



Massachusetts Institute of Technology

# MIT NULL

Andrew He, Kevin Sun, Yinzhan Xu

adapted from KTH ACM Contest Template Library

2019-03-23

Contest (1)

template.cpp15 lines

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

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define trav(a, x) for(auto& a : x)
#define all(x) x.begin(), x.end()
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

int main() {
    cin.sync_with_stdio(0); cin.tie(0);
    cin.exceptions(cin.failbit);
}
```

hash.sh1 lines

```
tr -d '[:space:]' | md5sum
```

hash-cpp.sh1 lines

```
cpp -P -fpreprocessed | tr -d '[:space:]' | md5sum
```

Makefile37 lines

```
CXX = g++
CXXFLAGS = -O2 -std=gnu++14 -Wall -Wextra -Wno-unused-
↳result -pedantic -Wshadow -Wformat=2 -Wfloat-equal -
↳Wconversion -Wlogical-op -Wshift-overflow=2 -
↳Wduplicated-cond -Wcast-qual -Wcast-align
# pause:#pragma GCC diagnostic {ignored|warning} "-Wshadow
↳"
DEBUGFLAGS = -D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC -
↳fsanitize=address -fsanitize=undefined -fno-sanitize-
↳recover=all -fstack-protector -D_FORTIFY_SOURCE=2
CXXFLAGS += $(DEBUGFLAGS) # flags with speed penalty

TARGET := $(notdir $(CURDIR))
EXECUTE := ./$(TARGET)

CASES := $(sort $(basename $(wildcard *.in)))
TESTS := $(sort $(basename $(wildcard *.out)))

all: $(TARGET)

clean:
    -rm -rf $(TARGET) *.res

%: %.cpp
    $(LINK.cpp) $< $(LOADLIBES) $(LDLIBS) -o $@

run: $(TARGET)
    time $(EXECUTE)

%.res: $(TARGET) %.in
    time $(EXECUTE) < $*.in > $*.res

%.out: %

test_%: %.res %.out
    diff $*.res $*.out
```

```
runs: $(patsubst %,%.res,$(CASES))
test: $(patsubst %,test_%,$(TESTS))
```

```
.PHONY: all clean run test test_% runs
```

```
.PRECIOUS: %.res
```

vimrc8 lines

```
set nosp ai bs=2 hls ic is lbr ls=2 mouse=a nu ru sc scs
↳smd so=3 sw=4 ts=4
filetype plugin indent on
syn on
map gA m'ggVG"+y'

com -range=% -nargs=1 P exe "<line1>,<line2>!".<q-args> |y
↳|sil u|echom @"
com -range=% Hash <line1>,<line2>P tr -d '[:space:]' |
↳md5sum
au FileType cpp com! -buffer -range=% Hash <line1>,<line2>
↳P cpp -P -fpreprocessed | tr -d '[:space:]' | md5sum
```

nanorc3 lines

```
set tabsize 4
set const
set autoindent
```

Data structures (2)

LCT.cppDescription: Link-Cut Tree.106 lines

```
struct T {
    bool rr;
    T *son[2], *pf, *fa;
} fl[N], *ff = fl, *f[N], *null;

void downdate(T *x) {
    if (x -> rr) {
        x -> son[0] -> rr = !x -> son[0] -> rr;
        x -> son[1] -> rr = !x -> son[1] -> rr;
        swap(x -> son[0], x -> son[1]);
        x -> rr = false;
    }
    // add stuff
}

void update(T *x) {
    // add stuff
}

void rotate(T *x, bool t) { // hash-cpp-1
    T *y = x -> fa, *z = y -> fa;
    if (z != null) z -> son[z -> son[1] == y] = x;
    x -> fa = z;
    y -> son[t] = x -> son[!t];
    x -> son[!t] -> fa = y;
    x -> son[!t] = y;
    y -> fa = x;
    update(y);
} // hash-cpp-1 = 28958e1067126a5892dcaa67307d2f1d

void xiao(T *x) {
    if (x -> fa != null) xiao(x -> fa), x -> pf = x -> fa
↳-> pf;
```

```
downdate(x);
}

void splay(T *x) { // hash-cpp-2
    xiao(x);
    T *y, *z;
    while (x -> fa != null) {
        y = x -> fa; z = y -> fa;
        bool t1 = (y -> son[1] == x), t2 = (z -> son[1] == y);
        if (z != null) {
            if (t1 == t2) rotate(y, t2), rotate(x, t1);
            else rotate(x, t1), rotate(x, t2);
        }else rotate(x, t1);
    }
    update(x);
} // hash-cpp-2 = 0bc1a3b77275f92cebc947211444fdb7

void access(T *x) { // hash-cpp-3
    splay(x);
    x -> son[1] -> pf = x;
    x -> son[1] -> fa = null;
    x -> son[1] = null;
    update(x);
    while (x -> pf != null) {
        splay(x -> pf);
        x -> pf -> son[1] -> pf = x -> pf;
        x -> pf -> son[1] -> fa = null;
        x -> pf -> son[1] = x;
        x -> fa = x -> pf;
        splay(x);
    }
    x -> rr = true;
} // hash-cpp-3 = db89159f01a2099d67e93163c3bfa384

bool Cut(T *x, T *y) { // hash-cpp-4
    access(x);
    access(y);
    downdate(y);
    downdate(x);
    if (y -> son[1] != x || x -> son[0] != null)
        return false;
    y -> son[1] = null;
    x -> fa = x -> pf = null;
    update(x);
    update(y);
    return true;
} // hash-cpp-4 = 42850d63565f84698378e8c2c23df1fe

bool Connected(T *x, T *y) {
    access(x);
    access(y);
    return x == y || x -> fa != null;
}

bool Link(T *x, T *y) {
    if (Connected(x, y))
        return false;
    access(x);
    access(y);
    x -> pf = y;
    return true;
}

int main() {
    read(n); read(m); read(q);
    null = new T; null -> son[0] = null -> son[1] = null ->
↳fa = null -> pf = null;
    for (int i = 1; i <= n; i++) {
```

```
f[i] = ++ff;
f[i] -> son[0] = f[i] -> son[1] = f[i] -> fa = f[i] ->
    ↪ pf = null;
f[i] -> rr = false;
}
// init null and f[i]
}
```

### LineContainer.h

**Description:** Container where you can add lines of the form  $kx+m$ , and query maximum values at points  $x$ . Useful for dynamic programming.

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

32 lines

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

struct LineContainer : multiset<Line> {
    // (for doubles, use inf = 1/.0, div(a,b) = a/b)
    const ll inf = LLONG_MAX;
    ll div(ll a, ll b) { // floored division
        return a / b - ((a ^ b) < 0 && a % b); }
    bool isect(iterator x, iterator y) {
        if (y == end()) { x->p = inf; return false; }
        if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
        else x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    }
    void add(ll k, ll m) {
        auto z = insert({k, m, 0}); y = z++, x = y;
        while (isect(y, z)) z = erase(z);
        if (x != begin() && isect(--x, y)) isect(x, y = erase(
            ↪y));
        while ((y = x) != begin() && (--x)->p >= y->p)
            isect(x, erase(y));
    }
    ll query(ll x) {
        assert(!empty());
        Q = 1; auto l = *lower_bound({0,0,x}); Q = 0;
        return l.k * x + l.m;
    }
}; // hash-cpp-all = 1a3c15147b3a3e2ea69bfa41ac9f0914
```

## Numerical (3)

### GoldenSectionSearch.h

**Description:** Finds the argument minimizing the function  $f$  in the interval  $[a,b]$  assuming  $f$  is unimodal on the interval, i.e. has only one local minimum. The maximum error in the result is  $\epsilon$ . Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.

**Usage:** double func(double x) { return  $4+x+.3*x*x$ ; }

double xmin = gss(-1000,1000,func);

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

14 lines

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

```
b = x2; x2 = x1; f2 = f1;
x1 = b - r*(b-a); f1 = f(x1);
} else {
    a = x1; x1 = x2; f1 = f2;
    x2 = a + r*(b-a); f2 = f(x2);
}
return a;
} // hash-cpp-all = 31d45b514727a298955001a74bb9b9fa
```

### Polynomial.h

17 lines

```
struct Poly {
    vector<double> a;
    double operator()(double x) const {
        double val = 0;
        for(int i = sz(a); i--;) (val += x) += a[i];
        return val;
    }
    void diff() {
        rep(i,1,sz(a)) a[i-1] = i*a[i];
        a.pop_back();
    }
    void divroot(double x0) {
        double b = a.back(), c; a.back() = 0;
        for(int i=sz(a)-1; i--;) c = a[i], a[i] = a[i+1]*x0+b,
            ↪ b=c;
        a.pop_back();
    }
}; // hash-cpp-all = c9b7b07a5aae7b0a6df1b8cdb046375f
```

### PolyRoots.h

**Description:** Finds the real roots to a polynomial.

**Usage:** poly\_roots({{2,-3,1}},-1e9,1e9) // solve  $x^2-3x+2 = 0$

**Time:**  $\mathcal{O}(n^2 \log(1/\epsilon))$

"Polynomial.h" 23 lines

```
vector<double> poly_roots(Poly p, double xmin, double xmax
    ↪) {
    if (sz(p.a) == 2) { return {-p.a[0]/p.a[1]}; }
    vector<double> ret;
    Poly der = p;
    der.diff();
    auto dr = poly_roots(der, xmin, xmax);
    dr.push_back(xmin-1);
    dr.push_back(xmax+1);
    sort(all(dr));
    rep(i,0,sz(dr)-1) {
        double l = dr[i], h = dr[i+1];
        bool sign = p(l) > 0;
        if (sign ^ (p(h) > 0)) {
            rep(it,0,60) { // while (h - l > 1e-8)
                double m = (l + h) / 2, f = p(m);
                if ((f <= 0) ^ sign) l = m;
                else h = m;
            }
            ret.push_back((l + h) / 2);
        }
    }
    return ret;
} // hash-cpp-all = 2cf1903cf3e930ecc5ea0059a9b7fce5
```

### PolyInterpolate.h

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

13 lines

```
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
    vd res(n), temp(n);
    rep(k,0,n-1) rep(i,k+1,n)
        y[i] = (y[i] - y[k]) / (x[i] - x[k]);
    double last = 0; temp[0] = 1;
    rep(k,0,n) rep(i,0,n) {
        res[i] += y[k] * temp[i];
        swap(last, temp[i]);
        temp[i] -= last * x[k];
    }
    return res;
} // hash-cpp-all = 08bf48c9301c849dfc6064b6450af6f3
```

### BerlekampMassey.h

**Description:** Recovers any  $n$ -order linear recurrence relation from the first  $2n$  terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size  $\leq n$ .

**Usage:** BerlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}

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

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

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

    C.resize(L + 1); C.erase(C.begin());
    trav(x, C) x = (mod - x) % mod;
    return C;
} // hash-cpp-all = 40387d9fed31766a705d6b2206790deb
```

### LinearRecurrence.h

**Description:** Generates the  $k$ 'th term of an  $n$ -order linear recurrence  $S[i] = \sum_j S[i-j-1]tr[j]$ , given  $S[0 \dots n-1]$  and  $tr[0 \dots n-1]$ . Faster than matrix multiplication. Useful together with Berlekamp-Massey.

**Usage:** linearRec({0, 1}, {1, 1}, k) //  $k$ 'th Fibonacci number

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

26 lines

```
typedef vector<ll> Poly;
ll linearRec(Poly S, Poly tr, ll k) { // hash-cpp-1
    int n = sz(S);

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

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

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

ll res = 0;
rep(i,0,n) res = (res + pol[i + 1] * S[i]) % mod;
return res;
} // hash-cpp-1 = 261dd85251df2df60ee444e087e8ffc2
```

HillClimbing.h

**Description:** Poor man’s optimization for unimodal functions. 16 lines

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

double func(P p);

pair<double, P> hillClimb(P start) {
    pair<double, P> cur(func(start), start);
    for (double jmp = 1e9; jmp > 1e-20; jmp /= 2) {
        rep(j,0,100) rep(dx,-1,2) rep(dy,-1,2) {
            P p = cur.second;
            p[0] += dx*jmp;
            p[1] += dy*jmp;
            cur = min(cur, make_pair(func(p), p));
        }
    }
    return cur;
} // hash-cpp-all = f40e55c81952cae743714899825b9fe4
```

Integrate.h

**Description:** Simple integration of a function over an interval using Simpson’s rule. The error should be proportional to  $h^4$ , although in practice you will want to verify that the result is stable to desired precision when epsilon changes. 8 lines

```
double quad(double (*f)(double), double a, double b) {
    const int n = 1000;
    double h = (b - a) / 2 / n;
    double v = f(a) + f(b);
    rep(i,1,n*2)
        v += f(a + i*h) * (i&1 ? 4 : 2);
    return v * h / 3;
} // hash-cpp-all = 65e2375b3152c23048b469eb414fe6b6
```

