; disc.rsz(N), adj.rsz(N); }
  void ae(int x, int y) {
    adj[x].eb(y,sz(ed)), adj[y].eb(x,sz(ed)), ed.eb(x,y); }
  int dfs(int x, int p = -1) { // return lowest disc
    int low = disc[x] = ++ti;
    each(e,adj[x]) if (e.s != p) {
      if (!disc[e.f]) {
        stk.pb(e.s); // disc[x] < LOW -> bridge
        int LOW = dfs(e.f,e.s); ckmin(low,LOW);
        if (disc[x] <= LOW) { // get edges in bcc
          edgeSets.eb(); vi& tmp = edgeSets.bk; // new bcc
          for (int y = -1; y != e.s; )
            tmp.pb(y = stk.bk), stk.pop_back();
        }
      } else if (disc[e.f] < disc[x]) // back-edge
        ckmin(low,disc[e.f]), stk.pb(e.s);
    }
    return low;
  }
  void gen() {
    F0R(i,N) if (!disc[i]) dfs(i);
    vb in(N);
    each(c,edgeSets) { // edges contained within each BCC
      vertSets.eb(); // so you can easily create block cut tree
      auto ad = [&](int x) {
        if (!in[x]) in[x] = 1, vertSets.bk.pb(x); };
      each(e,c) ad(ed[e].f), ad(ed[e].s);
      each(e,c) in[ed[e].f] = in[ed[e].s] = 0;
    }
  }
};
```

MaximalCliques.h

Description: Used only once. Finds all maximal cliques.
Time: $\mathcal{O}\left(3^{N/3}\right)$

f5cd93, 16 lines

```
using B = bitset<128>; B adj[128];
int N;
// possibly in clique, not in clique, in clique
void cliques(B P = ~B(), B X={}, B R={}) {
  if (!P.any()) {
    if (!X.any()) // do smth with R
    return;
  }
  int q = (P|X)._Find_first();
  // clique must contain q or non-neighbor of q
  B cands = P&~adj[q];
  F0R(i,N) if (cands[i]) {
    R[i] = 1; cliques(P&adj[i],X&adj[i],R);
    R[i] = P[i] = 0; X[i] = 1;
  }
}
```

7.5 Flows

Konig's Theorem: In a bipartite graph, max matching = min vertex cover.

Dilworth's Theorem: For any partially ordered set, the sizes of the max antichain and of the min chain decomposition are equal. Equivalent to Konig's theorem on the bipartite graph (U, V, E) where $U = V = S$ and (u, v) is an edge when $u < v$. Those vertices outside the min vertex cover in both U and V form a max antichain.

Dinic.h

Description: Fast flow. After computing flow, edges $\{u, v\}$ such that $lev[u] \neq 0$, $lev[v] = 0$ are part of min cut.
Time: $\mathcal{O}\left(N^2 M\right)$ flow

c76643, 43 lines

```
template<class F> struct Dinic {
  struct Edge { int to, rev; F cap; };
  int N; V<V<Edge>> adj;
  void init(int _N) { N = _N; adj.rsz(N); }
  pi ae(int a, int b, F cap, F rcap = 0) {
    assert(min(cap,rcap) >= 0); // saved me > once
    adj[a].pb({b,sz(adj[b]),cap});
    adj[b].pb({a,sz(adj[a])-1,rcap});
    return {a,sz(adj[a])-1};
  }
  F edgeFlow(pi loc) { // get flow along original edge
    const Edge& e = adj.at(loc.f).at(loc.s);
    return adj.at(e.to).at(e.rev).cap;
  }
  vi lev, ptr;
  bool bfs(int s, int t) { // level=shortest dist from source
    lev = ptr = vi(N);
    lev[s] = 1; queue<int> q({s});
    while (sz(q)) { int u = q.ft; q.pop();
      each(e,adj[u]) if (e.cap && !lev[e.to]) {
        q.push(e.to), lev[e.to] = lev[u]+1;
        if (e.to == t) return 1;
      }
    }
    return 0;
  }
  F dfs(int v, int t, F flo) {
    if (v == t) return flo;
    for (int& i = ptr[v]; i < sz(adj[v]); i++) {
      Edge& e = adj[v][i];
      if (lev[e.to]!=lev[v]+1||!e.cap) continue;
      if (F df = dfs(e.to,t,min(flo,e.cap))) {
        e.cap -= df; adj[e.to][e.rev].cap += df;
        return df; } // saturated >=1 one edge
      }
    return 0;
  }
  F maxFlow(int s, int t) {
    F tot = 0; while (bfs(s,t)) while (F df =
      dfs(s,t,numeric_limits<F>::max())) tot += df;
    return tot;
  }
};
```

GomoryHu.h

Description: Returns edges of Gomory-Hu tree (second element is weight). Max flow between pair of vertices of undirected graph is given by min edge weight along tree path. Uses the fact that for any i, j, k, $\lambda_{ik} \geq \min(\lambda_{ij}, \lambda_{jk})$, where λ_{ij} denotes the flow between i and j.
Time: $N - 1$ calls to Dinic

"Dinic.h" 0d712e, 16 lines

```
template<class F> V<pair<pi,F>> gomoryHu(int N,
    const V<pair<pi,F>>& ed) {
  vi par(N); Dinic<F> D; D.init(N);
  vpi ed_locs; each(t,ed)ed_locs.pb(D.ae(t.f.f,t.f.s,t.s,t.s));
  V<pair<pi,F>> ans;
  FOR(i,1,N) {
    each(p,ed_locs) { // reset capacities
      auto& e = D.adj.at(p.f).at(p.s);
      auto& e_rev = D.adj.at(e.to).at(e.rev);
      e.cap = e_rev.cap = (e.cap+e_rev.cap)/2;
    }
    ans.pb({{i,par[i]},D.maxFlow(i,par[i])});
    FOR(j,i+1,N) if (par[j] == par[i] && D.lev[j]) par[j] = i;
  }
  return ans;
}
```

MCMF.h

Description: Minimum-cost maximum flow, assumes no negative cycles. It is possible to choose negative edge costs such that the first run of Dijkstra is slow, but this hasn't been an issue in the past. Edge weights ≥ 0 for every subsequent run. To get flow through original edges, assign ID's during ae.
Time: Ignoring first run of Dijkstra, $\mathcal{O}(FM \log M)$ if caps are integers and F is max flow.

77bfb0, 46 lines

```
struct MCMF {
  using F = ll; using C = ll; // flow type, cost type
  struct Edge { int to, rev; F flo, cap; C cost; };
  int N; V<C> p, dist; vpi pre; V<V<Edge>> adj;
  void init(int _N) { N = _N;
    p.rsz(N), adj.rsz(N), dist.rsz(N), pre.rsz(N); }
  void ae(int u, int v, F cap, C cost) { assert(cap >= 0);
    adj[u].pb({v,sz(adj[v]),0,cap,cost});
    adj[v].pb({u,sz(adj[u])-1,0,0,-cost});
  } // use asserts, don't try smth dumb
  bool path(int s, int t) { // send flow through lowest cost
    ↪path
    const C inf = numeric_limits<C>::max(); dist.assign(N,inf);
    using T = pair<C,int>;
    priority_queue<T,V<T>,greater<T>> todo;
    todo.push({dist[s] = 0,s});
    while (sz(todo)) { // Dijkstra
      T x = todo.top(); todo.pop();
      if (x.f > dist[x.s]) continue;
      each(e,adj[x.s]) { // all weights should be non-negative
        if (e.flo < e.cap && ckmin(dist[e.to],
            x.f+e.cost+p[x.s]-p[e.to]))
          pre[e.to]={x.s,e.rev}, todo.push({dist[e.to],e.to});
      }
    } // if costs are doubles, add some EPS so you
    // don't traverse ~0-weight cycle repeatedly
```

```
    return dist[t] != inf; // true if augmenting path
  }
  pair<F,C> calc(int s, int t) { assert(s != t);
    F0R(_,N) F0R(i,N) each(e,adj[i]) // Bellman-Ford
      if (e.cap) ckmin(p[e.to],p[i]+e.cost);
    F totFlow = 0; C totCost = 0;
    while (path(s,t)) { // p -> potentials for Dijkstra
      F0R(i,N) p[i] += dist[i]; // don't matter for unreachable
      F df = numeric_limits<F>::max();
      for (int x = t; x != s; x = pre[x].f) {
        Edge& e = adj[pre[x].f][adj[x][pre[x].s].rev];
        ckmin(df,e.cap-e.flo); }
      totFlow += df; totCost += (p[t]-p[s])*df;
      for (int x = t; x != s; x = pre[x].f) {
        Edge& e = adj[x][pre[x].s]; e.flo -= df;
        adj[pre[x].f][e.rev].flo += df;
      }
    } // get max flow you can send along path
    return {totFlow,totCost};
  }
};
```

7.6 Matching

Hungarian.h

Description: Given array of (possibly negative) costs to complete each of N (1-indexed) jobs w/ each of M workers ($N \leq M$), finds min cost to complete all jobs such that each worker is assigned to at most one job. Dijkstra with potentials works in almost the same way as MCMF.
Time: $\mathcal{O}(N^2M)$

09d0ec, 28 lines

```
using C = ll;
C hungarian(const V<V<C>>& a) {
  int N = sz(a)-1, M = sz(a[0])-1; assert(N <= M);
  V<C> u(N+1), v(M+1); // potentials to make edge weights >= 0
  vi job(M+1);
  FOR(i,1,N+1) { // find alternating path with job i
    const C inf = numeric_limits<C>::max();
    int w = 0; job[w] = i; // add "dummy" worker 0
    V<C> dist(M+1,inf); vi pre(M+1,-1); vb done(M+1);
    while (job[w]) { // dijkstra
      done[w] = 1; int j = job[w], nexW; C delta = inf;
      // fix dist[j], update dists from j
      F0R(W,M+1) if (!done[W]) { // try all workers
        if (ckmin(dist[W],a[j][W]-u[j]-v[W])) pre[W] = w;
        if (ckmin(delta,dist[W])) nexW = W;
      }
      F0R(W,M+1) { // subtract constant from all edges going
        // from done -> not done vertices, lowers all
        // remaining dists by constant
        if (done[W]) u[job[W]] += delta, v[W] -= delta;
        else dist[W] -= delta;
      }
      w = nexW;
    } // potentials adjusted so all edge weights >= 0
    for (int W; w; w = W) job[w] = job[W = pre[w]];
  } // job[w] = 0, found alternating path
  return -v[0]; // min cost
}
```

UnweightedMatch.h

Description: Edmond's Blossom Algorithm. General unweighted matching with 1-based indexing. If `vis[v]=0` when `bfs` returns 0, `v` is not part of every max matching.
Time: $\mathcal{O}(N^3)$, faster in practice

513ca2, 54 lines

```
template<int SZ> struct UnweightedMatch {
  int match[SZ], N; vi adj[SZ];
  void ae(int u, int v) { adj[u].pb(v), adj[v].pb(u); }
  queue<int> q;
  int par[SZ], vis[SZ], orig[SZ], aux[SZ];
  void augment(int u, int v) { // toggle edges on u-v path
    while (1) { // one more matched pair
      int pv = par[v], nv = match[pv];
      match[v] = pv; match[pv] = v;
      v = nv; if (u == pv) return;
    }
  }
  int lca(int u, int v) { // find LCA of supernodes in O(dist)
    static int t = 0;
    for (++t;;swap(u,v)) {
      if (!u) continue;
      if (aux[u] == t) return u; // found LCA
      aux[u] = t; u = orig[par[match[u]]];
    }
  }
  void blossom(int u, int v, int a) { // go other way
    for (; orig[u] != a; u = par[v]) { // around cycle
      par[u] = v; v = match[u]; // treat u as if vis[u] = 1
      if (vis[v] == 1) vis[v] = 0, q.push(v);
      orig[u] = orig[v] = a; // merge into supernode
    }
  }
  bool bfs(int u) { // u is initially unmatched
    F0R(i,N+1) par[i] = 0, vis[i] = -1, orig[i] = i;
    q = queue<int>(); vis[u] = 0, q.push(u);
    while (sz(q)) { // each node is pushed to q at most once
      int v = q.ft; q.pop(); // 0 -> unmatched vertex
      each(x,adj[v]) {
        if (vis[x] == -1) { // neither of x, match[x] visited
          vis[x] = 1; par[x] = v;
          if (!match[x]) return augment(u,x),1;
          vis[match[x]] = 0, q.push(match[x]);
        } else if (vis[x] == 0 && orig[v] != orig[x]) {
          int a = lca(orig[v],orig[x]); // odd cycle
          blossom(x,v,a), blossom(v,x,a);
        } // contract O(n) times
      }
    }
    return 0;
  }
  int calc(int _N) { // rand matching -> constant improvement
    N = _N; F0R(i,N+1) match[i] = aux[i] = 0;
    int ans = 0; vi V(N); iota(all(V),1); shuffle(all(V),rng);
      ↪// find rand matching
    each(x,V) if (!match[x]) each(y,adj[x]) if (!match[y]) {
      match[x] = y, match[y] = x; ++ans; break; }
    FOR(i,1,N+1) if (!match[i] && bfs(i)) ++ans;
    return ans;
  }
};
```

WeightedMatch.h

Description: General max weight max matching with 1-based indexing. Edge weights must be positive, combo of UnweightedMatch and Hungarian.
Time: $\mathcal{O}(N^3)$?

120873, 145 lines

```
template<int SZ> struct WeightedMatch {
  struct edge { int u,v,w; }; edge g[SZ*2][SZ*2];
  void ae(int u, int v, int w) { g[u][v].w = g[v][u].w = w; }
  int N,NX,lab[SZ*2],match[SZ*2],slack[SZ*2],st[SZ*2];
  int par[SZ*2],floFrom[SZ*2][SZ],S[SZ*2],aux[SZ*2];
  vi flo[SZ*2]; queue<int> q;
  void init(int _N) { N = _N; // init all edges
    FOR(u,1,N+1) FOR(v,1,N+1) g[u][v] = {u,v,0}; }
  int eDelta(edge e) { // >= 0 at all times
    return lab[e.u]+lab[e.v]-g[e.u][e.v].w*2; }
  void updSlack(int u, int x) { // smallest edge -> blossom x
    if (!slack[x] || eDelta(g[u][x]) < eDelta(g[slack[x]][x]))
      slack[x] = u; }
  void setSlack(int x) {
    slack[x] = 0; FOR(u,1,N+1) if (g[u][x].w > 0
     && st[u] != x && S[st[u]] == 0) updSlack(u,x); }
  void qPush(int x) {
    if (x <= N) q.push(x);
    else each(t,flo[x]) qPush(t); }
  void setSt(int x, int b) {
    st[x] = b; if (x > N) each(t,flo[x]) setSt(t,b); }
  int getPr(int b, int xr) { // get even position of xr
    int pr = find(all(flo[b]),xr)-begin(flo[b]);
    if (pr&1) { reverse(1+all(flo[b])); return sz(flo[b])-pr; }
    return pr; }
  void setMatch(int u, int v) { // rearrange flo[u], matches
    edge e = g[u][v]; match[u] = e.v; if (u <= N) return;
    int xr = floFrom[u][e.u], pr = getPr(u,xr);
    F0R(i,pr) setMatch(flo[u][i],flo[u][i^1]);
    setMatch(xr,v); rotate(begin(flo[u]),pr+all(flo[u])); }
  void augment(int u, int v) { // set matches including u->v
    while (1) { // and previous ones
      int xnv = st[match[u]]; setMatch(u,v);
      if (!xnv) return;
      setMatch(xnv,st[par[xnv]]);
      u = st[par[xnv]], v = xnv;
    }
  }
  int lca(int u, int v) { // same as in unweighted
    static int t = 0; // except maybe return 0
    for (++t;u||v;swap(u,v)) {
      if (!u) continue;
      if (aux[u] == t) return u;
      aux[u] = t; u = st[match[u]];
      if (u) u = st[par[u]];
    }
    return 0;
  }
  void addBlossom(int u, int anc, int v) {
    int b = N+1; while (b <= NX && st[b]) ++b;
    if (b > NX) ++NX; // new blossom
    lab[b] = S[b] = 0; match[b] = match[anc]; flo[b] = {anc};
    auto blossom = [&](int x) {
      for (int y; x != anc; x = st[par[y]])
        flo[b].pb(x), flo[b].pb(y = st[match[x]]), qPush(y);
    };
    blossom(u); reverse(1+all(flo[b])); blossom(v); setSt(b,b);
    // identify all nodes in current blossom
    FOR(x,1,NX+1) g[b][x].w = g[x][b].w = 0;
    FOR(x,1,N+1) floFrom[b][x] = 0;
    each(xs,flo[b]) { // find tightest constraints
      FOR(x,1,NX+1) if (g[b][x].w == 0 || eDelta(g[xs][x]) <
        eDelta(g[b][x])) g[b][x]=g[xs][x], g[x][b]=g[x][xs];
      FOR(x,1,N+1) if (floFrom[xs][x]) floFrom[b][x] = xs;
    } // floFrom to deconstruct blossom
    setSlack(b); // since didn't qPush everything
  }
  void expandBlossom(int b) {
    each(t,flo[b]) setSt(t,t); // undo setSt(b,b)
    int xr = floFrom[b][g[b][par[b]].u], pr = getPr(b,xr);
    for(int i = 0; i < pr; i += 2) {
      int xs = flo[b][i], xns = flo[b][i+1];
      par[xs] = g[xns][xs].u; S[xs] = 1; // no setSlack(xns)?
      S[xns] = slack[xs] = slack[xns] = 0; qPush(xns);
    }
    S[xr] = 1, par[xr] = par[b];
    FOR(i,pr+1,sz(flo[b])) { // matches don't change
      int xs = flo[b][i]; S[xs] = -1, setSlack(xs); }
    st[b] = 0; // blossom killed
```

```
  }
  bool onFoundEdge(edge e) {
    int u = st[e.u], v = st[e.v];
    if (S[v] == -1) { // v unvisited, matched with smth else
      par[v] = e.u, S[v] = 1; slack[v] = 0;
      int nu = st[match[v]]; S[nu] = slack[nu] = 0; qPush(nu);
    } else if (S[v] == 0) {
      int anc = lca(u,v); // if 0 then match found!
      if (!anc) return augment(u,v),augment(v,u),1;
      addBlossom(u,anc,v);
    }
    return 0;
  }
  bool matching() {
    q = queue<int>();
    FOR(x,1,NX+1) {
      S[x] = -1, slack[x] = 0; // all initially unvisited
      if (st[x] == x && !match[x]) par[x] = S[x] = 0, qPush(x);
    }
    if (!sz(q)) return 0;
    while (1) {
      while (sz(q)) { // unweighted matching with tight edges
        int u = q.ft; q.pop(); if (S[st[u]] == 1) continue;
        FOR(v,1,N+1) if (g[u][v].w > 0 && st[u] != st[v]) {
          if (eDelta(g[u][v]) == 0) { // condition is strict
            if (onFoundEdge(g[u][v])) return 1;
          } else updSlack(u,st[v]);
        }
      }
      int d = INT_MAX;
      FOR(b,N+1,NX+1) if (st[b] == b && S[b] == 1)
        ckmin(d,lab[b]/2); // decrease lab[b]
      FOR(x,1,NX+1) if (st[x] == x && slack[x]) {
        if (S[x] == -1) ckmin(d,eDelta(g[slack[x]][x]));
        else if (S[x] == 0) ckmin(d,eDelta(g[slack[x]][x])/2);
      } // edge weights shouldn't go below 0
      FOR(u,1,N+1) {
        if (S[st[u]] == 0) {
          if (lab[u] <= d) return 0; // why?
          lab[u] -= d;
        } else if (S[st[u]] == 1) lab[u] += d;
      } // lab has opposite meaning for verts and blossoms
      FOR(b,N+1,NX+1) if (st[b] == b && S[b] != -1)
        lab[b] += (S[b] == 0 ? 1 : -1)*d*2;
      q = queue<int>();
      FOR(x,1,NX+1) if (st[x]==x && slack[x] // new tight edge
        && st[slack[x]] != x && eDelta(g[slack[x]][x]) == 0)
          if (onFoundEdge(g[slack[x]][x])) return 1;
      FOR(b,N+1,NX+1) if (st[b]==b && S[b]==1 && lab[b]==0)
        expandBlossom(b); // odd dist blossom taken apart
    }
    return 0;
  }
  pair<ll,int> calc() {
    NX = N; st[0] = 0; FOR(i,1,2*N+1) aux[i] = 0;
    FOR(i,1,N+1) match[i] = 0, st[i] = i, flo[i].clear();
    int wMax = 0;
    FOR(u,1,N+1) FOR(v,1,N+1)
      floFrom[u][v] = (u == v ? u : 0), ckmax(wMax,g[u][v].w);
    FOR(u,1,N+1) lab[u] = wMax; // start high and decrease
    int num = 0; ll wei = 0; while (matching()) ++num;
    FOR(u,1,N+1) if (match[u] && match[u] < u)
      wei += g[u][match[u]].w; // edges in matching
    return {wei,num};
  }
};
```

MaxMatchLexMin.h

Description: lexiographically least matching wrt left vertices
Usage: solve(L,R,sz(L))
Time: $\log |L|$ times sum of complexities of gen, maxMatch

44bb49, 26 lines

```
vpi maxMatch(vi L, vi R); // return pairs in max matching
pair<vi,vi> gen(vi L, vi R); // return {Lp,Rp}, vertices on
// left/right that can be reached by alternating path from
// unmatched node on left after finding max matching

vpi res; // stores answer
void solve(vi L, vi R, int x) { // first |L|-x elements of L
  if (x <= 1) { // are in matching, easy if x <= 1
    vpi v = maxMatch(L,R);
    if (sz(v) != sz(L)) L.pop_back(), v = maxMatch(L,R);
    assert(sz(v) == sz(L));
    res.insert(end(res),all(v)); return;
  }
  vi Lp,Rp; tie(Lp,Rp)=gen(L,R); vi Lm=sub(L,Lp),Rm=sub(R,Rp);
  // Lp U Rm is max indep set, Lm U Rp is min vertex cover
  // Lp and Rm independent, edges from Lm to Rp can be ignored
  vpi v = maxMatch(Lm,Rm); assert(sz(v) == sz(Lm));
  res.insert(end(res),all(v));
  vi L2(all(L)-x/2); vi Lp2,Rp2; tie(Lp2,Rp2) = gen(L2,R);
  int cnt = 0; each(t,Lp2) cnt += t >= L[sz(L)-x];
  solve(Lp2,Rp2,cnt); // Rp2 covered by best matching
  vi LL = sub(Lp,Lp2), RR = sub(Rp,Rp2); // those in Lp but not
  // Lp2 that are < L[sz(L)-x/2] must be in answer, not cnt
  cnt = 0; each(t,LL) cnt += t >= L[sz(L)-x/2];
  solve(LL,RR,cnt); // do rest
} // x reduced by factor of at least two
```

MaxMatchFast.h

Description: Fast bipartite matching.
Time: $\mathcal{O}\left(M\sqrt{N}\right)$

ec6c96, 31 lines

```
vpi maxMatch(int L, int R, const vpi& edges) {
  V<vi> adj = V<vi>(L);
  vi nxt(L,-1), prv(R,-1), lev, ptr;
  F0R(i,sz(edges)) adj.at(edges[i].f).pb(edges[i].s);
  while (true) {
    lev = ptr = vi(L); int max_lev = 0;
    queue<int> q; F0R(i,L) if (nxt[i]==-1) lev[i]=1, q.push(i);
    while (sz(q)) {
      int x = q.ft; q.pop();
      for (int y: adj[x]) {
        int z = prv[y];
        if (z == -1) max_lev = lev[x];
        else if (!lev[z]) lev[z] = lev[x]+1, q.push(z);
      }
      if (max_lev) break;
    }
    if (!max_lev) break;
    F0R(i,L) if (lev[i] > max_lev) lev[i] = 0;
    auto dfs = [&](auto self, int x) -> bool {
      for (;ptr[x] < sz(adj[x]);++ptr[x]) {
        int y = adj[x][ptr[x]], z = prv[y];
        if (z == -1 || (lev[z] == lev[x]+1 && self(self,z)))
          return nxt[x]=y, prv[y]=x, ptr[x]=sz(adj[x]), 1;
      }
      return 0;
    };
    F0R(i,L) if (nxt[i] == -1) dfs(dfs,i);
  }
  vpi ans; F0R(i,L) if (nxt[i] != -1) ans.pb({i,nxt[i]});
  return ans;
}
```

7.7 Advanced

ChordalGraphRecognition.h

Description: Recognizes graph where every induced cycle has length exactly 3 using maximum adjacency search.

6cc97d, 58 lines

```
int N,M;
set<int> adj[MX];
int cnt[MX];
vi ord, rord;

vi find_path(int x, int y, int z) {
  vi pre(N,-1);
  queue<int> q; q.push(x);
  while (sz(q)) {
    int t = q.ft; q.pop();
    if (adj[t].count(y)) {
      pre[y] = t; vi path = {y};
      while (path.bk != x) path.pb(pre[path.bk]);
      path.pb(z);
      return path;
    }
    each(u,adj[t]) if (u != z && !adj[u].count(z) && pre[u] ==
      ↪-1) {
      pre[u] = t;
      q.push(u);
    }
  }
  assert(0);
}

int main() {
  setIO(); re(N,M);
  F0R(i,M) {
    int a,b; re(a,b);
    adj[a].insert(b), adj[b].insert(a);
  }
  rord = vi(N,-1);
  priority_queue<pi> pq;
  F0R(i,N) pq.push({0,i});
  while (sz(pq)) {
    pi p = pq.top(); pq.pop();
    if (rord[p.s] != -1) continue;
    rord[p.s] = sz(ord); ord.pb(p.s);
    each(t,adj[p.s]) pq.push({++cnt[t],t});
  }
  assert(sz(ord) == N);
  each(z,ord) {
    pi big = {-1,-1};
    each(y,adj[z]) if (rord[y] < rord[z])
      ckmax(big,mp(rord[y],y));
    if (big.f == -1) continue;
    int y = big.s;
    each(x,adj[z]) if (rord[x] < rord[y]) if (!adj[y].count(x))
      ↪ {
      ps("NO");
      vi v = find_path(x,y,z);
      ps(sz(v));
      each(t,v) pr(t,' ');
      exit(0);
    }
  }
  ps("YES");
  reverse(all(ord));
  each(z,ord) pr(z,' ');
}
```

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]);
    }
  }
};
```