IntegrateAdaptive.h

**Description:** Fast integration using an adaptive Simpson’s rule.

**Usage:** double z, y;

```
double h(double x) { return x*x + y*y + z*z <= 1; }
double g(double y) { ::y = y; return quad(h, -1, 1); }
double f(double z) { ::z = z; return quad(g, -1, 1); }
double sphereVol = quad(f, -1, 1), pi = sphereVol*3/4;
```

```
typedef double d;
d simpson(d (*f)(d), d a, d b) {
    d c = (a+b) / 2;
    return (f(a) + 4*f(c) + f(b)) * (b-a) / 6;
}

d rec(d (*f)(d), d a, d b, d eps, d S) {
    d c = (a+b) / 2;
    d S1 = simpson(f, a, c);
    d 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)
    ↪;
}
d quad(d (*f)(d), d a, d b, d eps = 1e-8) {
    return rec(f, a, b, eps, simpson(f, a, b));
} // hash-cpp-all = ad8a754372ce74e5a3d07ce46c2fe0ca
```

Determinant.h

**Description:** Calculates determinant of a matrix. Destroys the matrix.

**Time:**  $\mathcal{O}(N^3)$  15 lines

```
double det(vector<vector<double>>& a) {
    int n = sz(a); double res = 1;
    rep(i,0,n) {
        int b = i;
        rep(j,i+1,n) if (fabs(a[j][i]) > fabs(a[b][i])) b = j;
        if (i != b) swap(a[i], a[b]), res *= -1;
        res *= a[i][i];
        if (res == 0) return 0;
        rep(j,i+1,n) {
            double v = a[j][i] / a[i][i];
            if (v != 0) rep(k,i+1,n) a[j][k] -= v * a[i][k];
        }
    }
    return res;
} // hash-cpp-all = bd5cec161e6ad4c483e662c34eae2d08
```

IntDeterminant.h

**Description:** Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.

**Time:**  $\mathcal{O}(N^3)$  18 lines

```
const ll mod = 12345;
ll det(vector<vector<ll>>& a) {
    int n = sz(a); ll ans = 1;
    rep(i,0,n) {
        rep(j,i+1,n) {
            while (a[j][i] != 0) { // gcd step
                ll t = a[i][i] / a[j][i];
                if (t) rep(k,i,n)
                    a[i][k] = (a[i][k] - a[j][k] * t) % mod;
                swap(a[i], a[j]);
                ans *= -1;
            }
            ans = ans * a[i][i] % mod;
            if (!ans) return 0;
        }
    }
    return (ans + mod) % mod;
} // hash-cpp-all = 3313dc3b38059fdf9f41220b469cfd13
```

Simplex.h

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

**Usage:** vvd A = {{1,-1}, {-1,1}, {-1,-2}};

vd b = {1,1,-4}, c = {-1,-1}, x;

T val = LPSolver(A, b, c).solve(x);

**Time:**  $\mathcal{O}(NM * \#pivots)$ , where a pivot may be e.g. an edge relaxation.  $\mathcal{O}(2^n)$  in the general case. 68 lines

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

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

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

    LPSolver(const vvd& A, const vd& b, const vd& c) :
        m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)) { //
            ↪ hash-cpp-1
                rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
                rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b
                    ↪ [i]; }
                rep(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
                N[n] = -1; D[m+1][n] = 1;
            } // hash-cpp-1 = 6ff8e92a6bb47fbd6606c75a07178914

    void pivot(int r, int s) { // hash-cpp-2
        T *a = D[r].data(), inv = 1 / a[s];
        rep(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
            T *b = D[i].data(), inv2 = b[s] * inv;
            rep(j,0,n+2) b[j] -= a[j] * inv2;
            b[s] = a[s] * inv2;
        }
        rep(j,0,n+2) if (j != s) D[r][j] *= inv;
        rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
    } // hash-cpp-2 = 9cd0a84b89fb678b2888e0defa688de2

    bool simplex(int phase) { // hash-cpp-3
        int x = m + phase - 1;
        for (;;) {
            int s = -1;
            rep(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
            if (D[x][s] >= -eps) return true;
            int r = -1;
            rep(i,0,m) {
                if (D[i][s] <= eps) continue;
                if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                    < MP(D[r][n+1] / D[r][s], B[r])) r =
                    ↪ i;
            }
            if (r == -1) return false;
            pivot(r, s);
        }
    } // hash-cpp-3 = f156440bce4f5370ea43b0efa7de25ed

    T solve(vd &x) { // hash-cpp-4
        int r = 0;
        rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
        if (D[r][n+1] < -eps) {
            pivot(r, n);
            if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
            rep(i,0,m) if (B[i] == -1) {
                int s = 0;
                rep(j,1,n+1) ltj(D[i]);
                pivot(i, s);
            }
        }
    }
}
```

```

    bool ok = simplex(1); x = vd(n);
    rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
} // hash-cpp-4 = 396a95621f5e196bb87eb95518560dfb
};

```

### math-simplex.cpp

**Description:** Simplex algorithm. WARNING- segfaults on empty (size 0) max cx st Ax<=b, x>=0 do 2 phases; 1st check feasibility; 2nd check boundedness and ans

```

vector<double> simplex(vector<vector<double>> > A, vector<
    double> b, vector<double> c) {
    int n = (int) A.size(), m = (int) A[0].size()+1, r = n,
        s = m-1;
    vector<vector<double>> > D = vector<vector<double>> > (n
        +2, vector<double>(m+1));
    vector<int> ix = vector<int>(n+m);
    for (int i=0; i<n+m; i++) ix[i] = i;
    for (int i=0; i<n; i++) {
        for (int j=0; j<m-1; j++) D[i][j] = -A[i][j];
        D[i][m-1] = 1;
        D[i][m] = b[i];
        if (D[r][m] > D[i][m]) r = i;
    }
    for (int j=0; j<m-1; j++) D[n][j] = c[j];
    D[n+1][m-1] = -1; int z = 0;
    for (double d;;) {
        if (r < n) {
            swap(ix[s], ix[r+m]);
            D[r][s] = 1.0/D[r][s];
            for (int j=0; j<=m; j++) if (j!=s) D[r][j] *= -D[r][
                s];
            for (int i=0; i<=n+1; i++) if (i!=r) {
                for (int j=0; j<=m; j++) if (j!=s) D[i][j] += D[r][
                    s] * D[i][s];
                D[i][s] *= D[r][s];
            }
        }
        r = -1; s = -1;
        for (int j=0; j < m; j++) if (s<0 || ix[s]>ix[j]) {
            if (D[n+1][j]>eps || D[n+1][j]>-eps && D[n][j]>eps)
                s = j;
        }
        if (s < 0) break;
        for (int i=0; i<n; i++) if (D[i][s]<-eps) {
            if (r < 0 || (d = D[r][m]/D[r][s]-D[i][m]/D[i][s]) <
                -eps || d < eps && ix[r+m] > ix[i+m]) r=i;
        }
        if (r < 0) return vector<double>(); // unbounded
    }
    if (D[n+1][m] < -eps) return vector<double>(); //
        infeasible
    vector<double> x(m-1);
    for (int i = m; i < n+m; i++) if (ix[i] < m-1) x[ix[i]]
        = D[i-m][m];
    printf("%.21f\n", D[n][m]);
    return x; // ans: D[n][m]
} // hash-cpp-all = 70201709abdf05eff90d9393c756b95

```

### SolveLinear.h

**Description:** Solves  $A * x = b$ . If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in  $A$  and  $b$  is lost.

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

38 lines

```

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

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

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

    x.assign(m, 0);
    for (int i = rank; i--;) {
        b[i] /= A[i][i];
        x[col[i]] = b[i];
        rep(j,0,i) b[j] -= A[j][i] * b[i];
    }
    return rank; // (multiple solutions if rank < m)
} // hash-cpp-all = 44c9ab90319b30df6719c5b5394bc618

```

### SolveLinear2.h

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

```

" SolveLinear.h"
8 lines

rep(j,0,n) if (j != i) // instead of rep(j,i+1,n)
    // ... then at the end:
    x.assign(m, undefined);
rep(i,0,rank) {
    rep(j,rank,m) if (fabs(A[i][j]) > eps) goto fail;
    x[col[i]] = b[i] / A[i][i];
fail:; }
// hash-cpp-all = 08e495d9d51e80a183ccd030e3bf6700

```

### SolveLinearBinary.h

**Description:** Solves  $Ax = b$  over  $\mathbb{F}_2$ . If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys  $A$  and  $b$ .

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

34 lines

```

typedef bitset<1000> bs;

int solveLinear(vector<bs>& A, vi& b, bs& x, int m) {
    int n = sz(A), rank = 0, br;
    assert(m <= sz(x));
    vi col(m); iota(all(col), 0);
    rep(i,0,n) {
        for (br=i; br<n; ++br) if (A[br].any()) break;
        if (br == n) {

```

```

            rep(j,i,n) if (b[j]) return -1;
            break;
        }
        int bc = (int)A[br]._Find_next(i-1);
        swap(A[i], A[br]);
        swap(b[i], b[br]);
        swap(col[i], col[bc]);
        rep(j,0,n) if (A[j][i] != A[j][bc]) {
            A[j].flip(i); A[j].flip(bc);
        }
        rep(j,i+1,n) if (A[j][i]) {
            b[j] ^= b[i];
            A[j] ^= A[i];
        }
        rank++;
    }

    x = bs();
    for (int i = rank; i--;) {
        if (!b[i]) continue;
        x[col[i]] = 1;
        rep(j,0,i) b[j] ^= A[j][i];
    }
    return rank; // (multiple solutions if rank < m)
} // hash-cpp-all = fa2d7a3e3a84d8fb47610cc474e77b4e

```

### MatrixInverse.h

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

**Time:**  $\mathcal{O}(n^3)$

35 lines

```

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

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

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

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

```

```
} // hash-cpp-all = ebfff64122d6372fde3a086c95e2cfc7
```

## Tridiagonal.h

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

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

This is useful for solving problems on the type

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

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

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

Fails if the solution is not unique.

If  $|d_i| > |p_i| + |q_{i-1}|$  for all  $i$ , or  $|d_i| > |p_{i-1}| + |q_i|$ , or the matrix is positive definite, the algorithm is numerically stable and neither  $\text{tr}$  nor the check for  $\text{diag}[i] == 0$  is needed.

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

```
26 lines
typedef double T;
vector<T> tridiagonal(vector<T> diag, const vector<T>&
    ↳super,
    const vector<T>& sub, vector<T> b) {
    int n = sz(b); vi tr(n);
    rep(i,0,n-1) {
        if (abs(diag[i]) < 1e-9 * abs(super[i])) { // diag[i]
            ↳== 0
            b[i+1] -= b[i] * diag[i+1] / super[i];
            if (i+2 < n) b[i+2] -= b[i] * sub[i+1] / super[i];
            diag[i+1] = sub[i]; tr[++i] = 1;
        } else {
            diag[i+1] -= super[i]*sub[i]/diag[i];
            b[i+1] -= b[i]*sub[i]/diag[i];
        }
    }
    for (int i = n; i--;) {
        if (tr[i]) {
            swap(b[i], b[i-1]);
            diag[i-1] = diag[i];
            b[i] /= super[i-1];
        } else {
            b[i] /= diag[i];
            if (i) b[i-1] -= b[i]*super[i-1];
        }
    }
    return b;
} // hash-cpp-all = 8f9fa8b1e5e82731da914aed0632312f
```

## 3.1 Fourier transforms

### fft.cpp

**Description:** FFT/NTT, polynomial mod/log/exp