EdgeColor.h

Description: Used only once. Naive implementation of Misra & Gries edge coloring. By Vizing's Theorem, a simple graph with max degree d can be edge colored with at most $d+1$ colors
Time: $\mathcal{O}(N^2 M)$, faster in practice

cc2b29, 40 lines

```
template<int SZ> struct EdgeColor {
  int N = 0, maxDeg = 0, adj[SZ][SZ], deg[SZ];
  void init(int _N) { N = _N;
    F0R(i,N) { deg[i] = 0; F0R(j,N) adj[i][j] = 0; } }
  void ae(int a, int b, int c) {
    adj[a][b] = adj[b][a] = c; }
  int delEdge(int a, int b) {
    int c = adj[a][b]; adj[a][b] = adj[b][a] = 0;
    return c; }
  V<bool> genCol(int x) {
    V<bool> col(N+1); F0R(i,N) col[adj[x][i]] = 1;
    return col; }
  int freeCol(int u) {
    auto col = genCol(u); int x = 1;
    while (col[x]) ++x; return x; }
  void invert(int x, int d, int c) {
    F0R(i,N) if (adj[x][i] == d)
      delEdge(x,i), invert(i,c,d), ae(x,i,c); }
  void ae(int u, int v) {
    // check if you can add edge w/o doing any work
    assert(N); ckmax(maxDeg,max(++deg[u],++deg[v]));
    auto a = genCol(u), b = genCol(v);
    FOR(i,1,maxDeg+2) if (!a[i] && !b[i])
      return ae(u,v,i);
    V<bool> use(N); vi fan = {v}; use[v] = 1;
    while (1) {
      auto col = genCol(fan.bk);
      if (sz(fan) > 1) col[adj[fan.bk][u]] = 0;
      int i=0; while (i<N && (use[i] || col[adj[u][i]])) i++;
      if (i < N) fan.pb(i), use[i] = 1;
      else break;
    }
    int c = freeCol(u), d = freeCol(fan.bk); invert(u,d,c);
    int i = 0; while (i < sz(fan) && genCol(fan[i])[d]
      && adj[u][fan[i]] != d) i ++;
    assert (i != sz(fan));
    F0R(j,i) ae(u,fan[j],delEdge(u,fan[j+1]));
    ae(u,fan[i],d);
  }
};
```

DirectedMST.h

Description: Chu-Liu-Edmonds algorithm. Computes minimum weight directed spanning tree rooted at r, edge from $par[i] \to i$ for all $i \neq r$. Use DSU with rollback if need to return edges.
Time: $\mathcal{O}(M \log M)$

"DSUrb.h" 5d5c10, 61 lines

```
struct Edge { int a, b; ll w; };
struct Node { // lazy skew heap 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, const vector<Edge>& g) {
  DSUrb dsu; dsu.init(n);
  vector<Node*> heap(n); // store edges entering each vertex
  // in increasing order of weight
  each(e,g) heap[e.b] = merge(heap[e.b], new Node{e});
  ll res = 0; vi seen(n,-1); seen[r] = r;
  vpi in(n,{-1,-1}); // edge entering each vertex in MST
  vector<pair<int,vector<Edge>>> cycs;
  F0R(s,n) {
    int u = s, w;
    vector<pair<int,Edge>> path;
    while (seen[u] < 0) {
      if (!heap[u]) return {-1,{}};
      seen[u] = s;
      Edge e = heap[u]->top(); path.pb({u,e});
      heap[u]->delta -= e.w, pop(heap[u]);
      res += e.w, u = dsu.get(e.a);
      if (seen[u] == s) { // found cycle, contract
        Node* cyc = 0; cycs.eb();
        do {
          cyc = merge(cyc, heap[w = path.bk.f]);
          cycs.bk.s.pb(path.bk.s);
          path.pop_back();
        } while (dsu.unite(u,w));
        u = dsu.get(u); heap[u] = cyc, seen[u] = -1;
        cycs.bk.f = u;
      }
    }
    each(t,path) in[dsu.get(t.s.b)] = {t.s.a,t.s.b};
  } // found path from root to s, done
  while (sz(cycs)) { // expand cycs to restore sol
    auto c = cycs.bk; cycs.pop_back();
    pi inEdge = in[c.f];
    each(t,c.s) dsu.rollback();
    each(t,c.s) in[dsu.get(t.b)] = {t.a,t.b};
    in[dsu.get(inEdge.s)] = inEdge;
  }
  vi par(n); F0R(i,n) par[i] = in[i].f;
  // i == r ? in[i].s == -1 : in[i].s == i
  return {res,par};
}
```

LCT.h

Description: Link-Cut Tree. Given a function $f(1 \dots N) \to 1 \dots N$, evaluates $f^b(a)$ for any a, b. `sz` is for path queries; `sub`, `vsub` are for subtree queries. `x->access()` brings `x` to the top and propagates it; its left subtree will be the path from `x` to the root and its right subtree will be empty. Then `sub` will be the number of nodes in the connected component of `x` and `vsub` will be the number of nodes under `x`. Use `makeRoot` for arbitrary path queries.
Usage: `FOR(i,1,N+1)LCT[i]=new snode(i); link(LCT[1],LCT[2],1);`
Time: $\mathcal{O}(\log N)$

e24bf7, 115 lines

```
typedef struct snode* sn;
struct snode { //////// VARIABLES
  sn p, c[2]; // parent, children
  sn extra; // extra cycle node for "The Applicant"
  bool flip = 0; // subtree flipped or not
  int val, sz; // value in node, # nodes in current splay tree
  int sub, vsub = 0; // vsub stores sum of virtual children
  snode(int _val) : val(_val) {
    p = c[0] = c[1] = extra = NULL; calc(); }
  friend int getSz(sn x) { return x?x->sz:0; }
  friend int getSub(sn x) { return x?x->sub:0; }
  void prop() { // lazy prop
    if (!flip) return;
    swap(c[0],c[1]); flip = 0;
    F0R(i,2) if (c[i]) c[i]->flip ^= 1;
  }
  void calc() { // recalc vals
    F0R(i,2) if (c[i]) c[i]->prop();
    sz = 1+getSz(c[0])+getSz(c[1]);
    sub = 1+getSub(c[0])+getSub(c[1])+vsub;
  }
  //////// SPLAY TREE OPERATIONS
  int dir() {
    if (!p) return -2;
    F0R(i,2) if (p->c[i] == this) return i;
    return -1; // p is path-parent pointer
  } // -> not in current splay tree
  // test if root of current splay tree
  bool isRoot() { return dir() < 0; }
  friend void setLink(sn x, sn y, int d) {
    if (y) y->p = x;
    if (d >= 0) x->c[d] = y; }
  void rot() { // assume p and p->p propagated
```

```
    assert(!isRoot()); int x = dir(); sn pa = p;
    setLink(pa->p, this, pa->dir());
    setLink(pa, c[x^1], x); setLink(this, pa, x^1);
    pa->calc();
  }
  void splay() {
    while (!isRoot() && !p->isRoot()) {
      p->p->prop(), p->prop(), prop();
      dir() == p->dir() ? p->rot() : rot();
      rot();
    }
    if (!isRoot()) p->prop(), prop(), rot();
    prop(); calc();
  }
  sn fbo(int b) { // find by order
    prop(); int z = getSz(c[0]); // of splay tree
    if (b == z) { splay(); return this; }
    return b < z ? c[0]->fbo(b) : c[1] -> fbo(b-z-1);
  }
  //////// BASE OPERATIONS
  void access() { // bring this to top of tree, propagate
    for (sn v = this, pre = NULL; v; v = v->p) {
      v->splay(); // now switch virtual children
      if (pre) v->vsub -= pre->sub;
      if (v->c[1]) v->vsub += v->c[1]->sub;
      v->c[1] = pre; v->calc(); pre = v;
    }
    splay(); assert(!c[1]); // right subtree is empty
  }
  void makeRoot() {
    access(); flip ^= 1; access(); assert(!c[0] && !c[1]); }
  //////// QUERIES
  friend sn lca(sn x, sn y) {
    if (x == y) return x;
    x->access(), y->access(); if (!x->p) return NULL;
    x->splay(); return x->p?:x; // y was below x in latter case
  } // access at y did not affect x -> not connected
  friend bool connected(sn x, sn y) { return lca(x,y); }
  // # nodes above
  int distRoot() { access(); return getSz(c[0]); }
  sn getRoot() { // get root of LCT component
    access(); sn a = this;
    while (a->c[0]) a = a->c[0], a->prop();
    a->access(); return a;
  }
  sn getPar(int b) { // get b-th parent on path to root
    access(); b = getSz(c[0])-b; assert(b >= 0);
    return fbo(b);
  } // can also get min, max on path to root, etc
  //////// MODIFICATIONS
  void set(int v) { access(); val = v; calc(); }
  friend void link(sn x, sn y, bool force = 0) {
    assert(!connected(x,y));
    if (force) y->makeRoot(); // make x par of y
    else { y->access(); assert(!y->c[0]); }
    x->access(); setLink(y,x,0); y->calc();
  }
  friend void cut(sn y) { // cut y from its parent
    y->access(); assert(y->c[0]);
    y->c[0]->p = NULL; y->c[0] = NULL; y->calc(); }
  friend void cut(sn x, sn y) { // if x, y adj in tree
    x->makeRoot(); y->access();
    assert(y->c[0] == x && !x->c[0] && !x->c[1]); cut(y); }
};
sn LCT[MX];

//////// THE APPLICANT SOLUTION
void setNex(sn a, sn b) { // set f[a] = b
  if (connected(a,b)) a->extra = b;
  else link(b,a); }
void delNex(sn a) { // set f[a] = NULL
  auto t = a->getRoot();
  if (t == a) { t->extra = NULL; return; }
  cut(a); assert(t->extra);
  if (!connected(t,t->extra))
    link(t->extra,t), t->extra = NULL;
}
sn getPar(sn a, int b) { // get f^b[a]
  int d = a->distRoot(); if (b <= d) return a->getPar(b);
  b -= d+1; auto r = a->getRoot()->extra; assert(r);
  d = r->distRoot()+1; return r->getPar(b%d);
}
```

TopTree.h

Description: Top tree (generalization of LCT to support path & subtree updates & queries)