```
299 lines
namespace fft {
    #if FFT
    // FFT
    using dbl = double;
```

```
struct num {
    dbl x, y;
    num(dbl x_ = 0, dbl y_ = 0) : x(x_), y(y_) { }
};
inline num operator+(num a, num b) { return num(a.x + b.x,
    ↳ a.y + b.y); }
inline num operator-(num a, num b) { return num(a.x - b.x,
    ↳ a.y - b.y); }
inline num operator*(num a, num b) { return num(a.x * b.x
    ↳ - a.y * b.y, a.x * b.y + a.y * b.x); }
inline num conj(num a) { return num(a.x, -a.y); }
inline num inv(num a) { dbl n = (a.x*a.x+a.y*a.y); return
    ↳ num(a.x/n, -a.y/n); }
#else
// NTT
const int mod = 998244353, g = 3;
// For p < 2^30 there is also (5 << 25, 3), (7 << 26, 3),
// (479 << 21, 3) and (483 << 21, 5). Last two are > 10^9.
struct num {
    int v;
    num(ll v_ = 0) : v(int(v_ % mod)) { if (v<0) v+=mod; }
    explicit operator int() const { return v; }
};
inline num operator+(num a, num b) { return num(a.v+b.v); }
inline num operator-(num a, num b) { return num(a.v+mod-b.v)
    ↳; }
inline num operator*(num a, num b) { return num(1ll*a.v*b.v)
    ↳; }
inline num pow(num a, int b) {
    num r = 1;
    do {if (b&1) r=r*a; a=a*a;} while (b>>=1);
    return r;
}
inline num inv(num a) { return pow(a, mod-2); }
#endif

using vn = vector<num>;
vi rev({0, 1});
vn rt(2, num(1)), fa, fb;

inline void init(int n) {
    if (n <= sz(rt)) return;
    rev.resize(n);
    rep(i,0,n) rev[i] = (rev[i]>>1] | ((i&1)*n)) >> 1;
    rt.reserve(n);
    for (int k = sz(rt); k < n; k *= 2) {
        rt.resize(2*k);
    }
    #if FFT
        double a=M_PI/k; num z(cos(a),sin(a)); // FFT
    #else
        num z = pow(num(g), (mod-1)/(2*k)); // NTT
    #endif
    rep(i,k/2,k) rt[2*i] = rt[i], rt[2*i+1] = rt[i]*z;
}

inline void fft(vector<num> &a, int n) { // hash-cpp-1
    init(n);
    int s = __builtin_ctz(sz(rev)/n);
    rep(i,0,n) if (i < rev[i]>>s) swap(a[i], a[rev[i]>>s]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
            num t = rt[j+k] * a[i+j+k];
            a[i+j+k] = a[i+j] - t;
            a[i+j] = a[i+j] + t;
        }
} // hash-cpp-1 = 1f0820b04997ddca9b78742df352d419
```

```
// Complex/NTT
vn multiply(vn a, vn b) {
    int s = sz(a) + sz(b) - 1;
    if (s <= 0) return {};
    int L = s > 1 ? 32 - __builtin_clz(s-1) : 0, n = 1 << L;
    a.resize(n), b.resize(n);
    fft(a, n);
    fft(b, n);
    num d = inv(num(n));
    rep(i,0,n) a[i] = a[i] * b[i] * d;
    reverse(a.begin()+1, a.end());
    fft(a, n);
    a.resize(s);
    return a;
}

// Complex/NTT power-series inverse
// Doubles b as b[:n] = (2 - a[:n] * b[:n/2]) * b[:n/2]
vn inverse(const vn& a) {
    if (a.empty()) return {};
    vn b({inv(a[0])});
    b.reserve(2*a.size());
    while (sz(b) < sz(a)) {
        int n = 2*sz(b);
        b.resize(2*n, 0);
        if (sz(fa) < 2*n) fa.resize(2*n);
        fill(fa.begin(), fa.begin()+2*n, 0);
        copy(a.begin(), a.begin()+min(n,sz(a)), fa.begin());
        fft(b, 2*n);
        fft(fa, 2*n);
        num d = inv(num(2*n));
        rep(i, 0, 2*n) b[i] = b[i] * (2 - fa[i] * b[i]) * d;
        reverse(b.begin()+1, b.end());
        fft(b, 2*n);
        b.resize(n);
    }
    b.resize(a.size());
    return b;
}

#if FFT
// Double multiply (num = complex)
using vd = vector<double>;
vd multiply(const vd& a, const vd& b) {
    int s = sz(a) + sz(b) - 1;
    if (s <= 0) return {};
    int L = s > 1 ? 32 - __builtin_clz(s-1) : 0, n = 1 << L;
    if (sz(fa) < n) fa.resize(n);
    if (sz(fb) < n) fb.resize(n);

    fill(fa.begin(), fa.begin() + n, 0);
    rep(i,0,sz(a)) fa[i].x = a[i];
    rep(i,0,sz(b)) fa[i].y = b[i];
    fft(fa, n);
    trav(x, fa) x = x * x;
    rep(i,0,n) fb[i] = fa[(n-i)&(n-1)] - conj(fa[i]);
    fft(fb, n);
    vd r(s);
    rep(i,0,s) r[i] = fb[i].y / (4*n);
    return r;
}

// Integer multiply mod m (num = complex) // hash-cpp-2
vi multiply_mod(const vi& a, const vi& b, int m) {
    int s = sz(a) + sz(b) - 1;
    if (s <= 0) return {};
    int L = s > 1 ? 32 - __builtin_clz(s-1) : 0, n = 1 << L;
    if (sz(fa) < n) fa.resize(n);
```



```

    if (sz(fb) < n) fb.resize(n);

    rep(i,0,sz(a)) fa[i] = num(a[i] & ((1<<15)-1), a[i] >>
        ↪15);
    fill(fa.begin()+sz(a), fa.begin() + n, 0);
    rep(i,0,sz(b)) fb[i] = num(b[i] & ((1<<15)-1), b[i] >>
        ↪15);
    fill(fb.begin()+sz(b), fb.begin() + n, 0);

    fft(fa, n);
    fft(fb, n);
    double r0 = 0.5 / n; // 1/2n
    rep(i,0,n/2+1) {
        int j = (n-i)&(n-1);
        num g0 = (fb[i] + conj(fb[j])) * r0;
        num g1 = (fb[i] - conj(fb[j])) * r0;
        swap(g1.x, g1.y); g1.y *= -1;
        if (j != i) {
            swap(fa[j], fa[i]);
            fb[j] = fa[j] * g1;
            fa[j] = fa[j] * g0;
        }
        fb[i] = fa[i] * conj(g1);
        fa[i] = fa[i] * conj(g0);
    }
    fft(fa, n);
    fft(fb, n);
    vi r(s);
    rep(i,0,s) r[i] = int((1l(fa[i].x+0.5)
        + 1l(fa[i].y+0.5) % m << 15)
        + 1l(fb[i].x+0.5) % m << 15)
        + 1l(fb[i].y+0.5) % m << 30)) % m);
    return r;
} // hash-cpp-2 = e8c5f6755ad1e5a976d6c6ffd37b3b22
#endif

} // namespace fft

// For multiply_mod, use num = modnum, poly = vector<num>
using fft::num;
using poly = fft::vn;
using fft::multiply;
using fft::inverse;
poly& operator+=(poly& a, const poly& b) {
    if (sz(a) < sz(b)) a.resize(b.size());
    rep(i,0,sz(b)) a[i]=a[i]+b[i];
    return a;
}
poly operator+(const poly& a, const poly& b) { poly r=a; r
    ↪+=b; return r; }
poly& operator-=(poly& a, const poly& b) {
    if (sz(a) < sz(b)) a.resize(b.size());
    rep(i,0,sz(b)) a[i]=a[i]-b[i];
    return a;
}
poly operator-(const poly& a, const poly& b) { poly r=a; r
    ↪-=b; return r; }
poly& operator*=(poly& a, const num& b) { // Optional
    trav(x, a) x = x * b;
    return a;
}
poly operator*(const poly& a, const num& b) { poly r=a; r
    ↪*=b; return r; }
poly operator*(const poly& a, const poly& b) {
    // TODO: small-case?
    return multiply(a, b);
}
poly& operator*=(poly& a, const poly& b) {return a = a*b;}

```

```

// Polynomial floor division; no leading 0's pls
poly operator/(poly a, poly b) { // hash-cpp-3
    if (sz(a) < sz(b)) return {};
    int s = sz(a)-sz(b)+1;
    reverse(a.begin(), a.end());
    reverse(b.begin(), b.end());
    a.resize(s);
    b.resize(s);
    a = a * inverse(move(b));
    a.resize(s);
    reverse(a.begin(), a.end());
    return a;
} // hash-cpp-3 = a6589ce8fcf1e33df3b42ee703a7fe60
poly& operator/=(poly& a, const poly& b) {return a = a/b;}
poly& operator%=(poly& a, const poly& b) { // hash-cpp-4
    if (sz(a) >= sz(b)) {
        poly c = (a / b) * b;
        a.resize(sz(b)-1);
        rep(i,0,sz(a)) a[i] = a[i]-c[i];
    }
    return a;
} // hash-cpp-4 = 9af255f48abbeafd8acde353357b84fd
poly operator%(const poly& a, const poly& b) { poly r=a; r
    ↪%=b; return r; }

// Log/exp/pow
poly deriv(const poly& a) { // hash-cpp-5
    if (a.empty()) return {};
    poly b(sz(a)-1);
    rep(i,1,sz(a)) b[i-1]=a[i]*i;
    return b;
} // hash-cpp-5 = 94aa209b3e956051e6b3131bf1faafd1
poly integ(const poly& a) { // hash-cpp-6
    poly b(sz(a)+1);
    b[1]=1; // mod p
    rep(i,2,sz(b)) b[i]=b[fft::mod%i]*(-fft::mod/i); // mod
        ↪p
    rep(i,1,sz(b)) b[i]=a[i-1]*b[i]; // mod p
    //rep(i,1,sz(b)) b[i]=a[i-1]*inv(num(i)); // else
    return b;
} // hash-cpp-6 = 6f13f6a43b2716a116d347000820f0bd
poly log(const poly& a) { // a[0] == 1 // hash-cpp-7
    poly b = integ(deriv(a)*inverse(a));
    b.resize(a.size());
    return b;
} // hash-cpp-7 = ce1533264298c5382f72a2a1b0947045
poly exp(const poly& a) { // a[0] == 0 // hash-cpp-8
    poly b(1,num(1));
    if (a.empty()) return b;
    while (sz(b) < sz(a)) {
        int n = min(sz(b) * 2, sz(a));
        b.resize(n);
        poly v = poly(a.begin(), a.begin() + n) - log(b);
        v[0] = v[0]+num(1);
        b *= v;
        b.resize(n);
    }
    return b;
} // hash-cpp-8 = f645d091e4ae3ee3dc2aa095d4aa699a
poly pow(const poly& a, int m) { // m >= 0 // hash-cpp-9
    poly b(a.size());
    if (!m) { b[0] = 1; return b; }
    int p = 0;
    while (p<sz(a) && a[p].v==0) ++p;
    if (1ll*m*p >= sz(a)) return b;
    num mu = pow(a[p], m), di = inv(a[p]);
    poly c(sz(a) - m*p);

```

```

    rep(i,0,sz(c)) c[i] = a[i+p] * di;
    c = log(c);
    trav(v,c) v = v * m;
    c = exp(c);
    rep(i,0,sz(c)) b[i+m*p] = c[i] * mu;
    return b;
} // hash-cpp-9 = 0f4830b9de34c26d39f170069827121f

// Multipoint evaluation/interpolation
// hash-cpp-10
vector<num> eval(const poly& a, const vector<num>& x) {
    int n=sz(x);
    if (!n) return {};
    vector<poly> up(2*n);
    rep(i,0,n) up[i+n] = poly({0-x[i], 1});
    per(i,1,n) up[i] = up[2*i]*up[2*i+1];
    vector<poly> down(2*n);
    down[1] = a % up[1];
    rep(i,2,2*n) down[i] = down[i/2] % up[i];
    vector<num> y(n);
    rep(i,0,n) y[i] = down[i+n][0];
    return y;
} // hash-cpp-10 = a079eba46c3110851ec6b0490b439931
// hash-cpp-11
poly interp(const vector<num>& x, const vector<num>& y) {
    int n=sz(x);
    assert(n);
    vector<poly> up(n*2);
    rep(i,0,n) up[i+n] = poly({0-x[i], 1});
    per(i,1,n) up[i] = up[2*i]*up[2*i+1];
    vector<num> a = eval(deriv(up[1]), x);
    vector<poly> down(2*n);
    rep(i,0,n) down[i+n] = poly({y[i]*inv(a[i])});
    per(i,1,n) down[i] = down[i*2] * up[i*2+1] + down[i*2+1]
        ↪ * up[i*2];
    return down[1];
} // hash-cpp-11 = 74f15e1e82d51e852b321a1ff75ba1fd

```

## FastSubsetTransform.h

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

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

16 lines

```

void FST(vi& a, bool inv) {
    for (int n = sz(a), step = 1; step < n; step *= 2) {
        for (int i = 0; i < n; i += 2 * step) rep(j,i,step)
            ↪{
                int &u = a[j], &v = a[j + step]; tie(u, v) =
                    inv ? pii(v - u, u) : pii(v, u + v); // AND
                    inv ? pii(v, u - v) : pii(u + v, u); // OR
                    pii(u + v, u - v); // XOR
            }
        if (inv) trav(x, a) x /= sz(a); // XOR only
    }
    vi conv(vi a, vi b) {
        FST(a, 0); FST(b, 0);
        rep(i,0,sz(a)) a[i] *= b[i];
        FST(a, 1); return a;
    } // hash-cpp-all = 3de473e2c1de97e6e9ff0f13542cf3fb

```

## Number theory (4)

## 4.1 Modular arithmetic

### ModularArithmetic.h

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

```
"euclid.h" 18 lines
const ll mod = 17; // change to something else
struct Mod {
    ll x;
    Mod(ll xx) : x(xx) {}
    Mod operator+(Mod b) { return Mod((x + b.x) % mod); }
    Mod operator-(Mod b) { return Mod((x - b.x + mod) % mod)
    ↪; }
    Mod operator*(Mod b) { return Mod((x * b.x) % mod); }
    Mod operator/(Mod b) { return *this * invert(b); }
    Mod invert(Mod a) {
        ll x, y, g = euclid(a.x, mod, x, y);
        assert(g == 1); return Mod((x + mod) % mod);
    }
    Mod operator^(ll e) {
        if (!e) return Mod(1);
        Mod r = *this ^ (e / 2); r = r * r;
        return e&1 ? *this * r : r;
    }
}; // hash-cpp-all = 35bfea8c111cb24c4ce84c658446961b
```

### ModInverse.h

**Description:** Pre-computation of modular inverses. Assumes LIM ≤ mod and that mod is a prime.