b31892, 417 lines

```
/*
 * Usage:
 *   Implement
 *     void update()
 *     void downdate()
 *     void do_flip_path()
 *     void do_other_operation() ...
 *   When update() is called, you can assume downdate() has
 ↪already been called.
 *
 *   In general, do_op() should eagerly apply the operation but
 ↪ not touch the
 *   children. In downdate(), you can push down to the children
 ↪ with ch->do_op().
 *   WARNING : if different operations do not trivially commute
 ↪, you *must*
 *   implement a way to swap/alter them to compose in a
 ↪consistent order, and you
 *   must use that order when implementing downdate(). This can
 ↪ be nontrivial!
 *
 *   Creating vertices:
 *     n->is_path = n->is_vert = true;
 *     n->update();
 *
 *   Creating edges: no setup/update() needed, just call
 *     link(e, va, vb);
 *
 *   Updates:
 *     auto cur = get_path(va, vb); // or get_subtree(va, vb)
 *     cur->do_stuff();
 *     cur->downdate();
 *     cur->update_all();
 *
 * Node types:
 *   path edges: compress(c[0], self, c[1])
 *     assert(is_path && !is_vert);
 *     assert(c[0] && c[1]);
 *     assert(c[0]->is_path && c[1]->is_path);
 *     assert(!c[2]);
 *   (path) vertices: self + rake(c[0], c[1])
 *     assert(is_path && is_vert);
 *     assert(!c[2]);
 *     if (c[0]) assert(!c[0]->is_path);
 *     if (c[1]) assert(!c[1]->is_path);
 *   non-path edges: rake(c[0], self + c[2], c[1])
 *     assert(!is_path && !is_vert);
 *     assert(c[2]);
 *     assert(c[2]->is_path);
 *     if (c[0]) assert(!c[0]->is_path);
 *     if (c[1]) assert(!c[1]->is_path);
 */

using pt = struct top_tree_node*;
struct top_tree_node {
private:
  mutable pt p = nullptr;
  AR<pt, 3> c{nullptr, nullptr, nullptr};
  // \return direction in which parent points to you
  int d() const {
    if (!p) return -1;
    F0R(i,3) if (this == p->c[i]) return i;
    assert(false);
  }
  // \return true if this is root of rake or compress tree
  bool r() const { return !p || p->is_path != is_path; }

public:
  // 3 types of verts: path edges, path verts, non-path edges
  bool is_path, is_vert;
  bool flip_path = false;

  // MODIFY STUFF BELOW
  int path_len,  best_path;
  bool own_parity, path_parity;
  AR<int,2> best_down, best_up;
  void do_flip_path() {
    assert(is_path); flip_path ^= 1;
    swap(best_down, best_up);
  }
  void downdate() {
    if (flip_path) {
      assert(is_path);
      if (!is_vert) F0R(i,2) if (c[i]) c[i]->do_flip_path(); //
        ↪ if vert, then you're at a leaf so don't propagate
      swap(c[0], c[1]);
      flip_path = false;
    }
  }
  void update() { // TODO: find longest path of each parity
    assert(!flip_path);
    if (is_path && !is_vert) {
      assert(c[0] && c[1] && !c[2]);
      path_len = 1+c[0]->path_len+c[1]->path_len;
      path_parity = own_parity^c[0]->path_parity^c[1]->
        ↪path_parity;

      best_up = c[0]->best_up;
      F0R(z,2)
        ckmax(best_up[c[0]->path_parity^own_parity^z],c[0]->
          ↪path_len+1+c[1]->best_up[z]);

      best_down = c[1]->best_down;
      F0R(z,2)
        ckmax(best_down[c[1]->path_parity^own_parity^z],c[1]->
          ↪path_len+1+c[0]->best_down[z]);

      best_path = max(c[0]->best_path,c[1]->best_path);
      F0R(z,2) ckmax(best_path,c[0]->best_down[z]+1+c[1]->
        ↪best_up[z^own_parity]);
    } else {
      path_len = 0;
      path_parity = 0;
      best_up = {0,-MOD};
      best_path = 0;
      if (!is_vert) {
        best_path = c[2]->best_path;
        AR<int,2> tmp_up = c[2]->best_up;
        F0R(i,2) ++tmp_up[i];
        if (own_parity) swap(tmp_up[0],tmp_up[1]);
```

```
        F0R(i,2) {
          ckmax(best_up[i],tmp_up[i]);
          if (i == 0) ckmax(best_path,best_up[i]);
        }
      }
      F0R(d,2) {
        if (!c[d]) continue;
        ckmax(best_path,c[d]->best_path);
        F0R(z,2) {
          ckmax(best_path,c[d]->best_up[z]+best_up[z]);
          ckmax(best_up[z],c[d]->best_up[z]);
        }
      }
      // rake(c[0],c[1],self+c[2])
      best_down = best_up;
    }
  }
  void downdate_all() {
    if (p) p->downdate_all();
    downdate();
  }
  // Returns the root
  pt update_all() {
    pt cur = this; cur->update();
    while (cur->p) { cur = cur->p; cur->update(); }
    return cur;
  }
private:
  // sets y to be the d'th child of x
  friend void setLink(pt x, pt y, int d) {
    if (y) y->p = x;
    if (d != -1) x->c[d] = y;
  }
  void rot() { // rotate this up, parent down
    assert(!is_vert && !r());
    pt pa = p;
    int x = d(); assert(x == 0 || x == 1);
    pt ch = c[!x];
    setLink(pa->p,this,pa->d());
    setLink(pa,ch,x);
    setLink(this,pa,!x);
    pa->update();
  }
  void rot_2(int c_d) { // rotate this up (along with c_d'th child), parent down
    assert(!is_vert && !r());
    assert(c[c_d] && !c[c_d]->is_vert);
    if (d() == c_d) { rot(); return; }
    pt pa = p;
    int x = d(); assert(x == 0 || x == 1);
    assert(c_d == !x);
    pt ch = c[c_d]->c[!x];
    setLink(pa->p,this,pa->d());
    setLink(pa,ch,x);
    setLink(this->c[c_d],pa,!x);
    pa->update();
  }
  void splay_dir(int x) { // splay while direction is x
    while (!r() && d() == x) {
      if (!p->r() && p->d() == x) p->rot();
      rot();
    }
  }
  // splay path edge along with child path edge
  void splay_2(int c_d) {
    assert(!is_vert && is_path);
    assert(c[c_d] && !c[c_d]->is_vert);
    while (!r()) {
      if (!p->r()) {
        if (p->d() == d()) p->rot();
        else rot_2(c_d);
      }
      rot_2(c_d);
    }
  }
  // splay parent edge to top of tree, bringing this along for the ride
  void splay_2() {
    assert(!is_vert && is_path && !r());
    p->splay_2(d());
  }
  // splay vertex as close to top of tree as possible
  void splay_vert() {
    assert(is_vert && is_path); if (r()) return; // if path is single vertex, done
    p->splay_dir(d()); if (p->r()) return; // if parent is already at top, done
    // otherwise vertex is in between two edges
    assert(p->d() != d());
    if (d() == 1) p->rot();
    assert(d() == 0);
    p->splay_2(); // splay parent of parent to the top
    assert(d() == 0 && p->d() == 1 && p->p->r());
  }
  void splay() { // normal splay of an edge
    assert(!is_vert);
    while (!r()) {
      if (!p->r()) {
        if (p->d() == d()) p->rot();
        else rot();
      }
      rot();
    }
  }
  // either brings root to top of splay tree, or to right child of top of splay tree
  // cuts part of compress tree to the right of this
  pt cut_right() {
    assert(is_vert && is_path);
    splay_vert();
    if (r() || d() == 1) { // if last vertex on path, do nothing
      assert(r() || (d() == 1 && p->r()));
      assert(c[0] == nullptr); // why?
      return nullptr; // don't need to cut anything
    }
    // goal: cut pa and everything to the right of it
    // make it a rake

    // before:
    // pa->p
    //   \
    //    \
    //     pa
    //    /
    //   /
    // this (vertex, children are rake trees)

    // after:
    // pa->p
    //   \
    //    \
    //     this (pa as foster child)

    // or -> single vertex
    pt pa = p;
    assert(pa->r() || (pa->d() == 1 && pa->p->r()));
    assert(!pa->is_vert && pa->is_path);
    assert(pa->c[0] == this && pa->c[2] == nullptr); // pa is path edge
    setLink(pa->p,this,pa->d()); // now this or this->p is root
    pa->is_path = false; pa->c[2] = pa->c[1]; // pa is now a rake tree
    F0R(i,2) setLink(pa,c[i],i); // pa inherits rake children of this
    c[0] = nullptr; setLink(this,pa,1); // set pa to be a rake
    assert(c[2] == nullptr);
    pa->update(); return pa;
  }
  pt splice_non_path() { // cut some previous path, replace with new path
    assert(!is_path && !is_vert); splay(); // bring to top of rake tree
    assert(p && p->is_vert && p->is_path); // parent is a vertex
    p->cut_right(); // cut part to right of vertex
    if (!p->is_path) rot(); // rotate this to top of rake tree again
    assert(p && p->is_vert && p->is_path); // same parent
    assert(p->r() || (p->d() == 1 && p->p->r()));
    assert(p->c[d()] == this && p->c[!d()] == nullptr); // parent vertex only has one rake child? why?
    pt pa = p;
    setLink(pa->p,this,pa->d());
    F0R(i,2) setLink(pa,c[i],i);
    assert(c[2] && c[2]->is_path);
    c[1] = c[2]; // don't need to change parent
    setLink(this,pa,0);
    c[2] = nullptr;
    is_path = true;
    assert(d() != 0);
    pa->update(); return pa;
  }

  // Return the topmost vertex which was spliced into
  pt splice_all() { // make this part of topmost path
    pt res = nullptr;
    for (pt cur = this; cur; cur = cur->p) {
      if (!cur->is_path) res = cur->splice_non_path();
      assert(cur->is_path);
    }
    return res;
  }

public:
  pt getRoot() {
    expose();
    pt v = this; while (v->p) v = v->p;
    assert(v);
    v->downdate();
    while (!v->is_vert) {
      assert(v->c[0]);
      v = v->c[0]; v->downdate();
    }
    v->expose();
    return v;
  }
  friend bool connected(pt a, pt b) {
    return a->getRoot() == b->getRoot();
  }
  // Return the topmost vertex which was spliced into
  pt expose() { // yay makes sense!
    assert(is_vert);
    downdate_all(); // make sure to propagate everything above
    pt res = splice_all(); cut_right(); update_all();
    return res;
  }
```

```
  // Brings edge to the top
  // Return the topmost vertex which was spliced into
  pt expose_edge() {
    assert(!is_vert); downdate_all();
    pt v = is_path ? c[1] : c[2]; v->downdate();
    // if is_path: path to right of edge
    // otherwise: compress tree under edge
    while (!v->is_vert) { v = v->c[0]; v->downdate(); }
    pt res = v->splice_all(); v->cut_right(); v->update_all();
      ↪// same as expose
    assert(!p && v == c[1]);
    return res;
  }
  // make sure path end only has one child
  // Return the new root
  pt meld_path_end() {
    assert(!p);
    pt rt = this;
    while (true) {
      rt->downdate();
      if (rt->is_vert) break;
      rt = rt->c[1];
    }
    assert(rt->is_vert);
    rt->splay_vert();
    if (rt->c[0] && rt->c[1]) { // make sure path end has only
      ↪one rake child ...
      pt ch = rt->c[1];
      while (true) {
        ch->downdate();
        if (!ch->c[0]) break;
        ch = ch->c[0];
      }
      ch->splay();
      assert(ch->c[0] == nullptr);
      setLink(ch,rt->c[0],0); rt->c[0] = nullptr;
      ch->update();
    } else if (rt->c[0]) {
      rt->c[1] = rt->c[0];
      rt->c[0] = nullptr;
    }
    assert(rt->c[0] == nullptr);
    return rt->update_all();
  }
  void make_root() {
    expose();
    pt rt = this;
    while (rt->p) {
      assert(rt->d() == 1);
      rt = rt->p;
    }
    rt->do_flip_path(); rt->meld_path_end();
    expose(); assert(!p); // root path is now single node
  }
  // link v2 as a child of v1 with edge e
  friend void link(pt e, pt v1, pt v2) {
    assert(e && v1 && v2); F0R(i,3) assert(!e->c[i]);
    e->is_path = true, e->is_vert = false;
    v1->expose(); while (v1->p) v1 = v1->p;
    v2->make_root();
    assert(!v1->p && !v2->p); // should both be at top
    setLink(e,v1,0); setLink(e,v2,1);
    e->update();
  }
  // Cuts the edge e
  // Returns the top-tree-root of the two halves; they are not
    ↪necessarily the split vertices.
  friend pair<pt, pt> cut(pt e) {
    assert(!e->is_vert); e->expose_edge();
    assert(!e->p && e->is_path && e->c[2] == nullptr);
    pt l = e->c[0], r = e->c[1]; assert(l && r);
    e->c[0] = e->c[1] = nullptr; l->p = r->p = nullptr; //
      ↪disconnect
    l = l->meld_path_end();
    return {l, r};
  }
  // bring path to the top
  friend pt get_path(pt a, pt b) {
    assert(a->is_vert && b->is_vert);
    a->make_root(); b->expose();
    if (a == b) { assert(!b->p); return b; } // top path is
      ↪single node
    assert(!b->p->p); return b->p; // b is at end of path
  }
  // root at rt, get subtree at n
  friend pt get_subtree(pt rt, pt n) {
    assert(rt->is_vert && n->is_vert);
    rt->make_root(); n->expose(); return n;
  }
  // easy: just expose one and then the other
  friend pt lca(pt a, pt b) {
    assert(a->is_vert && b->is_vert);
    assert(connected(a,b));
    a->expose(); return b->expose() ?: b;
  }
};

int main() {
  cin.tie(0)->sync_with_stdio(0);
  ints(N);
  V<top_tree_node> nodes(N);
  F0R(i,N) {
    pt n = &nodes[i];
    n->is_path = n->is_vert = true;
    n->update();
  }
  V<top_tree_node> edges(N-1);
  F0R(i,N-1) {
    ints(u,v,t); --u,--v;
    edges[i].own_parity = t;
    link(&edges[i],&nodes[u],&nodes[v]);
  }
  ints(M);
  rep(M) {
    ints(id); --id;
    edges[id].expose_edge();
    edges[id].own_parity ^= 1; edges[id].update();
    ps(edges[id].best_path);
  }
}
```

Geometry (8)

8.1 Primitives

ComplexComp.h

Description: Allows you to sort complex numbers. 6f828b, 5 lines

```
#define x real()
#define y imag()
using P = complex<db>;
namespace std {
  bool operator<(P l,P r) { return mp(l.x,l.y)<mp(r.x,r.y); } }
```

PointShort.h

Description: Use in place of complex<T>. cefef8, 36 lines

```
using T = db; // or ll
const T EPS = 1e-9; // adjust as needed
using P = pair<T,T>; using vP = V<P>; using Line = pair<P,P>;
int sgn(T a) { return (a>EPS)-(a<-EPS); }
T sq(T a) { return a*a; }

T norm(P p) { return sq(p.f)+sq(p.s); }
T abs(P p) { return sqrt(norm(p)); }
T arg(P p) { return atan2(p.s,p.f); }
P conj(P p) { return P(p.f,-p.s); }
P perp(P p) { return P(-p.s,p.f); }
P dir(T ang) { return P(cos(ang),sin(ang)); }

P operator+(P l, P r) { return P(l.f+r.f,l.s+r.s); }
P operator-(P l, P r) { return P(l.f-r.f,l.s-r.s); }
P operator*(P l, T r) { return P(l.f*r,l.s*r); }
P operator/(P l, T r) { return P(l.f/r,l.s/r); }
P operator*(P l, P r) { // complex # multiplication
  return P(l.f*r.f-l.s*r.s,l.s*r.f+l.f*r.s); }
P operator/(P l, P r) { return l*conj(r)/norm(r); }

P unit(const P& p) { return p/abs(p); }
T dot(const P& a, const P& b) { return a.f*b.f+a.s*b.s; }
T dot(const P& p, const P& a, const P& b) { return dot(a-p,b-p)
  ↪; }
T cross(const P& a, const P& b) { return a.f*b.s-a.s*b.f; }
T cross(const P& p, const P& a, const P& b) {
  return cross(a-p,b-p); }
P reflect(const P& p, const Line& l) {
  P a = l.f, d = l.s-l.f;
  return a+conj((p-a)/d)*d; }
P foot(const P& p, const Line& l) {
  return (p+reflect(p,l))/(T)2; }
bool onSeg(const P& p, const Line& l) {
  return sgn(cross(l.f,l.s,p)) == 0 && sgn(dot(p,l.f,l.s)) <= 0
    ↪; }
ostream& operator<<(ostream& os, P p) {
  return os << "(" << p.f << "," << p.s << ")"; }
```

AngleCmp.h

Description: Sorts points in ccw order about origin in the same way as atan2, which returns real in $(-\pi, \pi]$ so points on negative x-axis come last.
"Point.h" 2df5fc, 12 lines

```
// WARNING: you will get unexpected results if you mistype this
  ↪ as bool instead of int
// -1 if lower half, 0 if origin, 1 if upper half
int half(P x) { return x.s != 0 ? sgn(x.s) : -sgn(x.f); }
bool angleCmp(P a, P b) { int A = half(a), B = half(b);
  return A == B ? cross(a,b) > 0 : A < B; }

/* Usage:
  * vP v;
  * sort(all(v),[](P a, P b) { return
    atan2(a.s,a.f) < atan2(b.s,b.f); });
  * sort(all(v),angleCmp); // should give same result
*/
```

SegDist.h

Description: computes distance between P and line (segment) AB
"Point.h" 0cb69a, 6 lines

```
T lineDist(const P& p, const Line& l) {
  return abs(cross(p,l.f,l.s))/abs(l.f-l.s); }
T segDist(const P& p, const Line& l) {
  if (dot(l.f,p,l.s) <= 0) return abs(p-l.f);
  if (dot(l.s,p,l.f) <= 0) return abs(p-l.s);
```

```
  return lineDist(p,l); }
```

SegIsect.h

Description: computes the intersection point(s) of line (segments) a and b

"Point.h" 7f6ba0, 26 lines

```
// {unique intersection point} if it exists
// {b.f,b.s} if input lines are the same
// empty if lines do not intersect
vP lineIsect(const Line& a, const Line& b) {
  T a0 = cross(a.f,a.s,b.f), a1 = cross(a.f,a.s,b.s);
  if (a0 == a1) return a0 == 0 ? vP{b.f,b.s} : vP{};
  return {(b.s*a0-b.f*a1)/(a0-a1)};
}

// point in interior of both segments a and b, if it exists
vP strictIsect(const Line& a, const Line& b) {
  T a0 = cross(a.f,a.s,b.f), a1 = cross(a.f,a.s,b.s);
  T b0 = cross(b.f,b.s,a.f), b1 = cross(b.f,b.s,a.s);
  if (sgn(a0)*sgn(a1) < 0 && sgn(b0)*sgn(b1) < 0)
    return {(b.s*a0-b.f*a1)/(a0-a1)};
  return {};
}

// intersection of segments, a and b may be degenerate
vP segIsect(const Line& a, const Line& b) {
  vP v = strictIsect(a,b); if (sz(v)) return v;
  set<P> s;
  #define i(x,y) if (onSeg(x,y)) s.ins(x)
  i(a.f,b); i(a.s,b); i(b.f,a); i(b.s,a);
  return {all(s)};
}
```

8.2 Polygons

PolygonCenArea.h

Description: centroid (center of mass) of a polygon with constant mass per unit area and SIGNED area

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

"Point.h" 4ca221, 7 lines

```
pair<P,T> cenArea(const vP& v) { assert(sz(v) >= 3);
  P cen{}; T area{};
  F0R(i,sz(v)) {
    int j = (i+1)%sz(v); T a = cross(v[i],v[j]);
    cen += a*(v[i]+v[j]); area += a; }
  return {cen/area/(T)3,area/2}; // area is SIGNED
}
```

InPolygon.h

Description: Tests whether point is inside, on, or outside of a polygon (returns -1, 0, or 1). Both CW and CCW polygons are ok.

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

"Point.h" 2ed683, 9 lines

```
int inPolygon(const P& p, const vP& poly) {
  int n = sz(poly), ans = 0;
  F0R(i,n) {
    P x = poly[i], y = poly[(i+1)%n]; if (x.s > y.s) swap(x,y);
    if (onSeg(p,{x,y})) return 0;
    ans ^= (x.s <= p.s && p.s < y.s && cross(p,x,y) > 0);
  }
  return ans ? -1 : 1;
}
```

ConvexHull.h

Description: top-bottom convex hull

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

"Point.h" 868655, 18 lines

```
pair<vi,vi> ulHull(const vP& v) {
  vi p(sz(v)), u, l; iota(all(p), 0);
  sort(all(p), [&v](int a, int b) { return v[a] < v[b]; });
  each(i,p) {
    #define ADDP(C, cmp) while (sz(C) > 1 && cross(\
      v[C[sz(C)-2]],v[C.bk],v[i]) cmp 0) C.pop_back(); C.pb(i);
    ADDP(u, >=); ADDP(l, <=);
  }
  return {u,l};
}
vi hullInd(const vP& v) {
  vi u,l; tie(u,l) = ulHull(v); if (sz(l) <= 1) return l;
  if (v[l[0]] == v[l[1]]) return {0};
  l.insert(end(l),1+rall(u)-1); return l;
}
vP hull(const vP& v) {
  vi w = hullInd(v); vP res; each(t,w) res.pb(v[t]);
  return res; }
```

ConvexHull2.h

Description: Graham Scan

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

"Point.h" 5758fd, 13 lines

```
vi hullInd(const vP& v) {
  int ind = int(min_element(all(v))-begin(v));
  vi cand, C{ind}; F0R(i,sz(v)) if (v[i] != v[ind]) cand.pb(i);
  sort(all(cand),[&](int a, int b) {
    // sort by angle, tiebreak by distance
    P x = v[a]-v[ind], y = v[b]-v[ind]; T t = cross(x,y);
    return t != 0 ? t > 0 : norm(x) < norm(y); });
  each(c,cand) {
    while (sz(C) > 1 && cross(v[end(C)[-2]],v[C.bk],v[c]) <= 0)
      C.pop_back();
    C.pb(c); }
  return C;
}
```

Diameter.h

Description: rotating calipers, gives greatest distance between two points in P

Time: $\mathcal{O}(N)$ given convex hull

"ConvexHull.h" ee92de, 9 lines

```
db diameter(vP P) {
  P = hull(P);
  int n = sz(P), ind = 1; db ans = 0;
  if (n > 1) F0R(i,n) for (int j = (i+1)%n;;ind = (ind+1)%n) {
    ckmax(ans,abs(P[i]-P[ind]));
    if (cross(P[j]-P[i],P[(ind+1)%n]-P[ind]) <= 0) break;
  }
  return ans;
}
```

LineHull.h

Description: `lineHull` accepts line and ccw convex polygon. If all vertices in poly lie to one side of the line, returns a vector of closest vertices to line as well as orientation of poly with respect to line (± 1 for above/below). Otherwise, returns the range of vertices that lie on or below the line. `extrVertex` returns the point of a hull with the max projection onto a line.

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

"Point.h" 40e5a6, 41 lines

```
using Line = AR<P,2>;
#define cmp(i,j) sgn(-dot(dir,poly[(i)%n]-poly[(j)%n]))
#define extr(i) cmp(i+1,i) >= 0 && cmp(i,i-1+n) < 0
int extrVertex(const vP& poly, P dir) {
  int n = sz(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo+1 < hi) {
    int m = (lo+hi)/2;
    if (extr(m)) return m;
    int ls = cmp(lo+1,lo), ms = cmp(m+1,m);
    (ls < ms || (ls == ms && ls == cmp(lo,m)) ? hi : lo) = m;
  }
  return lo;
}
vi same(Line line, const vP& poly, int a) {
  // points on same parallel as a
  int n = sz(poly); P dir = perp(line[0]-line[1]);
  if (cmp(a+n-1,a) == 0) return {(a+n-1)%n,a};
  if (cmp(a,a+1) == 0) return {a,(a+1)%n};
  return {a};
}
#define cmpL(i) sgn(cross(line[0],line[1],poly[i]))
pair<int,vi> lineHull(Line line, const vP& poly) {
  int n = sz(poly); assert(n>1);
  int endA = extrVertex(poly,perp(line[0]-line[1])); // lowest
  if (cmpL(endA) >= 0) return {1,same(line,poly,endA)};
  int endB = extrVertex(poly,perp(line[1]-line[0])); // highest
  if (cmpL(endB) <= 0) return {-1,same(line,poly,endB)};
  AR<int,2> res;
  F0R(i,2) {
    int lo = endA, hi = endB; if (hi < lo) hi += n;
    while (lo < hi) {
      int m = (lo+hi+1)/2;
      if (cmpL(m%n) == cmpL(endA)) lo = m;
      else hi = m-1;
    }
    res[i] = lo%n; swap(endA,endB);
  }
  if (cmpL((res[0]+1)%n) == 0) res[0] = (res[0]+1)%n;
  return {0,{(res[1]+1)%n,res[0]}};
}
```

HalfPlaneIsect2.h

Description: Returns vertices of half-plane intersection. A half-plane is the area to the left of a ray, which is defined by a point `p` and a direction `dp`. Area of intersection should be sufficiently precise when all inputs are integers with magnitude $\leq 10^5$. Assumes intersection is bounded (easiest way to ensure this is to uncomment the code below).

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

"AngleCmp.h" d9e261, 46 lines

```
struct Ray {
  P p, dp; // origin, direction
  P isect(const Ray& L) const {
    return p+dp*(cross(L.dp,L.p-p)/cross(L.dp,dp)); }
  bool operator<(const Ray& L) const {
    return angleCmp(dp,L.dp); }
};

vP halfPlaneIsect(V<Ray> _segs) {
  // int DX = 1e9, DY = 1e9; // bound input by rectangle [0,DX]
      ↪ x [0,DY]
  // _segs.pb({P{0,0},P{1,0}});
  // _segs.pb({P{DX,0},P{0,1}});
  // _segs.pb({P{DX,DY},P{-1,0}});
  // _segs.pb({P{0,DY},P{0,-1}});
  sor(_segs); // sort planes by angle
  V<Ray> segs; // remove parallel planes
  each(t,_segs) {
    if (!sz(segs) || segs.bk < t) { segs.pb(t); continue; }
    if (cross(t.dp,t.p-segs.bk.p) > 0) segs.bk = t;
  }
  auto bad = [&](const Ray& a, const Ray& b, const Ray& c) {
```

```
    P p1 = a.isect(b), p2 = b.isect(c);
    if (dot(p2-p1,b.dp) <= EPS) { // this EPS is required ...
      if (cross(a.dp,c.dp) <= 0) return 2; // isect(a,b,c) =
        ↪empty
      return 1;
    }
    return 0; // isect(a,c) == isect(a,b,c)
  };
  #define reduce(t) \
    while (sz(poly) > 1) { \
      int b = bad(poly.at(sz(poly)-2),poly.bk,t); \
      if (b == 2) return {}; \
      if (b == 1) poly.pop_back(); \
      else break; \
    }
  deque<Ray> poly;
  each(t,segs) { reduce(t); poly.pb(t); }
  for(;;poly.pop_front()) {
    reduce(poly[0]);
    if (!bad(poly.bk,poly[0],poly[1])) break;
  }
  assert(sz(poly) >= 3); // if you reach this point, area
    ↪should be nonzero
  vP poly_points; F0R(i,sz(poly))
    poly_points.pb(poly[i].isect(poly[(i+1)%sz(poly)]));
  return poly_points;
}
```

8.3 Circles

Circle.h

Description: represent circle as {center,radius}

"Point.h" 91f3fc, 6 lines

```
using Circ = pair<P,T>;
int in(const Circ& x, const P& y) { // -1 if inside, 0, 1
  return sgn(abs(y-x.f)-x.s); }
T arcLength(const Circ& x, P a, P b) {
    // precondition: a and b on x
  P d = (a-x.f)/(b-x.f); return x.s*acos(d.f); }
```

CircleIsect.h

Description: Circle intersection points and intersection area. Tangents will be returned twice.