```
4 lines
const ll mod = 1000000007, LIM = 200000;
ll* inv = new ll[LIM] - 1; inv[1] = 1;
rep(i,2,LIM) inv[i] = mod - (mod / i) * inv[mod % i] % mod
    ↪;
// hash-cpp-all = 6f684f0b9ae6c69f42de68f023a81de5
```

### ModPow.h

```
6 lines
const ll mod = 1000000007; // faster if const
ll modpow(ll a, ll e) {
    if (e == 0) return 1;
    ll x = modpow(a * a % mod, e >> 1);
    return e & 1 ? x * a % mod : x;
} // hash-cpp-all = 2fa6d9ccac4586cba0618aad18cdc9de
```

### ModSum.h

**Description:** Sums of mod'ed arithmetic progressions.

$\text{modsum}(to, c, k, m) = \sum_{i=0}^{t-1} (ki + c) \% m$ .  $\text{divsum}$  is similar but for floored division.

**Time:**  $\log(m)$ , with a large constant.

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

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

ll modsum(ull to, ll c, ll k, ll m) {
    c = ((c % m) + m) % m;
```

```
k = ((k % m) + m) % m;
return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
} // hash-cpp-all = 8d6e082e0ea6be867eaea12670d08dcc
```

### ModMulLL.h

**Description:** Calculate  $a \cdot b \bmod c$  (or  $a^b \bmod c$ ) for large  $c$ .

**Time:**  $\mathcal{O}(64/\text{bits} \cdot \log b)$ , where  $\text{bits} = 64 - k$ , if we want to deal with  $k$ -bit numbers.

```
19 lines
typedef unsigned long long ull;
const int bits = 10;
// if all numbers are less than 2^k, set bits = 64-k
const ull po = 1 << bits;
ull mod_mul(ull a, ull b, ull &c) {
    ull x = a * (b & (po - 1)) % c;
    while ((b >= bits) > 0) {
        a = (a << bits) % c;
        x += (a * (b & (po - 1))) % c;
    }
    return x % c;
}
ull mod_pow(ull a, ull b, ull mod) {
    if (b == 0) return 1;
    ull res = mod_pow(a, b / 2, mod);
    res = mod_mul(res, res, mod);
    if (b & 1) return mod_mul(res, a, mod);
    return res;
} // hash-cpp-all = 40cd743544228d297c803154525107ab
```

### ModSqrt.h

**Description:** Tonelli-Shanks algorithm for modular square roots.

**Time:**  $\mathcal{O}(\log^2 p)$  worst case, often  $\mathcal{O}(\log p)$

```
30 lines
"ModPow.h"
ll sqrt(ll a, ll p) {
    a %= p; if (a < 0) a += p;
    if (a == 0) return 0;
    assert(modpow(a, (p-1)/2, p) == 1);
    if (p % 4 == 3) return modpow(a, (p+1)/4, p);
    // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 ==
    ↪ 5
    ll s = p - 1;
    int r = 0;
    while (s % 2 == 0)
        ++r, s /= 2;
    ll n = 2; // find a non-square mod p
    while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
    ll x = modpow(a, (s + 1) / 2, p);
    ll b = modpow(a, s, p);
    ll g = modpow(n, s, p);
    for (;;) {
        ll t = b;
        int m = 0;
        for (; m < r; ++m) {
            if (t == 1) break;
            t = t * t % p;
        }
        if (m == 0) return x;
        ll gs = modpow(g, 1 << (r - m - 1), p);
        g = gs * gs % p;
        x = x * gs % p;
        b = b * g % p;
        r = m;
    }
} // hash-cpp-all = 83e24bd39c8c93946ad3021b8ca6c3c4
```

## 4.2 Primality

### eratosthenes.h

**Description:** Prime sieve for generating all primes up to a certain limit.  $\text{isprime}[i]$  is true iff  $i$  is a prime.

**Time:**  $\text{lim} = 100'000'000 \approx 0.8$  s. Runs 30% faster if only odd indices are stored.

```
11 lines
const int MAX_PR = 5000000;
bitset<MAX_PR> isprime;
vi eratosthenes_sieve(int lim) {
    isprime.set(); isprime[0] = isprime[1] = 0;
    for (int i = 4; i < lim; i += 2) isprime[i] = 0;
    for (int i = 3; i*i < lim; i += 2) if (isprime[i])
        for (int j = i*i; j < lim; j += i*2) isprime[j] = 0;
    vi pr;
    rep(i,2,lim) if (isprime[i]) pr.push_back(i);
    return pr;
} // hash-cpp-all = 0564a3337fb69c0b87dfd3c56cdf2e3
```

### MillerRabin.h

**Description:** Miller-Rabin primality probabilistic test. Probability of failing one iteration is at most  $1/4$ . 15 iterations should be enough for 50-bit numbers.

**Time:** 15 times the complexity of  $a^b \bmod c$ .

```
16 lines
"ModMulLL.h"
bool prime(ull p) {
    if (p == 2) return true;
    if (p == 1 || p % 2 == 0) return false;
    ull s = p - 1;
    while (s % 2 == 0) s /= 2;
    rep(i,0,15) {
        ull a = rand() % (p - 1) + 1, tmp = s;
        ull mod = mod_pow(a, tmp, p);
        while (tmp != p - 1 && mod != 1 && mod != p - 1) {
            mod = mod_mul(mod, mod, p);
            tmp *= 2;
        }
        if (mod != p - 1 && tmp % 2 == 0) return false;
    }
    return true;
} // hash-cpp-all = ccddf18bab60a654ff4af45e95dd60b6
```

### factor.h

**Description:** Pollard's rho algorithm. It is a probabilistic factorisation algorithm, whose expected time complexity is good. Before you start using it, run  $\text{init}(\text{bits})$ , where  $\text{bits}$  is the length of the numbers you use. Returns factors of the input without duplicates.

**Time:** Expected running time should be good enough for 50-bit numbers.

```
35 lines
"ModMulLL.h", "MillerRabin.h", "eratosthenes.h"
vector<ull> pr;
ull f(ull a, ull n, ull &has) {
    return (mod_mul(a, a, n) + has) % n;
}
vector<ull> factor(ull d) {
    vector<ull> res;
    for (int i = 0; i < sz(pr) && pr[i]*pr[i] <= d; i++)
        if (d % pr[i] == 0) {
            while (d % pr[i] == 0) d /= pr[i];
            res.push_back(pr[i]);
        }
    //d is now a product of at most 2 primes.
    if (d > 1) {
        if (prime(d))
            res.push_back(d);
        else while (true) {
```



```
ull has = rand() % 2321 + 47;
ull x = 2, y = 2, c = 1;
for (; c=1; c = __gcd((y > x ? y - x : x - y), d))
    ↪{
    x = f(x, d, has);
    y = f(y, d, has), d, has);
}
if (c != d) {
    res.push_back(c); d /= c;
    if (d != c) res.push_back(d);
    break;
}
}
}
return res;
}
void init(int bits) { //how many bits do we use?
    vi p = eratosthenes_sieve(1 << ((bits + 2) / 3));
    pr.assign(all(p));
} // hash-cpp-all = 67b304bd690b2a8445a7b4dbf93996d7
```

4.3 Divisibility

euclid.h  
**Description:** Finds the Greatest Common Divisor to the integers  $a$  and  $b$ . Euclid also finds two integers  $x$  and  $y$ , such that  $ax+by=\gcd(a,b)$ . If  $a$  and  $b$  are coprime, then  $x$  is the inverse of  $a \pmod b$ . 7 lines

```
ll gcd(ll a, ll b) { return __gcd(a, b); }

ll euclid(ll a, ll b, ll &x, ll &y) {
    if (b) { ll d = euclid(b, a % b, y, x);
        return y -= a/b * x, d; }
    return x = 1, y = 0, a;
} // hash-cpp-all = 63e6f8d2f560b27cb800273d63d2102c
```

Euclid.java  
**Description:** Finds  $\{x, y, d\}$  s.t.  $ax + by = d = \gcd(a, b)$ . 11 lines

```
static BigInteger[] euclid(BigInteger a, BigInteger b) {
    BigInteger x = BigInteger.ONE, yy = x;
    BigInteger y = BigInteger.ZERO, xx = y;
    while (b.signum() != 0) {
        BigInteger q = a.divide(b), t = b;
        b = a.mod(b); a = t;
        t = xx; xx = x.subtract(q.multiply(xx)); x = t;
        t = yy; yy = y.subtract(q.multiply(yy)); y = t;
    }
    return new BigInteger[]{x, y, a};
}
```

4.4 Fractions

ContinuedFractions.h  
**Description:** Given  $N$  and a real number  $x \geq 0$ , finds the closest rational approximation  $p/q$  with  $p, q \leq N$ . It will obey  $|p/q - x| \leq 1/qN$ . For consecutive convergents,  $p_{k+1}q_k - q_{k+1}p_k = (-1)^k$ . ( $p_k/q_k$  alternates between  $> x$  and  $< x$ .) If  $x$  is rational,  $y$  eventually becomes  $\infty$ ; if  $x$  is the root of a degree 2 polynomial the  $a$ 's eventually become cyclic. Time:  $\mathcal{O}(\log N)$  21 lines

```
typedef double d; // for N ~ 1e7; long double for N ~ 1e9
pair<ll, ll> approximate(d x, ll N) {
    ll LP = 0, LQ = 1, P = 1, Q = 0, inf = LLONG_MAX; d y =
    ↪x;
    for (;;) {
```

```
ll lim = min(P ? (N-LP) / P : inf, Q ? (N-LQ) / Q :
    ↪inf),
    a = (ll)floor(y), b = min(a, lim),
    NP = b*P + LP, NQ = b*Q + LQ;
if (a > b) {
    // If b > a/2, we have a semi-convergent that gives
    ↪us a
    // better approximation; if b = a/2, we *may* have
    ↪one.
    // Return {P, Q} here for a more canonical
    ↪approximation.
    return (abs(x - (d)NP / (d)NQ) < abs(x - (d)P / (d)Q
    ↪)) ?
        make_pair(NP, NQ) : make_pair(P, Q);
}
if (abs(y = 1/(y - (d)a)) > 3*N) {
    return {NP, NQ};
}
LP = P; P = NP;
LQ = Q; Q = NQ;
}
} // hash-cpp-all = dd6c5e1084a26365dc6321bd935975d9
```

FracBinarySearch.h  
**Description:** Given  $f$  and  $N$ , finds the smallest fraction  $p/q \in [0, 1]$  such that  $f(p/q)$  is true, and  $p, q \leq N$ . You may want to throw an exception from  $f$  if it finds an exact solution, in which case  $N$  can be removed. Usage: fracBS({}(Frac f) { return f.p>=3\*f.q; }, 10); // {1,3} Time:  $\mathcal{O}(\log(N))$  24 lines

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

template<class F>
Frac fracBS(F f, ll N) {
    bool dir = 1, A = 1, B = 1;
    Frac lo{0, 1}, hi{1, 1}; // Set hi to 1/0 to search (0,
    ↪N]
    assert(!f(lo)); assert(f(hi));
    while (A || B) {
        ll adv = 0, step = 1; // move hi if dir, else lo
        for (int si = 0; step; (step *= 2) >= si) {
            adv += step;
            Frac mid{lo.p * adv + hi.p, lo.q * adv + hi.q};
            if (abs(mid.p) > N || mid.q > N || dir == !f(mid)) {
                adv -= step; si = 2;
            }
        }
        hi.p += lo.p * adv;
        hi.q += lo.q * adv;
        dir = !dir;
        swap(lo, hi);
        A = B; B = !adv;
    }
    return dir ? hi : lo;
} // hash-cpp-all = 214844f17d0c347ff436141729e0c829
```

4.5 Chinese remainder theorem

chinese.h  
**Description:** Chinese Remainder Theorem. chinese(a, m, b, n) returns a number  $x$ , such that  $x \equiv a \pmod m$  and  $x \equiv b \pmod n$ . For not coprime  $n, m$ , use chinese\_common. Note that all numbers must be less than  $2^{31}$  if you have  $Z = \text{unsigned long long}$ .

Time:  $\log(m+n)$   
"euclid.h" 13 lines