"Circle.h" a0b0f8, 22 lines

```
vP isect(const Circ& x, const Circ& y) { // precondition: x!=y
  T d = abs(x.f-y.f), a = x.s, b = y.s;
  if (sgn(d) == 0) { assert(a != b); return {}; }
  T C = (a*a+d*d-b*b)/(2*a*d);
  if (abs(C) > 1+EPS) return {};
  T S = sqrt(max(1-C*C,(T)0)); P tmp = (y.f-x.f)/d*x.s;
  return {x.f+tmp*P(C,S),x.f+tmp*P(C,-S)};
}
vP isect(const Circ& x, const Line& y) {
  P c = foot(x.f,y); T sq_dist = sq(x.s)-norm(x.f-c);
  if (sgn(sq_dist) < 0) return {};
  P offset = unit(y.s-y.f)*sqrt(max(sq_dist,T(0)));
  return {c+offset,c-offset};
}
T isect_area(Circ x, Circ y) { // not thoroughly tested
  T d = abs(x.f-y.f), a = x.s, b = y.s; if (a < b) swap(a,b);
  if (d >= a+b) return 0;
  if (d <= a-b) return PI*b*b;
  T ca = (a*a+d*d-b*b)/(2*a*d), cb = (b*b+d*d-a*a)/(2*b*d);
  T s = (a+b+d)/2, h = 2*sqrt(s*(s-a)*(s-b)*(s-d))/d;
  return a*a*acos(ca)+b*b*acos(cb)-d*h;
}
```

CircleTangents.h

Description: internal and external tangents between two circles

"Circle.h" d9a76f, 22 lines

```
P tangent(P x, Circ y, int t = 0) {
  y.s = abs(y.s); // abs needed because internal calls y.s < 0
  if (y.s == 0) return y.f;
  T d = abs(x-y.f);
  P a = pow(y.s/d,2)*(x-y.f)+y.f;
  P b = sqrt(d*d-y.s*y.s)/d*y.s*unit(x-y.f)*dir(PI/2);
  return t == 0 ? a+b : a-b;
}
V<pair<P,P>> external(Circ x, Circ y) {
  V<pair<P,P>> v;
  if (x.s == y.s) {
    P tmp = unit(x.f-y.f)*x.s*dir(PI/2);
    v.eb(x.f+tmp,y.f+tmp);
    v.eb(x.f-tmp,y.f-tmp);
  } else {
    P p = (y.s*x.f-x.s*y.f)/(y.s-x.s);
    F0R(i,2) v.eb(tangent(p,x,i),tangent(p,y,i));
  }
  return v;
}
V<pair<P,P>> internal(Circ x, Circ y) {
  return external({x.f,-x.s},y); }
```

Circumcenter.h

Description: returns {circumcenter,circumradius}

"Circle.h" a2c6a6, 5 lines

```
Circ ccCenter(P a, P b, P c) {
  b -= a; c -= a;
  P res = b*c*(conj(c)-conj(b))/(b*conj(c)-conj(b)*c);
  return {a+res,abs(res)};
}
```

MinEnclosingCirc.h

Description: minimum enclosing circle

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

"Circumcenter.h" 53963d, 13 lines

```
circ mec(vP ps) {
  shuffle(all(ps), rng);
  P o = ps[0]; T r = 0, EPS = 1+1e-8;
  F0R(i,sz(ps)) if (abs(o-ps[i]) > r*EPS) {
    o = ps[i], r = 0; // point is on MEC
    F0R(j,i) if (abs(o-ps[j]) > r*EPS) {
      o = (ps[i]+ps[j])/2, r = abs(o-ps[i]);
      F0R(k,j) if (abs(o-ps[k]) > r*EPS)
        tie(o,r) = ccCenter(ps[i],ps[j],ps[k]);
    }
  }
  return {o,r};
}
```

8.4 Misc

ClosestPair.h

Description: line sweep to find two closest points

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

"Point.h" 2b60fa, 17 lines

```
pair<P,P> solve(vP v) {
  pair<db,pair<P,P>> bes; bes.f = INF;
  set<P> S; int ind = 0;
  sort(all(v));
  F0R(i,sz(v)) {
    if (i && v[i] == v[i-1]) return {v[i],v[i]};
    for (; v[i].f-v[ind].f >= bes.f; ++ind)
      S.erase({v[ind].s,v[ind].f});
    for (auto it = S.ub({v[i].s-bes.f,INF});
      it != end(S) && it->f < v[i].s+bes.f; ++it) {
      P t = {it->s,it->f};
      ckmin(bes,{abs(t-v[i]),{t,v[i]}});
    }
    S.insert({v[i].s,v[i].f});
  }
  return bes.s;
}
```

DelaunayIncremental.h

Description: Bowyer-Watson where not all points collinear. Works for $|x|, |y| \le 10^4$, assuming that all circumradii in final triangulation are $\ll 10^9$.

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

"DelaunayFast.h" 57c54d, 23 lines

```
// include inCircle from DelaunayFast

const T BIG = 1e9; // >>(10^4)^2
V<AR<int,3>> triIncrement(vP v) {
  v.pb({-BIG,-BIG}); v.pb({BIG,0}); v.pb({0,BIG});
  V<AR<int,3>> ret, tmp;
  ret.pb({sz(v)-3,sz(v)-2,sz(v)-1});
  F0R(i,sz(v)-3) {
    map<pi,int> m;
    each(a,ret) {
      if (inCircle(v[i],v[a[0]],v[a[1]],v[a[2]]))
        m[{a[0],a[1]}]++, m[{a[1],a[2]}]++, m[{a[0],a[2]}]++;
      else tmp.pb(a);
    }
    each(a,m) if (a.s == 1) {
      AR<int,3> x{a.f.f,a.f.s,i};
      sor(x); tmp.pb(x);
    }
    swap(ret,tmp); tmp.clear();
  }
  each(a,ret) if (a[2] < sz(v)-3) tmp.pb(a);
  return tmp;
}
```

DelaunayFast.h

Description: Fast Delaunay triangulation assuming no duplicates and not all points collinear (in latter case, result will be empty). Should work for doubles as well, though there may be precision issues in 'circ'. Returns triangles in ccw order. Each circumcircle will contain none of the input points. If coordinates are ints at most B then `T` should be large enough to support ints on the order of B^4.

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

"Point.h" 0e7085, 82 lines

```
// using T = ll; (if coords are < 2e4)
using lll = __int128;
// return true if p strictly within circumcircle(a,b,c)
bool inCircle(P p, P a, P b, P c) {
  a -= p, b -= p, c -= p; // assert(cross(a,b,c)>0);
  lll x = (lll)norm(a)*cross(b,c)+(lll)norm(b)*cross(c,a)
      +(lll)norm(c)*cross(a,b);
  return x*(cross(a,b,c)>0?1:-1) > 0;
}

P arb(LLONG_MAX,LLONG_MAX); // not equal to any other point
using Q = struct Quad*;
struct Quad {
  bool mark; Q o, rot; P p;
  P F() { return r()->p; }
  Q r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
```

```
};
Q makeEdge(P orig, P dest) {
  Q q[]{new Quad{0,0,0,orig}, new Quad{0,0,0,arb},
       new Quad{0,0,0,dest}, new Quad{0,0,0,arb}};
  F0R(i,4) q[i]->o = q[-i & 3], q[i]->rot = q[(i+1) & 3];
  return *q;
}
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 vP& s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.bk);
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = cross(s[0], 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) (cross(e->F(),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 ((cross(B->p,H(A)) < 0 && (A = A->next())) ||
         (cross(A->p,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 (inCircle(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e = t; \
    }
  while (1) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && inCircle(H(RC), H(LC))))
      base = connect(RC, base->r());
    else base = connect(base->r(), LC->r());
  }
  return {ra, rb};
}
V<AR<P,3>> triangulate(vP pts) {
  sor(pts); assert(unique(all(pts)) == end(pts)); // no
    ↪duplicates
  if (sz(pts) < 2) return {};
  Q e = rec(pts).f; V<Q> q = {e};
  while (cross(e->o->F(), e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.pb(c->p); \
  q.pb(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  int qi = 0; while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  V<AR<P,3>> ret(sz(pts)/3);
  F0R(i,sz(pts)) ret[i/3][i%3] = pts[i];
  return ret;
}
```

ManhattanMST.h

Description: Given N points, returns up to $4N$ edges which are guaranteed to contain a MST for graph with edge weights $w(p,q) = |p.x - q.x| + |p.y - q.y|$. Edges are in the form {dist, {src, dst}}.
Time: $\mathcal{O}(N \log N)$

"DSU.h" b7a3bd, 24 lines

```
// use standard MST algorithm on result to find final MST
V<pair<int,pi>> manhattanMst(vpi v) {
  vi id(sz(v)); iota(all(id),0);
  V<pair<int,pi>> ed;
  F0R(k,4) {
    sort(all(id),[&](int i, int j) {
      return v[i].f+v[i].s < v[j].f+v[j].s; });
    map<int,int> sweep; // find first octant neighbors
    each(i,id) { // those in sweep haven't found neighbor yet
      for (auto it = sweep.lb(-v[i].s);
        it != end(sweep); sweep.erase(it++)) {
        int j = it->s;
        pi d{v[i].f-v[j].f,v[i].s-v[j].s};if (d.s>d.f)break;
        ed.pb({d.f+d.s,{i,j}});
      }
      sweep[-v[i].s] = i;
    }
    each(p,v) {
      if (k&1) p.f *= -1;
      else swap(p.f,p.s);
    }
  }
  return ed;
}
```

8.5 3D

Point3D.h

Description: Basic 3D geometry.

"Point.h" 10a63a, 82 lines

```
using P3 = AR<T,3>; using Tri = AR<P3,3>; using vP3 = V<P3>;
T norm(const P3& x) {
  T sum = 0; F0R(i,3) sum += sq(x[i]);
  return sum; }
T abs(const P3& x) { return sqrt(norm(x)); }

P3& operator+=(P3& l, const P3& r) { F0R(i,3) l[i] += r[i];
  return l; }
P3& operator-=(P3& l, const P3& r) { F0R(i,3) l[i] -= r[i];
  return l; }
P3& operator*=(P3& l, const T& r) { F0R(i,3) l[i] *= r;
  return l; }
P3& operator/=(P3& l, const T& r) { F0R(i,3) l[i] /= r;
  return l; }
P3 operator-(P3 l) { l *= -1; return l; }
P3 operator+(P3 l, const P3& r) { return l += r; }
P3 operator-(P3 l, const P3& r) { return l -= r; }
P3 operator*(P3 l, const T& r) { return l *= r; }
P3 operator*(const T& r, const P3& l) { return l*r; }
P3 operator/(P3 l, const T& r) { return l /= r; }

P3 unit(const P3& x) { return x/abs(x); }
T dot(const P3& a, const P3& b) {
  T sum = 0; F0R(i,3) sum += a[i]*b[i];
  return sum; }
P3 cross(const P3& a, const P3& b) {
  return {a[1]*b[2]-a[2]*b[1],a[2]*b[0]-a[0]*b[2],
      a[0]*b[1]-a[1]*b[0]}; }
P3 cross(const P3& a, const P3& b, const P3& c) {
  return cross(b-a,c-a); }
P3 perp(const P3& a, const P3& b, const P3& c) {
  return unit(cross(a,b,c)); }

bool isMult(const P3& a, const P3& b) { // for long longs
  P3 c = cross(a,b); F0R(i,sz(c)) if (c[i] != 0) return 0;
  return 1; }
bool collinear(const P3& a, const P3& b, const P3& c) {
  return isMult(b-a,c-a); }

T DC(const P3&a,const P3&b,const P3&c,const P3&p) {
  return dot(cross(a,b,c),p-a); }
bool coplanar(const P3&a,const P3&b,const P3&c,const P3&p) {
  return DC(a,b,c,p) == 0; }
bool op(const P3& a, const P3& b) {
  int ind = 0; // going in opposite directions?
  FOR(i,1,3) if (std::abs(a[i]*b[i])>std::abs(a[ind]*b[ind]))
    ind = i;
  return a[ind]*b[ind] < 0;
}
// coplanar points, b0 and b1 on opposite sides of a0-a1?
bool opSide(const P3&a,const P3&b,const P3&c,const P3&d) {
  return op(cross(a,b,c),cross(a,b,d)); }
// coplanar points, is a in Triangle b
bool inTri(const P3& a, const Tri& b) {
  F0R(i,3)if(opSide(b[i],b[(i+1)%3],b[(i+2)%3],a))return 0;
  return 1; }

// point-seg dist
T psDist(const P3&p,const P3&a,const P3&b) {
  if (dot(a-p,a-b) <= 0) return abs(a-p);
  if (dot(b-p,b-a) <= 0) return abs(b-p);
  return abs(cross(p,a,b))/abs(a-b);
}
// projection onto line
P3 foot(const P3& p, const P3& a, const P3& b) {
  P3 d = unit(b-a); return a+dot(p-a,d)*d; }
// rotate p about axis
P3 rotAxis(const P3& p, const P3& a, const P3& b, T theta) {
  P3 dz = unit(b-a), f = foot(p,a,b);
  P3 dx = p-f, dy = cross(dz,dx);
  return f+cos(theta)*dx+sin(theta)*dy;
}
// projection onto plane
P3 foot(const P3& a, const Tri& b) {
  P3 c = perp(b[0],b[1],b[2]);
  return a-c*(dot(a,c)-dot(b[0],c)); }
// line-plane intersection
P3 lpIntersect(const P3&a0,const P3&a1,const Tri&b) {
  P3 c = unit(cross(b[2]-b[0],b[1]-b[0]));
  T x = dot(a0,c)-dot(b[0],c), y = dot(a1,c)-dot(b[0],c);
  return (y*a0-x*a1)/(y-x);
}
```

Hull3D.h

Description: Incremental 3D convex hull where not all points are coplanar. Normals to returned faces point outwards. If coordinates are ints at most B then T should be large enough to support ints on the order of B^3. Changes order of points.
Time: $\mathcal{O}(N^2)$, $\mathcal{O}(N \log N)$