```
template<class Z> Z chinese(Z a, Z m, Z b, Z n) {
    Z x, y; euclid(m, n, x, y);
    Z ret = a * (y + m) % m * n + b * (x + n) % n * m;
    if (ret >= m * n) ret -= m * n;
    return ret;
}

template<class Z> Z chinese_common(Z a, Z m, Z b, Z n) {
    Z d = gcd(m, n);
    if ((b -= a) % n < 0) b += n;
    if (b % d) return -1; // No solution
    return d * chinese(Z(0), m/d, b/d, n/d) + a;
} // hash-cpp-all = da3099704e14964aa045c152bb478c14
```

4.6 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.

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

4.8 Estimates

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

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

Combinatorial (5)

5.1 Permutations

5.1.1 Factorial

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

IntPerm.h  
**Description:** Permutation -> integer conversion. (Not order preserving.)  
**Time:**  $\mathcal{O}(n)$

```
6 lines
int permToInt(vi& v) {
    int use = 0, i = 0, r = 0;
    trav(x,v) r=r * ++i + __builtin_popcount(use & -(1 << x))
    ↪,
    use |= 1 << x; // (note: minus, not ~!)
    return r;
} // hash-cpp-all = e1b8eaea02324af14a3da94f409019b8
```

5.1.2 Cycles

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

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

5.1.3 Derangements

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

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

5.1.4 Burnside’s lemma

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

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

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

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

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

5.2 Partitions and subsets

5.2.1 Partition function

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

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

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

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

5.2.2 Binomials

binomialModPrime.h  
**Description:** Lucas’ thm: Let  $n, m$  be non-negative integers and  $p$  a prime. Write  $n = n_k p^k + \dots + n_1 p + n_0$  and  $m = m_k p^k + \dots + m_1 p + m_0$ . Then  $\binom{n}{m} \equiv \prod_{i=0}^k \binom{n_i}{m_i} \pmod{p}$ . fact and invfact must hold pre-computed factorials / inverse factorials, e.g. from ModInverse.h.  
**Time:**  $\mathcal{O}(\log_p n)$

```
10 lines
ll chooseModP(ll n, ll m, int p, vi& fact, vi& invfact) {
    ll c = 1;
    while (n || m) {
        ll a = n % p, b = m % p;
        if (a < b) return 0;
        c = c * fact[a] % p * invfact[b] % p * invfact[a - b]
        ↪ % p;
        n /= p; m /= p;
    }
    return c;
} // hash-cpp-all = 81845faa6ecd635c391e4f0134f0676c
```

multinomial.h

**Description:** Computes  $\binom{k_1 + \dots + k_n}{k_1, k_2, \dots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \dots k_n!}$ . 6 lines

```
ll multinomial(vi& v) {
    ll c = 1, m = v.empty() ? 1 : v[0];
    rep(i,1,sz(v)) rep(j,0,v[i])
        c = c * ++m / (j+1);
    return c;
} // hash-cpp-all = a0a3128f6afa721166feb182b82f130
```

5.3 General purpose numbers

5.3.1 Bernoulli numbers

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

Sums of powers:

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

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.

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

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

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

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

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

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

- sub-diagonal monotone paths in an  $n \times n$  grid.
- strings with  $n$  pairs of parenthesis, correctly nested.
- binary trees 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 Other

nim-product.cpp

**Description:** Nim Product.

17 lines

```
using ull = uint64_t;
ull _nimProd2[64][64];
ull nimProd2(int i, int j) {
    if (_nimProd2[i][j]) return _nimProd2[i][j];
    if ((i & j) == 0) return _nimProd2[i][j] = 1ull << (i|j)
    ↪;
    int a = (i&j) & -(i&j);
    return _nimProd2[i][j] = nimProd2(i ^ a, j) ^ nimProd2(
    ↪ i ^ a | (a-1), (j ^ a) | (i & (a-1)));
}
ull nimProd(ull x, ull y) {
    ull res = 0;
    for (int i = 0; x >> i; i++)
        if ((x >> i) & 1)
            for (int j = 0; y >> j; j++)
                if ((y >> j) & 1)
                    res ^= nimProd2(i, j);
    return res;
} // hash-cpp-all = e0411498c7a77d77ae793efab5500851
```

schreier-sims.cpp

**Description:** Check group membership of permutation groups

52 lines

```
struct Perm {
    int a[N];
    Perm() {
```

```
    for (int i = 1; i <= n; ++i) a[i] = i;
    }
    friend Perm operator* (const Perm &lhs, const Perm &rhs)
    ↪ {
        static Perm res;
        for (int i = 1; i <= n; ++i) res.a[i] = lhs.a[rhs.a[i]
    ↪];
        return res;
    }
    friend Perm inv(const Perm &cur) {
        static Perm res;
        for (int i = 1; i <= n; ++i) res.a[cur.a[i]] = i;
        return res;
    }
};
class Group {
    bool flag[N];
    Perm w[N];
    std::vector<Perm> x;
public:
    void clear(int p) {
        memset(flag, 0, sizeof flag);
        for (int i = 1; i <= n; ++i) w[i] = Perm();
        flag[p] = true;
        x.clear();
    }
    friend bool check(const Perm&, int);
    friend void insert(const Perm&, int);
    friend void updateX(const Perm&, int);
} g[N];
bool check(const Perm &cur, int k) {
    if (!k) return true;
    int t = cur.a[k];
    return g[k].flag[t] ? check(g[k].w[t] * cur, k - 1) :
    ↪ false;
}
void updateX(const Perm&, int);
void insert(const Perm &cur, int k) {
    if (check(cur, k)) return;
    g[k].x.push_back(cur);
    for (int i = 1; i <= n; ++i) if (g[k].flag[i]) updateX(
    ↪ cur * inv(g[k].w[i]), k);
}
void updateX(const Perm &cur, int k) {
    int t = cur.a[k];
    if (g[k].flag[t]) {
        insert(g[k].w[t] * cur, k - 1);
    } else {
        g[k].w[t] = inv(cur);
        g[k].flag[t] = true;
        for (int i = 0; i < g[k].x.size(); ++i) updateX(g[k].x
    ↪ [i] * cur, k);
    }
} // hash-cpp-all = 949a6e50dbdae9cda09928c7eabedbc
```

## Graph (6)

### 6.1 Euler walk

EulerWalk.h

**Description:** Eulerian undirected/directed path/cycle algorithm. Returns a list of nodes in the Eulerian path/cycle with src at both start and end, or empty list if no cycle/path exists. To get edge indices back, also put it->second in s (and then ret).

**Time:**  $\mathcal{O}(E)$  where  $E$  is the number of edges.

27 lines

```
struct V {
    vector<pii> outs; // (dest, edge index)
    int nins = 0;
};

vi euler_walk(vector<V>& nodes, int nedges, int src=0) {
    int c = 0;
    trav(n, nodes) c += abs(n.nins - sz(n.outs));
    if (c > 2) return {};
    vector<vector<pii>::iterator> its;
    trav(n, nodes)
        its.push_back(n.outs.begin());
    vector<bool> eu(nedges);
    vi ret, s = {src};
    while(!s.empty()) {
        int x = s.back();
        auto& it = its[x], end = nodes[x].outs.end();
        while(it != end && eu[it->second]) ++it;
        if(it == end) { ret.push_back(x); s.pop_back(); }
        else { s.push_back(it->first); eu[it->second] = true;
    ↪ }
    }
    if(sz(ret) != nedges+1)
        ret.clear(); // No Eulerian cycles/paths.
    // else, non-cycle if ret.front() != ret.back()
    reverse(all(ret));
    return ret;
} // hash-cpp-all = f8bd47ef7a9ffb45f7541c41e476f5f9
```

## 6.2 Network flow

PushRelabel.h

**Description:** Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.

**Time:**  $\mathcal{O}(V^2\sqrt{E})$

51 lines

```
typedef ll Flow;
struct Edge {
    int dest, back;
    Flow f, c;
};

struct PushRelabel {
    vector<vector<Edge>> g;
    vector<Flow> ec;
    vector<Edge*> cur;
    vector<vi> hs; vi H;
    PushRelabel(int n) : g(n), ec(n), cur(n), hs(2*n), H(n)
    ↪ {}
}

void add_edge(int s, int t, Flow cap, Flow rcap=0) {
    if (s == t) return;
    Edge a = {t, sz(g[t]), 0, cap};
    Edge b = {s, sz(g[s]), 0, rcap};
    g[s].push_back(a);
    g[t].push_back(b);
}

void add_flow(Edge& e, Flow f) {
    Edge &back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f; e.c -= f; ec[e.dest] += f;
    back.f -= f; back.c += f; ec[back.dest] -= f;
}

Flow maxflow(int s, int t) {
    int v = sz(g); H[s] = v; ec[t] = 1;
    vi co(2*v); co[0] = v-1;
```

```

rep(i,0,v) cur[i] = g[i].data();
trav(e, g[s]) add_flow(e, e.c);

for (int hi = 0;;) {
    while (hs[hi].empty()) if (!hi--) return -ec[s];
    int u = hs[hi].back(); hs[hi].pop_back();
    while (ec[u] > 0) // discharge u
        if (cur[u] == g[u].data() + sz(g[u])) {
            H[u] = 1e9;
            trav(e, g[u]) if (e.c && H[u] > H[e.dest]+1)
                H[u] = H[e.dest]+1, cur[u] = &e;
            if (++co[H[u]], !--co[hi] && hi < v)
                rep(i,0,v) if (hi < H[i] && H[i] < v)
                    --co[H[i]], H[i] = v + 1;
            hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[u]->dest]+1)
            add_flow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
    }
}
}; // hash-cpp-all = aaa2dd3fd7d9e6d994b295a959664c9a

```

### MinCostMaxFlow.h

**Description:** Min-cost max-flow.  $\text{cap}[i][j] \neq \text{cap}[j][i]$  is allowed; double edges are not. If costs can be negative, call `setpi` before `maxflow`, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.

**Time:** Approximately  $\mathcal{O}(E^2)$

81 lines

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

```

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

```

```

struct MCMF {
    int N;
    vector<vi> ed, red;
    vector<VL> cap, flow, cost;
    vi seen;
    VL dist, pi;
    vector<pii> par;

    MCMF(int N) :
        N(N), ed(N), red(N), cap(N, VL(N)), flow(cap), cost(
            ↪cap),
        seen(N), dist(N), pi(N), par(N) {}

    void addEdge(int from, int to, ll cap, ll cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
        ed[from].push_back(to);
        red[to].push_back(from);
    }

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

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

        auto relax = [&](int i, ll cap, ll cost, int dir) {
            ll val = di - pi[i] + cost;
            if (cap && val < dist[i]) {
                dist[i] = val;

```

```

        par[i] = {s, dir};
        if (its[i] == q.end()) its[i] = q.push({-dist[i],
            ↪i});
        else q.modify(its[i], {-dist[i], i});
    }
};

while (!q.empty()) {
    s = q.top().second; q.pop();
    seen[s] = 1; di = dist[s] + pi[s];
    trav(i, ed[s]) if (!seen[i])
        relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
    trav(i, red[s]) if (!seen[i])
        relax(i, flow[i][s], -cost[i][s], 0);
}
rep(i,0,N) pi[i] = min(pi[i] + dist[i], INF);
}

pair<ll, ll> maxflow(int s, int t) {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
        ll fl = INF;
        for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p
            ↪)
            fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][
                ↪p]);
        totflow += fl;
        for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p
            ↪)
            if (r) flow[p][x] += fl;
            else flow[x][p] -= fl;
    }
    rep(i,0,N) rep(j,0,N) totcost += cost[i][j] * flow[i][
        ↪j];
    return {totflow, totcost};
}

// If some costs can be negative, call this before
↪maxflow:
void setpi(int s) { // (otherwise, leave this out)
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--)
        rep(i,0,N) if (pi[i] != INF)
            trav(to, ed[i]) if (cap[i][to])
                if ((v = pi[i] + cost[i][to]) < pi[to])
                    pi[to] = v, ch = 1;
    assert(it >= 0); // negative cost cycle
}
}; // hash-cpp-all = 6915cee27314b77b2f5e256f1a96cdc0

```

### EdmondsKarp.h

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

35 lines

```

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

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

```

```

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

flow += inc;
for (int y = sink; y != source; y = par[y]) {
    int p = par[y];
    if ((graph[p][y] - inc) <= 0) graph[p].erase(y);
    graph[y][p] += inc;
}
}
}; // hash-cpp-all = 979bb9ccc85090e328209bf565a2af26

```

### MinCut.h

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

1 lines

```
// hash-cpp-all = d41d8cd98f00b204e9800998ecf8427e
```

### GlobalMinCut.h

**Description:** Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.

**Time:**  $\mathcal{O}(V^3)$

31 lines

```

pair<int, vi> GetMinCut(vector<vi>& weights) {
    int N = sz(weights);
    vi used(N), cut, best_cut;
    int best_weight = -1;

    for (int phase = N-1; phase >= 0; phase--) {
        vi w = weights[0], added = used;
        int prev, k = 0;
        rep(i,0,phase) {
            prev = k;
            k = -1;
            rep(j,1,N)
                if (!added[j] && (k == -1 || w[j] > w[k])) k = j;
            if (i == phase-1) {
                rep(j,0,N) weights[prev][j] += weights[k][j];
                rep(j,0,N) weights[j][prev] = weights[prev][j];
                used[k] = true;
                cut.push_back(k);
                if (best_weight == -1 || w[k] < best_weight) {
                    best_cut = cut;
                    best_weight = w[k];
                }
            } else {
                rep(j,0,N)
                    w[j] += weights[k][j];
                added[k] = true;
            }
        }
    }
}

```

```
    return {best_weight, best_cut};
} // hash-cpp-all = 03261f13665169d285596975383c72b3
```

### 6.3 Matching

#### hopcroftKarp.h

**Description:** Find a maximum matching in a bipartite graph.

**Usage:** vi ba(m, -1); hopcroftKarp(g, ba);

**Time:**  $\mathcal{O}(\sqrt{VE})$

```
bool dfs(int a, int layer, const vector<vi>& g, vi& btoa,
         vi& A, vi& B) {
    if (A[a] != layer) return 0;
    A[a] = -1;
    trav(b, g[a]) if (B[b] == layer + 1) {
        B[b] = -1;
        if (btoa[b] == -1 || dfs(btoa[b], layer+2, g, btoa, A,
            ↪ B))
            return btoa[b] = a, 1;
    }
    return 0;
}
```

```
int hopcroftKarp(const vector<vi>& g, vi& btoa) {
    int res = 0;
    vi A(g.size()), B(btoa.size()), cur, next;
    for (;;) {
        fill(all(A), 0);
        fill(all(B), -1);
        cur.clear();
        trav(a, btoa) if(a != -1) A[a] = -1;
        rep(a, 0, sz(g)) if(A[a] == 0) cur.push_back(a);
        for (int lay = 1; lay += 2) {
            bool islast = 0;
            next.clear();
            trav(a, cur) trav(b, g[a]) {
                if (btoa[b] == -1) {
                    B[b] = lay;
                    islast = 1;
                }
                else if (btoa[b] != a && B[b] == -1) {
                    B[b] = lay;
                    next.push_back(btoa[b]);
                }
            }
            if (islast) break;
            if (next.empty()) return res;
            trav(a, next) A[a] = lay+1;
            cur.swap(next);
        }
        rep(a, 0, sz(g)) {
            if(dfs(a, 0, g, btoa, A, B))
                ++res;
        }
    }
} // hash-cpp-all = ee9fe891045fe156e995ef0276b80af6
```

#### DFSMatching.h

**Description:** This is a simple matching algorithm but should be just fine in most cases. Graph  $g$  should be a list of neighbours of the left partition.  $n$  is the size of the left partition and  $m$  is the size of the right partition. If you want to get the matched pairs,  $match[i]$  contains match for vertex  $i$  on the right side or  $-1$  if it's not matched.

**Time:**  $\mathcal{O}(EV)$  where  $E$  is the number of edges and  $V$  is the number of vertices.

```
vi match;
```

```
vector<bool> seen;
bool find(int j, const vector<vi>& g) {
    if (match[j] == -1) return 1;
    seen[j] = 1; int di = match[j];
    trav(e, g[di])
        if (!seen[e] && find(e, g)) {
            match[e] = di;
            return 1;
        }
    return 0;
}
int dfs_matching(const vector<vi>& g, int n, int m) {
    match.assign(m, -1);
    rep(i, 0, n) {
        seen.assign(m, 0);
        trav(j, g[i])
            if (find(j, g)) {
                match[j] = i;
                break;
            }
    }
    return m - (int)count(all(match), -1);
} // hash-cpp-all = 178c94b6091dc009a15d348aef80dff0
```

#### WeightedMatching.h

**Description:** Min cost bipartite matching. Negate costs for max cost.

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

```
typedef vector<double> vd;
bool zero(double x) { return fabs(x) < 1e-10; }
double MinCostMatching(const vector<vd>& cost, vi& L, vi&
    ↪ R) {
    int n = sz(cost), mated = 0;
    vd dist(n), u(n), v(n);
    vi dad(n), seen(n);

    rep(i, 0, n) {
        u[i] = cost[i][0];
        rep(j, 1, n) u[i] = min(u[i], cost[i][j]);
    }
    rep(j, 0, n) {
        v[j] = cost[0][j] - u[0];
        rep(i, 1, n) v[j] = min(v[j], cost[i][j] - u[i]);
    }

    L = R = vi(n, -1);
    rep(i, 0, n) rep(j, 0, n) {
        if (R[j] != -1) continue;
        if (zero(cost[i][j] - u[i] - v[j])) {
            L[i] = j;
            R[j] = i;
            mated++;
            break;
        }
    }

    for (; mated < n; mated++) { // until solution is
        ↪ feasible
        int s = 0;
        while (L[s] != -1) s++;
        fill(all(dad), -1);
        fill(all(seen), 0);
        rep(k, 0, n)
            dist[k] = cost[s][k] - u[s] - v[k];

        int j = 0;
        for (;;) {
```

```
        j = -1;
        rep(k, 0, n) {
            if (seen[k]) continue;
            if (j == -1 || dist[k] < dist[j]) j = k;
        }
        seen[j] = 1;
        int i = R[j];
        if (i == -1) break;
        rep(k, 0, n) {
            if (seen[k]) continue;
            auto new_dist = dist[j] + cost[i][k] - u[i] - v[k]
                ↪ ;
            if (dist[k] > new_dist) {
                dist[k] = new_dist;
                dad[k] = j;
            }
        }

        rep(k, 0, n) {
            if (k == j || !seen[k]) continue;
            auto w = dist[k] - dist[j];
            v[k] += w, u[R[k]] -= w;
        }
        u[s] += dist[j];

        while (dad[j] >= 0) {
            int d = dad[j];
            R[j] = R[d];
            L[R[j]] = j;
            j = d;
        }
        R[j] = s;
        L[s] = j;
    }
    auto value = vd(1)[0];
    rep(i, 0, n) value += cost[i][L[i]];
    return value;
} // hash-cpp-all = 055ca9687f72b2dd5e2d2c6921f1c51d
```

#### GeneralMatching.h

**Description:** Matching for general graphs. Fails with probability  $N/\text{mod}$ .

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

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

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

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

    vi has(M, 1); vector<pii> ret;
```

```

rep(it,0,M/2) {
    rep(i,0,M) if (has[i])
        rep(j,i+1,M) if (A[i][j] && mat[i][j]) {
            fi = i; fj = j; goto done;
        } assert(0); done:
    if (fj < N) ret.emplace_back(fi, fj);
    has[fi] = has[fj] = 0;
    rep(sw,0,2) {
        ll a = modpow(A[fi][fj], mod-2);
        rep(i,0,M) if (has[i] && A[i][fj]) {
            ll b = A[i][fj] * a % mod;
            rep(j,0,M) A[i][j] = (A[i][j] - A[fi][j] * b) %
                mod;
        }
        swap(fi,fj);
    }
}
return ret;
} // hash-cpp-all = bb8be4f4f83b4e4ccafaebf8534e4f82

```

### MinimumVertexCover.h

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

"DFSMatching.h" 20 lines

```

vi cover(vector<vi>& g, int n, int m) {
    int res = dfs_matching(g, n, m);
    seen.assign(m, false);
    vector<bool> lfound(n, true);
    trav(it, match) if (it != -1) lfound[it] = false;
    vi q, cover;
    rep(i,0,n) if (lfound[i]) q.push_back(i);
    while (!q.empty()) {
        int i = q.back(); q.pop_back();
        lfound[i] = 1;
        trav(e, g[i]) if (!seen[e] && match[e] != -1) {
            seen[e] = true;
            q.push_back(match[e]);
        }
    }
    rep(i,0,n) if (!lfound[i]) cover.push_back(i);
    rep(i,0,m) if (seen[i]) cover.push_back(n+i);
    assert(sz(cover) == res);
    return cover;
} // hash-cpp-all = 9eeda105ef373dfc9bd11d0139e4fc82

```

## 6.4 DFS algorithms

### SCC.h

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

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

**Time:**  $\mathcal{O}(E + V)$  24 lines

```

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

```

```

if (low == val[j]) {
    do {
        x = z.back(); z.pop_back();
        comp[x] = ncomps;
        cont.push_back(x);
    } while (x != j);
    f(cont); cont.clear();
    ncomps++;
}
return val[j] = low;
}
template<class G, class F> void scc(G& g, F f) {
    int n = sz(g);
    val.assign(n, 0); comp.assign(n, -1);
    Time = ncomps = 0;
    rep(i,0,n) if (comp[i] < 0) dfs(i, g, f);
} // hash-cpp-all = 2c7a153ddd31436517cf3ad28efa4ac5

```

### BiconnectedComponents.h

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

**Usage:** `int eid = 0; ed.resize(N);` for each edge  $(a,b)$  { `ed[a].emplace_back(b, eid);` `ed[b].emplace_back(a, eid++);` } `bicomps([&](const vi& edgelist) {...});` **Time:**  $\mathcal{O}(E + V)$  33 lines

```

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

template<class F>
void bicomps(F f) {
    num.assign(sz(ed), 0);
    rep(i,0,sz(ed)) if (!num[i]) dfs(i, -1, f);
} // hash-cpp-all = e183ffd0266ca965525c2788c540f8f0

```

### 2sat.h

**Description:** Calculates a valid assignment to boolean variables  $a, b, c, \dots$  to a 2-SAT problem, so that an expression of the type  $(a \vee b) \wedge (\neg a \vee c) \wedge (d \vee \neg b) \wedge \dots$  becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions ( $\sim x$ ).

**Usage:** `TwoSat ts(number of boolean variables);` `ts.either(0, ~3);` // Var 0 is true or var 3 is false `ts.set_value(2);` // Var 2 is true `ts.at_most_one({0, ~1, 2});` //  $\leq 1$  of vars 0, ~1 and 2 are true `ts.solve();` // Returns true iff it is solvable `ts.values[0..N-1]` holds the assigned values to the vars **Time:**  $\mathcal{O}(N + E)$ , where  $N$  is the number of boolean variables, and  $E$  is the number of clauses. 57 lines

```

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

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

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

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

    void set_value(int x) { either(x, x); }

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

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

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

```



```

    rep(i,0,N) if (comp[2*i] == comp[2*i+1]) return 0;
    return 1;
}; // hash-cpp-all = 288fb44b52e9016a30ce849e38390eb9

```

## 6.5 Heuristics

### MaximalCliques.h

**Description:** Runs a callback for all maximal cliques in a graph (given as a symmetric bitset matrix; self-edges not allowed). Possible optimization: on the top-most recursion level, ignore 'cands', and go through nodes in order of increasing degree, where degrees go down as nodes are removed.

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

```

12 lines
typedef bitset<128> B;
template<class F>
void cliques(vector<B>& eds, F f, B P = ~B(), B X={}, B R
    ⇐={}) {
    if (!P.any()) { if (!X.any()) f(R); return; }
    auto q = (P | X)._Find_first();
    auto cands = P & ~eds[q];
    rep(i,0,sz(eds)) if (cands[i]) {
        R[i] = 1;
        cliques(eds, f, P & eds[i], X & eds[i], R);
        R[i] = P[i] = 0; X[i] = 1;
    }
}; // hash-cpp-all = b0d5b15b7ebdcde7ff57f0761c050583

```

### graph-clique.cpp

**Description:** Max clique  $N \leq 64$ . Bit trick for speed. clique solver calculates both size and constitution of maximum clique uses bit operation to accelerate searching graph size limit is 63, the graph should be undirected can optimize to calculate on each component, and sort on vertex degrees can be used to solve maximum independent set

```

80 lines
class clique {
public:
    static const long long ONE = 1;
    static const long long MASK = (1 << 21) - 1;
    char* bits;
    int n, size, cmax[63];
    long long mask[63], cons;
    // initiate lookup table
    clique() {
        bits = new char[1 << 21];
        bits[0] = 0;
        for (int i = 1; i < (1<<21); ++i)
            bits[i] = bits[i >> 1] + (i & 1);
    }
    ~clique() {
        delete bits;
    }
    // search routine
    bool search(int step,int siz,LL mor,LL con);
    // solve maximum clique and return size
    int sizeClique(vector<vector<int>> &mat);
    // solve maximum clique and return set
    vector<int>getClq(vector<vector<int>> &mat);
};
// step is node id, size is current sol., more is
⇐available mask, cons is constitution mask
bool clique::search(int step, int size,
    LL more, LL cons) {
    if (step >= n) {
        // a new solution reached
        this->size = size;
        this->cons = cons;
    }
}