292bc6, 92 lines

```
// using T = ll;
T above(const P3&a,const P3&b,const P3&c,const P3&p) {
  return DC(a,b,c,p) > 0; } // is p strictly above plane
void prep(vP3& p) { // rearrange points such that
  shuffle(all(p),rng); // first four are not coplanar
  int dim = 1;
  FOR(i,1,sz(p))
    if (dim == 1) {
      if (p[0] != p[i]) swap(p[1],p[i]), ++dim;
    } else if (dim == 2) {
```

```
    if (!collinear(p[0],p[1],p[i]))
      swap(p[2],p[i]), ++dim;
  } else if (dim == 3) {
    if (!coplanar(p[0],p[1],p[2],p[i]))
      swap(p[3],p[i]), ++dim;
  }
  assert(dim == 4);
}

using F = AR<int,3>; // face
V<F> hull3d(vP3& p) {
  // s.t. first four points form tetra
  prep(p); int N = sz(p); V<F> hull; // triangle for each face
  auto ad = [&](int a, int b, int c) { hull.pb({a,b,c}); };
  // +new face to hull
  ad(0,1,2), ad(0,2,1); // initialize hull as first 3 points
  V<vb> in(N,vb(N)); // is zero before each iteration
  FOR(i,3,N) { // incremental construction
    V<F> def, HULL; swap(hull,HULL);
    // HULL now contains old hull
    auto ins = [&](int a, int b, int c) {
      if (in[b][a]) in[b][a] = 0; // kill reverse face
      else in[a][b] = 1, ad(a,b,c);
    };
    each(f,HULL) {
      if (above(p[f[0]],p[f[1]],p[f[2]],p[i]))
        F0R(j,3) ins(f[j],f[(j+1)%3],i);
        // recalc all faces s.t. point is above face
      else def.pb(f);
    }
    each(t,hull) if (in[t[0]][t[1]]) // edge exposed,
      in[t[0]][t[1]] = 0, def.pb(t); // add a new face
    swap(hull,def);
  }
  return hull;
}
V<F> hull3dFast(vP3& p) {
  prep(p); int N = sz(p); V<F> hull;
  vb active; V<vi> rvis; V<AR<pi,3>> other;
  // whether face is active
  // points visible from each face
  // other face adjacent to each edge of face
  V<vi> vis(N); // faces visible from each point
  auto ad = [&](int a, int b, int c) {
    hull.pb({a,b,c}); active.pb(1); rvis.eb(); other.eb(); };
  auto ae = [&](int a, int b) { vis[b].pb(a), rvis[a].pb(b); };
  auto abv = [&](int a, int b) {
    F f=hull[a]; return above(p[f[0]],p[f[1]],p[f[2]],p[b]);};
  auto edge = [&](pi e) -> pi {
    return {hull[e.f][e.s],hull[e.f][(e.s+1)%3]}; };
  auto glue = [&](pi a, pi b) { // link two faces by an edge
    pi x = edge(a); assert(edge(b) == mp(x.s,x.f));
    other[a.f][a.s] = b, other[b.f][b.s] = a;
  }; // ensure face 0 is removed when i=3
  ad(0,1,2), ad(0,2,1); if (abv(1,3)) swap(p[1],p[2]);
  F0R(i,3) glue({0,i},{1,2-i});
  FOR(i,3,N) ae(abv(1,i),i); // coplanar points go in rvis[0]
  vi label(N,-1);
  FOR(i,3,N) { // incremental construction
    vi rem; each(t,vis[i]) if (active[t]) active[t]=0, rem.pb(t
      ↪);
    if (!sz(rem)) continue; // hull unchanged
    int st = -1;
    each(r,rem) F0R(j,3) {
      int o = other[r][j].f;
      if (active[o]) { // create new face!
        int a,b; tie(a,b) = edge({r,j}); ad(a,b,i); st = a;
        int cur = sz(rvis)-1; label[a] = cur;
        vi tmp; set_union(all(rvis[r]),all(rvis[o]),
                  back_inserter(tmp));
        // merge sorted vectors ignoring duplicates
        each(x,tmp) if (abv(cur,x)) ae(cur,x);
        glue({cur,0},other[r][j]); // glue old w/ new face
      }
    }
    for (int x = st, y; ; x = y) { // glue new faces together
      int X = label[x]; glue({X,1},{label[y=hull[X][1]],2});
      if (y == st) break;
    }
  }
  V<F> ans; F0R(i,sz(hull)) if (active[i]) ans.pb(hull[i]);
  return ans;
}
```

PolySaVol.h

Description: surface area and volume of polyhedron, normals to faces must point outwards

"Hull3D.h" 52fc2b, 8 lines

```
pair<T,T> SaVol(vP3 p, V<F> faces) {
  T s = 0, v = 0;
  each(i,faces) {
    P3 a = p[i[0]], b = p[i[1]], c = p[i[2]];
    s += abs(cross(a,b,c)); v += dot(cross(a,b),c);
  }
  return {s/2,v/6};
}
```

Delaunay3.h

Description: Delaunay triangulation with 3D hull. Fails when all points collinear. If coordinates are ints at most B, T should be large enough to support ints on the order of B^4.

"Point.h", "Hull3D.h", "AngleCmp.h" 14907e, 15 lines

```
V<AR<P,3>> triHull(vP p) {
  V<P3> p3; V<AR<P,3>> res; each(x,p) p3.pb({x.f,x.s,norm(x)});
  bool ok = 0; each(t,p3) ok |= !coplanar(p3[0],p3[1],p3[2],t);
  if (!ok) { // all points concyclic
    sort(1+all(p),[&p](P a, P b) {
      return cross(a-p[0],b-p[0])>0; });
    FOR(i,1,sz(p)-1) res.pb({p[0],p[i],p[i+1]});
  } else {
    #define nor(x) P(p3[x][0],p3[x][1])
    each(t,hull3dFast(p3))
      if (dot(cross(p3[t[0]],p3[t[1]],p3[t[2]]),{0,0,1}) < 0)
        res.pb({nor(t[0]),nor(t[2]),nor(t[1])});
  }
  return res;
}
```

Strings (9)

9.1 Light

KMP.h

Description: f[i] is length of the longest proper suffix of the i-th prefix of s that is a prefix of s

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

4538e4, 13 lines

```
vi kmp(str s) {
  int N = sz(s); vi f(N+1); f[0] = -1;
  FOR(i,1,N+1) {
    for (f[i]=f[i-1];f[i]!=-1&&s[f[i]]!=s[i-1];)f[i]=f[f[i]];
    ++f[i]; }
  return f;
}
vi getOc(str a, str b) { // find occurrences of a in b
  vi f = kmp(a+"@"+b), ret;
  FOR(i,sz(a),sz(b)+1) if (f[i+sz(a)+1] == sz(a))
    ret.pb(i-sz(a));
  return ret;
}
```

Z.h

Description: f[i] is the max len such that s.substr(0,len) == s.substr(i,len)

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

566170, 15 lines

```
vi z(str s) {
  int N = sz(s), L = 1, R = 0; s += '#';
  vi ans(N); ans[0] = N;
  FOR(i,1,N) {
    if (i <= R) ans[i] = min(R-i+1,ans[i-L]);
    while (s[i+ans[i]] == s[ans[i]]) ++ans[i];
    if (i+ans[i]-1 > R) L = i, R = i+ans[i]-1;
  }
  return ans;
}
vi getPrefix(str a, str b) { // find prefixes of a in b
  vi t = z(a+b); t = vi(sz(a)+all(t));
  each(u,t) ckmin(u,sz(a));
  return t;
}
```

Manacher.h

Description: length of largest palindrome centered at each character of string and between every consecutive pair

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

fcc3f7, 13 lines

```
vi manacher(str _S) {
  str S = "@"; each(c,_S) S += c, S += "#";
  S.bk = '&';
  vi ans(sz(S)-1); int lo = 0, hi = 0;
  FOR(i,1,sz(S)-1) {
    if (i != 1) ans[i] = min(hi-i,ans[hi-i+lo]);
    while (S[i-ans[i]-1] == S[i+ans[i]+1]) ++ans[i];
    if (i+ans[i] > hi) lo = i-ans[i], hi = i+ans[i];
  }
  ans.erase(begin(ans));
  F0R(i,sz(ans)) if (i%2 == ans[i]%2) ++ans[i];
  return ans;
}
```

LyndonFactor.h

Description: A string is "simple" if it is strictly smaller than any of its own nontrivial suffixes. The Lyndon factorization of the string s is a factorization $s = w_1 w_2 \dots w_k$ where all strings w_i are simple and $w_1 \geq w_2 \geq \cdots \geq w_k$. Min rotation gets min index i such that cyclic shift of s starting at i is minimum.

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

40c2f1, 19 lines

```
vs duval(str s) {
  int N = sz(s); vs factors;
  for (int i = 0; i < N; ) {
    int j = i+1, k = i;
    for (; j < n && s[k] <= s[j]; j++) {
      if (s[k] < s[j]) k = i;
      else ++k;
    }
    for (; i <= k; i += j-k) factors.pb(s.substr(i,j-k));
  }
  return factors;
}
int minRotation(str s) {
  int n = sz(s); s += s;
  auto d = duval(s); int ind = 0, ans = 0;
```

```
  while (ans+sz(d[ind]) < n) ans += sz(d[ind++]);
  while (ind && d[ind] == d[ind-1]) ans -= sz(d[ind--]);
  return ans;
}
```

HashRange.h

Description: Polynomial hash for substrings with two bases.

fc0b90, 24 lines

```
using H = AR<int,2>; // bases not too close to ends
H makeH(char c) { return {c,c}; }
uniform_int_distribution<int> BDIST(0.1*MOD,0.9*MOD);
const H base{BDIST(rng),BDIST(rng)};
H operator+(H l, H r) {
  F0R(i,2) if ((l[i] += r[i]) >= MOD) l[i] -= MOD;
  return l; }
H operator-(H l, H r) {
  F0R(i,2) if ((l[i] -= r[i]) < 0) l[i] += MOD;
  return l; }
H operator*(H l, H r) {
  F0R(i,2) l[i] = (ll)l[i]*r[i]%MOD;
  return l; }

V<H> pows{{1,1}};
struct HashRange {
  str S; V<H> cum{{}};
  void add(char c) { S += c; cum.pb(base*cum.bk+makeH(c)); }
  void add(str s) { each(c,s) add(c); }
  void extend(int len) { while (sz(pows) <= len)
    pows.pb(base*pows.bk); }
  H hash(int l, int r) { int len = r+1-l; extend(len);
    return cum[r+1]-pows[len]*cum[l]; }
};
```

ReverseBW.h

Description: Used only once. Burrows-Wheeler Transform appends # to a string, sorts the rotations of the string in increasing order, and constructs a new string that contains the last character of each rotation. This function reverses the transform.

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

e400d8, 7 lines

```
str reverseBW(str t) {
  vi nex(sz(t)); iota(all(nex),0);
  stable_sort(all(nex),[&t](int a,int b){return t[a]<t[b];});
  str ret; for (int i = nex[0]; i; )
    ret += t[i = nex[i]];
  return ret;
}
```

ACfixed.h

Description: Aho-Corasick for fixed alphabet. For each prefix, stores link to max length suffix which is also a prefix.

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

c0b364, 28 lines

```
struct ACfixed { // fixed alphabet
  static const int ASZ = 26;
  struct node { AR<int,ASZ> to; int link; };
  V<node> d = {{}};
  int add(str s) { // add word
    int v = 0;
    each(C,s) {
      int c = C-'a';
      if (!d[v].to[c]) d[v].to[c] = sz(d), d.eb();
      v = d[v].to[c];
    }
    return v;
  }
  void init() { // generate links
    d[0].link = -1;
    queue<int> q; q.push(0);
    while (sz(q)) {
      int v = q.ft; q.pop();
      F0R(c,ASZ) {
        int u = d[v].to[c]; if (!u) continue;
        d[u].link = d[v].link == -1 ? 0 : d[d[v].link].to[c];
        q.push(u);
      }
      if (v) F0R(c,ASZ) if (!d[v].to[c])
        d[v].to[c] = d[d[v].link].to[c];
    }
  }
};
```

SuffixArray.h

Description: Sort suffixes. First element of `sa` is `sz(S)`, `isa` is the inverse of `sa`, and `lcp` stores the longest common prefix between every two consecutive elements of `sa`.