```

```

return true;
}
long long now = ONE << step;
if ((now & more) > 0) {
    long long next = more & mask[step];
    if (size + bits[next & MASK] +
        bits[(next >> 21) & MASK] +
        bits[next >> 42] >= this->size
        && size + cmax[step] > this->size) {
        // the current node is in the clique
        if (search(step+1, size+1, next, cons|now))
            return true;
    }
}
}
long long next = more & ~now;
if (size + bits[next & MASK] +
    bits[(next >> 21) & MASK] +
    bits[next >> 42] > this->size) {
    // the current node is not in the clique
    if (search(step + 1, size, next, cons))
        return true;
}
return false;
}
// solve maximum clique and return size
int clique::sizeClique(vector<vector<int>> &mat) {
    n = mat.size();
    // generate mask vectors
    for (int i = 0; i < n; ++i) {
        mask[i] = 0;
        for (int j = 0; j < n; ++j)
            if (mat[i][j] > 0) mask[i] |= ONE << j;
    }
    size = 0;
    for (int i = n - 1; i >= 0; --i) {
        search(i + 1, 1, mask[i], ONE << i);
        cmax[i] = size;
    }
    return size;
}
// calls sizeClique and restore cons
vector<int> clique::getClq(
    vector<vector<int>> &mat) {
    sizeClique(mat);
    vector<int> ret;
    for (int i = 0; i < n; ++i)
        if ((cons & (ONE << i)) > 0) ret.push_back(i);
    return ret;
} // hash-cpp-all = fbf6cf3d9cbb4f5d32d6245cfbe40fd0

```

### cycle-counting.cpp

**Description:** Counts 3 and 4 cycles

```

62 lines
<bits/stdc++.h>
#define P 10000000007
#define N 110000

int n, m;
vector<int> go[N], lk[N];

int w[N];
int circle3() { // hash-cpp-1
    int ans=0;
    for (int i = 1; i <= n; i++)
        w[i]=0;

    for (int x = 1; x <= n; x++) {
        for(int y:lk[x])w[y]=1;
    }
}

```

```

for(int y:lk[x])for(int z:lk[y])if(w[z]){
    ans=(ans+go[x].size()+go[y].size()+go[z].size()-6)%P
    ⇐;
}

for(int y:lk[x])w[y]=0;
}
return ans;
} // hash-cpp-1 = 719dcec935e20551fd984c12c3bfa3ba

int deg[N], pos[N], id[N];

int circle4() { // hash-cpp-2
    for (int i = 1; i <= n; i++)
        w[i]=0;
    int ans=0;
    for (int x = 1; x <= n; x++) {
        for(int y:go[x])for(int z:lk[y])if(pos[z]>pos[x]){
            ans=(ans+w[z])%P;
            w[z]++;
        }
        for(int y:go[x])for(int z:lk[y])w[z]=0;
    }
    return ans;
} // hash-cpp-2 = 39b3aaf47e9fdc4dfff3fdffdf22d3a8e

inline bool cmp(const int &x,const int &y){
    return deg[x]<deg[y];
}

void init() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
        deg[i] = 0, go[i].clear(), lk[i].clear();
    while (m--) {
        int a,b;
        scanf("%d%d",&a,&b);
        deg[a]++;deg[b]++;
        go[a].push_back(b);go[b].push_back(a);
    }
    for (int i = 1; i <= n; i++)
        id[i] = i;
    sort(id+1,id+1+n,cmp);
    for (int i = 1; i <= n; i++) pos[id[i]]=i;
    for (int x = 1; x <= n; x++)
        for(int y:go[x])
            if(pos[y]>pos[x])lk[x].push_back(y);
}

```

## 6.6 Trees

### CompressTree.h

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

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

```

20 lines
"LCA.h"
vpi compressTree(LCA& lca, const vi& subset) {
    static vi rev; rev.resize(sz(lca.dist));
    vi li = subset, &T = lca.time;
    auto cmp = [&](int a, int b) { return T[a] < T[b]; };
    sort(all(li), cmp);
    int m = sz(li)-1;
    rep(i,0,m) {
        int a = li[i], b = li[i+1];
    }
}

```

```

    li.push_back(lca.query(a, b));
}
sort(all(li), cmp);
li.erase(unique(all(li)), li.end());
rep(i, 0, sz(li)) rev[li[i]] = i;
vpl ret = {pli(0, li[0])};
rep(i, 0, sz(li)-1) {
    int a = li[i], b = li[i+1];
    ret.emplace_back(rev[lca.query(a, b)], b);
}
return ret;
} // hash-cpp-all = dabd7520dba8306be5675979add23011

```

## MatrixTree.h

**Description:** To count the number of spanning trees in an undirected graph  $G$ : create an  $N \times N$  matrix  $mat$ , and for each edge  $(a, b) \in G$ , do  $mat[a][a]++$ ,  $mat[b][b]++$ ,  $mat[a][b]--$ ,  $mat[b][a]--$ . Remove the last row and column, and take the determinant.

1 lines

```
// hash-cpp-all = d41d8cd98f00b204e9800998ecf8427e
```

## 6.7 Other

### directed-MST.cpp

**Description:** Finds the minimum spanning arborescence from the root. (any more notes?)

73 lines

```
#define rep(i, n) for (int i = 0; i < n; i++)
```

```

#define N 110000
#define M 110000
#define inf 2000000000

```

```

struct edg {
    int u, v;
    int cost;
} E[M], E_copy[M];

```

```
int In[N], ID[N], vis[N], pre[N];
```

```
// edges pointed from root.
```

```

int Directed_MST(int root, int NV, int NE) {
    for (int i = 0; i < NE; i++)
        E_copy[i] = E[i];
    int ret = 0;
    int u, v;
    while (true) {
        rep(i, NV) In[i] = inf;
        rep(i, NE) {
            u = E_copy[i].u;
            v = E_copy[i].v;
            if (E_copy[i].cost < In[v] && u != v) {
                In[v] = E_copy[i].cost;
                pre[v] = u;
            }
        }
        rep(i, NV) {
            if (i == root) continue;
            if (In[i] == inf) return -1; // no solution
        }

        int cnt = 0;
        rep(i, NV) {
            ID[i] = -1;
            vis[i] = -1;
        }
        In[root] = 0;

```

```

        rep(i, NV) {
            ret += In[i];
            int v = i;
            while (vis[v] != i && ID[v] == -1 && v != root)
                vis[v] = i;
                v = pre[v];
            }
            if (v != root && ID[v] == -1) {
                for (u = pre[v]; u != v; u = pre[u]) {
                    ID[u] = cnt;
                }
                ID[v] = cnt++;
            }
        }
        if (cnt == 0) break;
        rep(i, NV) {
            if (ID[i] == -1) ID[i] = cnt++;
        }
        rep(i, NE) {
            v = E_copy[i].v;
            E_copy[i].u = ID[E_copy[i].u];
            E_copy[i].v = ID[E_copy[i].v];
            if (E_copy[i].u != E_copy[i].v) {
                E_copy[i].cost -= In[v];
            }
        }
        NV = cnt;
        root = ID[root];
    }
    return ret;
} // hash-cpp-all = 8d5af0080b124fcb50e7cbefa704eaa

```

### graph-dominator-tree.cpp

**Description:** Dominator Tree.

107 lines

```
#define N 110000 //max number of vertices
```

```

vector<int> succ[N], prod[N], bucket[N], dom_t[N];
int semi[N], anc[N], idom[N], best[N], fa[N], tmp_idom[N];
int dfn[N], redfn[N];
int child[N], size[N];
int timestamp;

```

```

void dfs(int now) { // hash-cpp-1
    dfn[now] = ++timestamp;
    redfn[timestamp] = now;
    anc[timestamp] = idom[timestamp] = child[timestamp] =
        size[timestamp] = 0;
    semi[timestamp] = best[timestamp] = timestamp;
    int sz = succ[now].size();
    for (int i = 0; i < sz; ++i) {
        if (dfn[succ[now][i]] == -1) {
            dfs(succ[now][i]);
            fa[dfn[succ[now][i]]] = dfn[now];
        }
        prod[dfn[succ[now][i]]].push_back(dfn[now]);
    }
} // hash-cpp-1 = 6412bfd6a0d21b66ddaa51ea79cbe7bd

```

```

void compress(int now) { // hash-cpp-2
    if (anc[anc[now]] != 0) {
        compress(anc[now]);
        if (semi[best[now]] > semi[best[anc[now]]])
            best[now] = best[anc[now]];
        anc[now] = anc[anc[now]];
    }
}

```

```

    }
} // hash-cpp-2 = 1c9444eb3f768b7af8741fafbf3afb5a

inline int eval(int now) { // hash-cpp-3
    if (anc[now] == 0)
        return now;
    else {
        compress(now);
        return semi[best[anc[now]]] >= semi[best[now]] ? best[anc[now]] : best[now];
    }
} // hash-cpp-3 = 4e235f39666315b46dcd3455d5f866d1

inline void link(int v, int w) { // hash-cpp-4
    int s = w;
    while (semi[best[w]] < semi[best[child[w]]]) {
        if (size[s] + size[child[child[s]]] >= 2 * size[child[s]])
            anc[child[s]] = s;
            child[s] = child[child[s]];
        } else {
            size[child[s]] = size[s];
            s = anc[s] = child[s];
        }
    }
    best[s] = best[w];
    size[v] += size[w];
    if (size[v] < 2 * size[w])
        swap(s, child[v]);
    while (s != 0) {
        anc[s] = v;
        s = child[s];
    }
} // hash-cpp-4 = 270548fd021351ae21e97878f367b6f9

// idom[n] and other vertices that cannot be reached from
// n will be 0
void Lengauer_Tarjan(int n) { // n is the root's number //
    hash-cpp-5
    memset(dfn, -1, sizeof dfn);
    memset(fa, -1, sizeof fa);
    timestamp = 0;
    dfs(n);
    fa[1] = 0;
    for (int w = timestamp; w > 1; --w) {
        int sz = prod[w].size();
        for (int i = 0; i < sz; ++i) {
            int u = eval(prod[w][i]);
            if (semi[w] > semi[u])
                semi[w] = semi[u];
        }
        bucket[semi[w]].push_back(w);
        //anc[w] = fa[w]; link operation for o(mlogm) version
        link(fa[w], w);
        if (fa[w] == 0)
            continue;
        sz = bucket[fa[w]].size();
        for (int i = 0; i < sz; ++i) {
            int u = eval(bucket[fa[w]][i]);
            if (semi[u] < fa[w])
                idom[bucket[fa[w]][i]] = u;
            else
                idom[bucket[fa[w]][i]] = fa[w];
        }
        bucket[fa[w]].clear();
    }
    for (int w = 2; w <= timestamp; ++w) {

```

```

    if(idom[w] != semi[w])
        idom[w] = idom[idom[w]];
}
idom[1] = 0;
for(int i = timestamp; i > 1; --i) {
    if(fa[i] == -1)
        continue;
    dom_t[idom[i]].push_back(i);
}
memset(tmp_idom, 0, sizeof tmp_idom);
for (int i = 1; i <= timestamp; i++)
    tmp_idom[redfn[i]] = redfn[idom[i]];
memcpy(idom, tmp_idom, sizeof idom);
} // hash-cpp-5 = f49c40461d9222d8d39b28b0de66828

```

## graph-negative-cycle.cpp

**Description:** negative cycle

33 lines

```

double b[N][N];

double dis[N];
int vis[N], pc[N];

bool dfs(int k) {
    vis[k] += 1; pc[k] = true;
    if (vis[k] > N)
        return true;
    for (int i = 0; i < N; i++)
        if (dis[k] + b[k][i] < dis[i]) {
            dis[i] = dis[k] + b[k][i];
            if (!pc[i]) {
                if (dfs(i))
                    return true;
            } else return true;
        }
    pc[k] = false;
    return false;
}

bool chk(double d) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            b[i][j] = -a[i][j] + d;
        }
    for (int i = 0; i < N; i++)
        vis[i] = false, dis[i] = 0, pc[i] = false;
    for (int i = 0; i < N; i++)
        if (!vis[i] && dfs(i))
            return true;
    return false;
} // hash-cpp-all = ec5cf9bc61e058959ce8649f1e707b1b

```

## Geometry (7)

### 7.1 Geometric primitives

#### Point.h

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

25 lines

```

template<class T>
struct Point {
    typedef Point P;
    T x, y;
    explicit Point(T x=0, T y=0) : x(x), y(y) {}
}

```

```

bool operator<(P p) const { return tie(x,y) < tie(p.x,p.
    ↪y); }
bool operator==(P p) const { return tie(x,y)==tie(p.x,p.
    ↪y); }
P operator+(P p) const { return P(x+p.x, y+p.y); }
P operator-(P p) const { return P(x-p.x, y-p.y); }
P operator*(T d) const { return P(x*d, y*d); }
P operator/(T d) const { return P(x/d, y/d); }
T dot(P p) const { return x*p.x + y*p.y; }
T cross(P p) const { return x*p.y - y*p.x; }
T cross(P a, P b) const { return (a-*this).cross(b-*this
    ↪); }
T dist2() const { return x*x + y*y; }
double dist() const { return sqrt((double)dist2()); }
// angle to x-axis in interval [-pi, pi]
double angle() const { return atan2(y, x); }
P unit() const { return *this/dist(); } // makes dist()
    ↪=1
P perp() const { return P(-y, x); } // rotates +90
    ↪degrees
P normal() const { return perp().unit(); }
// returns point rotated 'a' radians ccw around the
    ↪origin
P rotate(double a) const {
    return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
}; // hash-cpp-all = f698493d48eeaa76063407bf935b5a3

```

#### lineDistance.h

##### Description:

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

"Point.h" 4 lines

```

template<class P>
double lineDist(const P& a, const P& b, const P& p) {
    return (double) (b-a).cross(p-a)/(b-a).dist();
} // hash-cpp-all = f6bf6b556d99b09f42b86d28d1eaa86d

```

#### SegmentDistance.h

##### Description:

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

**Usage:** Point<double> a, b(2,2), p(1,1);

bool onSegment = segDist(a,b,p) < 1e-10;

"Point.h" 6 lines

```

typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
    if (s==e) return (p-s).dist();
    auto d = (e-s).dist2(), t = min(d,max(.0, (p-s).dot(e-s)
    ↪);
    return ((p-s)*d-(e-s)*t).dist()/d;
} // hash-cpp-all = 5c88f46fb14a05a4f47bbd23b8a9c427

```

#### SegmentIntersection.h

##### Description:

If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists r1 is set to this point and 1 is returned. If no intersection point exists 0 is returned and if infinitely many exists 2 is returned and r1 and r2 are set to the two ends of the common line. The wrong position will be returned if P is Point<int> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long. Use segmentIntersectionQ to get just a true/false answer.