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

"RMQ.h" 27a566, 30 lines

```
struct SuffixArray {
  str S; int N; vi sa, isa, lcp;
  void init(str _S) { N = sz(S = _S)+1; genSa(); genLcp(); }
  void genSa() { // sa has size sz(S)+1, starts with sz(S)
    sa = isa = vi(N); sa[0] = N-1; iota(1+all(sa),0);
    sort(1+all(sa),[&](int a, int b) { return S[a] < S[b]; });
    FOR(i,1,N) { int a = sa[i-1], b = sa[i];
      isa[b] = i > 1 && S[a] == S[b] ? isa[a] : i; }
    for (int len = 1; len < N; len *= 2) { // currently sorted
      // by first len chars
      vi s(sa), is(isa), pos(N); iota(all(pos),0);
      each(t,s) {int T=t-len;if (T>=0) sa[pos[isa[T]]++] = T;}
      FOR(i,1,N) { int a = sa[i-1], b = sa[i];
        isa[b] = is[a]==is[b]&&is[a+len]==is[b+len]?isa[a]:i; }
    }
  }
  void genLcp() { // Kasai's Algo
    lcp = vi(N-1); int h = 0;
    F0R(b,N-1) { int a = sa[isa[b]-1];
      while (a+h < sz(S) && S[a+h] == S[b+h]) ++h;
      lcp[isa[b]-1] = h; if (h) h--; }
    R.init(lcp);
  }
  RMQ<int> R;
  int getLCP(int a, int b) { // lcp of suffixes starting at a,b
    if (a == b) return sz(S)-a;
    int l = isa[a], r = isa[b]; if (l > r) swap(l,r);
    return R.query(l,r-1);
  }
};
```

SuffixArrayLinear.h

Description: Linear Time Suffix Array

ed0bb4, 46 lines

```
vi sa_is(const vi& s, int upper) {
  int n = sz(s); if (!n) return {};
  vi sa(n); vb ls(n);
  R0F(i,n-1) ls[i] = s[i] == s[i+1] ? ls[i+1] : s[i] < s[i+1];
  vi sum_l(upper), sum_s(upper);
  F0R(i,n) (ls[i] ? sum_l[s[i]+1] : sum_s[s[i]])++;
  F0R(i,upper) {
    if (i) sum_l[i] += sum_s[i-1];
    sum_s[i] += sum_l[i];
  }
  auto induce = [&](const vi& lms) {
    fill(all(sa),-1);
    vi buf = sum_s;
    for (int d: lms) if (d != n) sa[buf[s[d]]++] = d;
    buf = sum_l; sa[buf[s[n-1]]++] = n-1;
    F0R(i,n) {
      int v = sa[i]-1;
      if (v >= 0 && !ls[v]) sa[buf[s[v]]++] = v;
    }
    buf = sum_l;
    R0F(i,n) {
      int v = sa[i]-1;
      if (v >= 0 && ls[v]) sa[--buf[s[v]+1]] = v;
    }
  };
  vi lms_map(n+1,-1), lms; int m = 0;
  FOR(i,1,n) if (!ls[i-1] && ls[i]) lms_map[i]=m++, lms.pb(i);
  induce(lms); // sorts LMS prefixes
  vi sorted_lms;each(v,sa)if (lms_map[v]!=-1)sorted_lms.pb(v);
  vi rec_s(m); int rec_upper = 0; // smaller subproblem
  FOR(i,1,m) { // compare two lms substrings in sorted order
    int l = sorted_lms[i-1], r = sorted_lms[i];
    int end_l = lms_map[l]+1 < m ? lms[lms_map[l]+1] : n;
    int end_r = lms_map[r]+1 < m ? lms[lms_map[r]+1] : n;
    bool same = 0; // whether lms substrings are same
    if (end_l-l == end_r-r) {
      for (;l < end_l && s[l] == s[r]; ++l,++r);
      if (l != n && s[l] == s[r]) same = 1;
    }
    rec_s[lms_map[sorted_lms[i]]] = (rec_upper += !same);
  }
  vi rec_sa = sa_is(rec_s,rec_upper+1);
  F0R(i,m) sorted_lms[i] = lms[rec_sa[i]];
  induce(sorted_lms); // sorts LMS suffixes
  return sa;
}
```

TandemRepeats.h

Description: Find all (i,p) such that `s.substr(i,p) == s.substr(i+p,p)`. No two intervals with the same period intersect or touch.

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

"SuffixArray.h" ca7fb6, 13 lines

```
V<array<int,3>> solve(str s) {
  int N = sz(s); SuffixArray A,B;
  A.init(s); reverse(all(s)); B.init(s);
  V<array<int,3>> runs;
  for (int p = 1; 2*p <= N; ++p) { // do in O(N/p) for period p
    for (int i = 0, lst = -1; i+p <= N; i += p) {
      int l = i-B.getLCP(N-i-p,N-i), r = i-p+A.getLCP(i,i+p);
      if (l > r || l == lst) continue;
      runs.pb({lst = l,r,p}); // for each i in [l,r],
    } // s.substr(i,p) == s.substr(i+p,p)
  }
  return runs;
} // ps(solve("aaabababa"));
```

9.2 Heavy

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; }
};
```

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
}
```

SuffixTree.h

Description: Used infrequently. Ukkonen's algorithm for suffix tree. Longest non-unique suffix of `s` has length `len[p]+lef` after each call to `add` terminates. Each iteration of loop within `add` decreases this quantity by one.
Time: $\mathcal{O}(N \log \sum)$

39751c, 51 lines

```
struct SuffixTree {
  str s; int N = 0;
  vi pos, len, lnk; V<map<char,int>> to;
  int make(int POS, int LEN) { // lnk[x] is meaningful when
    // x!=0 and len[x] != MOD
    pos.pb(POS);len.pb(LEN);lnk.pb(-1);to.eb();return N++; }
  void add(int& p, int& lef, char c) { // longest
    // non-unique suffix is at node p with lef extra chars
    s += c; ++lef; int lst = 0;
    for (;lef;p?p=lnk[p]:lef--) { // if p != root then lnk[p]
      // must be defined
      while (lef>1 && lef>len[to[p][s[sz(s)-lef]]])
        p = to[p][s[sz(s)-lef]], lef -= len[p];
      // traverse edges of suffix tree while you can
      char e = s[sz(s)-lef]; int& q = to[p][e];
      // next edge of suffix tree
      if (!q) q = make(sz(s)-lef,MOD), lnk[lst] = p, lst = 0;
      // make new edge
      else {
        char t = s[pos[q]+lef-1];
        if (t == c) { lnk[lst] = p; return; } // suffix not
          ↪unique
        int u = make(pos[q],lef-1);
        // new node for current suffix-1, define its link
        to[u][c] = make(sz(s)-1,MOD); to[u][t] = q;
        // new, old nodes
        pos[q] += lef-1; if (len[q] != MOD) len[q] -= lef-1;
        q = u, lnk[lst] = u, lst = u;
      }
    }
  }
  void init(str _s) {
    make(-1,0); int p = 0, lef = 0;
    each(c,_s) add(p,lef,c);
    add(p,lef,'$'); s.pop_back(); // terminal char
  }
  int maxPre(str x) { // max prefix of x which is substring
    for (int p = 0, ind = 0;;) {
      if (ind == sz(x) || !to[p].count(x[ind])) return ind;
      p = to[p][x[ind]];
      F0R(i,len[p]) {
        if (ind == sz(x) || x[ind] != s[pos[p]+i]) return ind;
        ind ++;
      }
    }
  }
  vi sa; // generate suffix array
  void genSa(int x = 0, int Len = 0) {
    if (!sz(to[x])) sa.pb(pos[x]-Len); // found terminal node
    else each(t,to[x]) genSa(t.s,Len+len[x]);
  }
};
```

Various (10)

10.1 Dynamic programming

When doing DP on intervals:
$a[i][j] = \min_{i<k<j}(a[i][k] + a[k][j]) + f(i,j)$, where the (minimal) optimal k increases with both i and j,

- one can solve intervals in increasing order of length, and search $k = p[i][j]$ for $a[i][j]$ only between $p[i][j-1]$ and $p[i+1][j]$.
- This is known as Knuth DP. Sufficient criteria for this are if $f(b,c) \le f(a,d)$ and $f(a,c) + f(b,d) \le f(a,d) + f(b,c)$ for all $a \le b \le c \le d$.
- Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.

CircularLCS.h

Description: Used only twice. For strs A, B calculates longest common subsequence of A with all rotations of B
Time: $\mathcal{O}(|A| \cdot |B|)$

db21cf, 26 lines

```
int circular_lcs(str A, str B) {
  B += B;
  int max_lcs = 0;
  V<vb> dif_left(sz(A)+1,vb(sz(B)+1)), dif_up(sz(A)+1,vb(sz(B)
    ↪+1));
```

```
  auto recalc = [&](int x, int y) { assert(x && y);
    int res = (A.at(x-1) == B.at(y-1)) |
      dif_up[x][y-1] | dif_left[x-1][y];
    dif_left[x][y] = res-dif_up[x][y-1];
    dif_up[x][y]   = res-dif_left[x-1][y];
  };
  FOR(i,1,sz(A)+1) FOR(j,1,sz(B)+1) recalc(i,j);
  F0R(j,sz(B)/2) {
    // 1. zero out dp[.][j], update dif_left and dif_right
    if (j) for (int x = 1, y = j; x <= sz(A) && y <= sz(B); ) {
      int pre_up = dif_up[x][y];
      if (y == j) dif_up[x][y] = 0;
      else recalc(x,y);
      (pre_up == dif_up[x][y]) ? ++x : ++y;
    }
    // 2. calculate LCS(A[0:sz(A)),B[j:j+sz(B)/2))
    int cur_lcs = 0;
    FOR(x,1,sz(A)+1) cur_lcs += dif_up[x][j+sz(B)/2];
    ckmax(max_lcs,cur_lcs);
  }
  return max_lcs;
}
```

SMAWK.h

Description: Given negation of totally monotone matrix with entries of type `D`, find indices of row maxima (their indices increase for every submatrix). If tie, take lesser index. `f` returns matrix entry at (r, c) in $O(1)$. Use in place of divide & conquer to remove a log factor.

Time: $\mathcal{O}(R+C)$, can be reduced to $\mathcal{O}(C(1+\log R/C))$ evaluations of f 0a9abb, 25 lines

```
template<class F, class D=ll> vi smawk (F f, vi x, vi y) {
  vi ans(sz(x),-1); // x = rows, y = cols
  #define upd() if (ans[i] == -1 || w > mx) ans[i] = c, mx = w
  if (min(sz(x),sz(y)) <= 8) {
    F0R(i,sz(x)) { int r = x[i]; D mx;
      each(c,y) { D w = f(r,c); upd(); } }
    return ans;
  }
  if (sz(x) < sz(y)) { // reduce subset of cols to consider
    vi Y; each(c,y) {
      for (;sz(Y);Y.pop_back()) { int X = x[sz(Y)-1];
        if (f(X,Y.bk) >= f(X,c)) break; }
      if (sz(Y) < sz(x)) Y.pb(c);
    } y = Y;
  } // recurse on half the rows
  vi X; for (int i = 1; i < sz(x); i += 2) X.pb(x[i]);
  vi ANS = smawk(f,X,y); F0R(i,sz(ANS)) ans[2*i+1] = ANS[i];
  for (int i = 0, k = 0; i < sz(x); i += 2){
    int to = i+1 < sz(ans) ? ans[i+1] : y.bk; D mx;
    for(int r = x[i];;++k) {
      int c = y[k]; D w = f(r,c); upd();
      if (c == to) break; }
  }
  return ans;
};
```

10.2 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` violations 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`).

10.3 Optimization tricks

10.3.1 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.

10.3.2 Pragmas

- `#pragma GCC optimize ("Ofast")` will make GCC auto-vectorize for loops and optimizes floating points better (assumes associativity and turns off denormals).
- `#pragma GCC target ("avx,avx2")` can double performance of vectorized code, but causes crashes on old machines. Also consider older `#pragma GCC target ("sse4")`.
- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).

FastIO.h

Description: Fast input and output.

Time: input is ~300ms faster for 10^6 long longs on CF 6a2f14, 37 lines

```
namespace FastIO {
  const int BSZ = 1<<15; ////// INPUT
  char ibuf[BSZ]; int ipos, ilen;
  char nc() { // next char
    if (ipos == ilen) {
      ipos = 0; ilen = fread(ibuf,1,BSZ,stdin);
      if (!ilen) return EOF;
    }
    return ibuf[ipos++];
  }
  void rs(str& x) { // read str
    char ch; while (isspace(ch = nc()));
    do { x += ch; } while (!isspace(ch = nc()) && ch != EOF);
  }
  tcT> void ri(T& x) { // read int or ll
    char ch; int sgn = 1;
    while (!isdigit(ch = nc())) if (ch == '-') sgn *= -1;
    x = ch-'0'; while (isdigit(ch = nc())) x = x*10+(ch-'0');
    x *= sgn;
  }
  tcT, class... Ts> void ri(T& t, Ts&... ts) {
    ri(t); ri(ts...); } // read ints
  ////// OUTPUT (call initO() at start)
  char obuf[BSZ], numBuf[100]; int opos;
  void flushOut() { fwrite(obuf,1,opos,stdout); opos = 0; }
  void wc(char c) { // write char
    if (opos == BSZ) flushOut();
    obuf[opos++] = c; }
  void ws(str s) { each(c,s) wc(c); } // write str
  tcT> void wi(T x, char after = '\0') {
    if (x < 0) wc('-'), x *= -1;
    int len = 0; for (;x>=10;x/=10) numBuf[len++] = '0'+(x%10);
    wc('0'+x); R0F(i,len) wc(numBuf[i]);
    if (after) wc(after);
  }
  void initO() { assert(atexit(flushOut) == 0); }
}
```

10.4 Other languages

Python3.py

Description: not PyPy3, solves CF Factorisation Collaboration 47 lines

```
from math import *
import sys, random
def nextInt():
  return int(input())
def nextStrs():
  return input().split()
def nextInts():
  return list(map(int,nextStrs()))

n = nextInt()
v = [n]
def process(x):
  global v
  x = abs(x)
  V = []
  for t in v: # print(type(t)) -> <class 'int'>
    g = gcd(t,x)
    if g != 1:
      V.append(g)
    if g != t:
      V.append(t//g)
  v = V
for i in range(50):
  x = random.randint(0,n-1)
  if gcd(x,n) != 1:
    process(x)
  else:
    sx = x*x%n # assert(gcd(sx,n) == 1)
    print(f"sqrt {sx}") # print value of var
    sys.stdout.flush()
    X = nextInt()
    process(x+X)
    process(x-X)
print(f'! {len(v)}',end='')
for i in v:
  print(f' {i}',end='')
print()
sys.stdout.flush() # sys.exit(0) -> exit
# sys.setrecursionlimit(int(1e9)) -> stack size
# print(f'{ans:=.6f}') -> print ans to 6 decimal places

from decimal import * # arbitrary precision decimals

ctx = getcontext()
ctx.prec = 28
print(Decimal(1) / Decimal(7)) # 0.1428571428571428571428571429
print(ctx.power(Decimal(10),-30)) # 1E-30
```