**Usage:** Point<double> intersection, dummy;

if (segmentIntersection(s1,e1,s2,e2,intersection,dummy)==1) cout << "segments intersect at " << intersection << endl;

"Point.h" 27 lines

```

template<class P>
int segmentIntersection(const P& s1, const P& e1,
    const P& s2, const P& e2, P& r1, P& r2) {
    if (e1==s1) {
        if (e2==s2) {
            if (e1==e2) { r1 = e1; return 1; } //all equal
            else return 0; //different point segments
        } else return segmentIntersection(s2,e2,s1,e1,r1,r2);
        ↪//swap
    }
    //segment directions and separation
    P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
    auto a = v1.cross(v2), a1 = v1.cross(d), a2 = v2.cross(d
    ↪);
    if (a == 0) { //if parallel
        auto b1=s1.dot(v1), c1=e1.dot(v1),
            b2=s2.dot(v1), c2=e2.dot(v1);
        if (a1 || a2 || max(b1,min(b2,c2))>min(c1,max(b2,c2)))
            return 0;
        r1 = min(b2,c2)<b1 ? s1 : (b2<c2 ? s2 : e2);
        r2 = max(b2,c2)>c1 ? e1 : (b2>c2 ? s2 : e2);
        return 2-(r1==r2);
    }
    if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
    if (0<a1 || a<-a1 || 0<a2 || a<-a2)
        return 0;
    r1 = s1-v1*a2/a;
    return 1;
} // hash-cpp-all = 1181b7cc739b442c29bada6b0d73a550

```

#### SegmentIntersectionQ.h

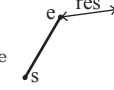
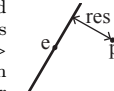
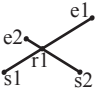
**Description:** Like segmentIntersection, but only returns true/false. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.

"Point.h" 16 lines

```

template<class P>
bool segmentIntersectionQ(P s1, P e1, P s2, P e2) {
    if (e1 == s1) {
        if (e2 == s2) return e1 == e2;
        swap(s1,s2); swap(e1,e2);
    }
    P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
    auto a = v1.cross(v2), a1 = d.cross(v1), a2 = d.cross(v2
    ↪);
    if (a == 0) { // parallel
        auto b1 = s1.dot(v1), c1 = e1.dot(v1),
            b2 = s2.dot(v1), c2 = e2.dot(v1);
        return !a1 && max(b1,min(b2,c2)) <= min(c1,max(b2,c2))
            ↪;
    }
    if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
}

```



```

    return (0 <= a1 && a1 <= a && 0 <= a2 && a2 <= a);
} // hash-cpp-all = 1ff4ba22bd0aefb04bf48cca4d6a7d8c

```

## lineIntersection.h

### Description:

If a unique intersection point of the lines going through  $s_1, e_1$  and  $s_2, e_2$  exists  $r$  is set to this point and 1 is returned. If no intersection point exists 0 is returned and if infinitely many exists -1 is returned. If  $s_1 == e_1$  or  $s_2 == e_2$  -1 is returned. The wrong position will be returned if  $P$  is  $\text{Point}<\text{int}>$  and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using  $\text{int}$  or long long.

**Usage:** `point<double> intersection;`  
if (1 == LineIntersection( $s_1, e_1, s_2, e_2, \text{intersection}$ ))  
cout << "intersection point at " << intersection << endl;

```

"Point.h" 9 lines
template<class P>
int lineIntersection(const P& s1, const P& e1, const P& s2
    ↪,
    const P& e2, P& r) {
    if ((e1-s1).cross(e2-s2)) { //if not parallel
        r = s2-(e2-s2)*(e1-s1).cross(s2-s1)/(e1-s1).cross(e2-
            ↪s2);
        return 1;
    } else
        return -(e1-s1).cross(s2-s1)==0 || s2==e2;
} // hash-cpp-all = aalf17f0dbde5177e697038a420bb078

```

## sideOf.h

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

**Usage:** `bool left = sideOf( $p_1, p_2, q$ )==1;`

```

"Point.h" 11 lines
template<class P>
int sideOf(const P& s, const P& e, const P& p) {
    auto a = (e-s).cross(p-s);
    return (a > 0) - (a < 0);
}
template<class P>
int sideOf(const P& s, const P& e, const P& p, double eps)
    ↪ {
    auto a = (e-s).cross(p-s);
    double l = (e-s).dist()*eps;
    return (a > l) - (a < -l);
} // hash-cpp-all = 2eb6fe62d7f3750fd3a0ec3d91329ed6

```

## onSegment.h

**Description:** Returns true iff  $p$  lies on the line segment from  $s$  to  $e$ . Intended for use with e.g. `Point<long long>` where overflow is an issue. Use (`segDist( $s, e, p$ )<=epsilon`) instead when using `Point<double>`.

```

"Point.h" 5 lines
template<class P>
bool onSegment(const P& s, const P& e, const P& p) {
    P ds = p-s, de = p-e;
    return ds.cross(de) == 0 && ds.dot(de) <= 0;
} // hash-cpp-all = 0b2b1c6866c98c2d2003acac0701e693

```

## linearTransformation.h

### Description:

Apply the linear transformation (translation, rotation and scaling) which takes line  $p_0-p_1$  to line  $q_0-q_1$  to point  $r$ .

```

"Point.h" 6 lines
typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) {
    P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
    return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.
        ↪dist2();
} // hash-cpp-all = 03a3061b3ef024b4e29ea06169932b21

```

## Angle.h

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

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

```

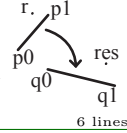
37 lines
struct Angle {
    int x, y;
    int t;
    Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
    Angle operator-(Angle b) const { return {x-b.x, y-b.y, t
        ↪}; }
    int quad() const {
        assert(x || y);
        if (y < 0) return (x >= 0) + 2;
        if (y > 0) return (x <= 0);
        return (x <= 0) * 2;
    }
    Angle t90() const { return {-y, x, t + (quad() == 3)}; }
    Angle t180() const { return {-x, -y, t + (quad() >= 2)}; }
        ↪
    Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
    // add a.dist2() and b.dist2() to also compare distances
    return make_tuple(a.t, a.quad(), a.y * (ll)b.x) <
        make_tuple(b.t, b.quad(), a.x * (ll)b.y);
}

```

```

// Given two points, this calculates the smallest angle
    ↪between
// them, i.e., the angle that covers the defined line
    ↪segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
    if (b < a) swap(a, b);
    return (b < a.t180() ?
        make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
    Angle r(a.x + b.x, a.y + b.y, a.t);
    if (a.t180() < r) r.t--;
    return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
    int tu = b.t - a.t; a.t = b.t;
    return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b <
        ↪a)};
} // hash-cpp-all = 1856c5d371c2f8f342a22615fa92cd54

```



## angleCmp.h

**Description:** Useful utilities for dealing with angles of rays from origin. OK for integers, only uses cross product. Doesn't support  $(0,0)$ .

```

22 lines
template<class P>
bool sameDir(P s, P t) {
    return s.cross(t) == 0 && s.dot(t) > 0;
}
// checks 180 <= s..t < 360?
template<class P>
bool isReflex(P s, P t) {
    auto c = s.cross(t);
    return c ? (c < 0) : (s.dot(t) < 0);
}
// operator < (s,t) for angles in [base,base+2pi)
template<class P>
bool angleCmp(P base, P s, P t) {
    int r = isReflex(base, s) - isReflex(base, t);
    return r ? (r < 0) : (0 < s.cross(t));
}
// is x in [s,t] taken ccw? 1/0/-1 for in/border/out
template<class P>
int angleBetween(P s, P t, P x) {
    if (sameDir(x, s) || sameDir(x, t)) return 0;
    return angleCmp(s, x, t) ? 1 : -1;
} // hash-cpp-all = 6edd25f30f9c69989bbd2115b4fdceda

```

## 7.2 Circles

### CircleIntersection.h

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

```

"Point.h" 14 lines
typedef Point<double> P;
bool circleIntersection(P a, P b, double r1, double r2,
    pair<P, P>* out) {
    P delta = b - a;
    assert(delta.x || delta.y || r1 != r2);
    if (!delta.x && !delta.y) return false;
    double r = r1 + r2, d2 = delta.dist2();
    double p = (d2 + r1*r1 - r2*r2) / (2.0 * d2);
    double h2 = r1*r1 - p*p*d2;
    if (d2 > r*r || h2 < 0) return false;
    P mid = a + delta*p, per = delta.perp() * sqrt(h2 / d2);
    *out = {mid + per, mid - per};
    return true;
} // hash-cpp-all = 828fbb1fff1469ed43b2284c8e07a06c

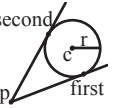
```

### circleTangents.h

#### Description:

Returns a pair of the two points on the circle with radius  $r$  centered around  $c$  whose tangent lines intersect  $p$ . If  $p$  lies within the circle  $\text{NaN}$ -points are returned.  $P$  is intended to be  $\text{Point}<\text{double}>$ . The first point is the one to the right as seen from the  $p$  towards  $c$ .

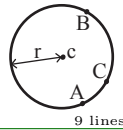
**Usage:** `typedef Point<double> P;`  
`pair<P,P> p = circleTangents(P(100,2),P(0,0),2);`  
"Point.h" 6 lines  
template<class P>
pair<P,P> circleTangents(const P& p, const P& c, double r)
 ↪ {
 P a = p-c;
 double x = r\*r/a.dist2(), y = sqrt(x-x\*x);
 return make\_pair(c+a\*x+a.perp()\*y, c+a\*x-a.perp()\*y);
} // hash-cpp-all = b70bc575e85c140131116e64926b4ce1



### circumcircle.h

**Description:**

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



```
"Point.h"
9 lines

typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
    return (B-A).dist()*(C-B).dist()*(A-C).dist()/
        abs((B-A).cross(C-A))/2;
}

P ccCenter(const P& A, const P& B, const P& C) {
    P b = C-A, c = B-A;
    return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)
        ↪/2;
} // hash-cpp-all = 1caa3aea364671cb961900d4811f0282
```

### MinimumEnclosingCircle.h

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

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

```
"circumcircle.h"
28 lines

pair<double, P> mec2(vector<P>& S, P a, P b, int n) {
    double hi = INFINITY, lo = -hi;
    rep(i,0,n) {
        auto si = (b-a).cross(S[i]-a);
        if (si == 0) continue;
        P m = ccCenter(a, b, S[i]);
        auto cr = (b-a).cross(m-a);
        if (si < 0) hi = min(hi, cr);
        else lo = max(lo, cr);
    }
    double v = (0 < lo ? lo : hi < 0 ? hi : 0);
    P c = (a + b) / 2 + (b - a).perp() * v / (b - a).dist2()
        ↪;
    return {(a - c).dist2(), c};
}

pair<double, P> mec(vector<P>& S, P a, int n) {
    random_shuffle(S.begin(), S.begin() + n);
    P b = S[0], c = (a + b) / 2;
    double r = (a - c).dist2();
    rep(i,1,n) if ((S[i] - c).dist2() > r * (1 + 1e-8)) {
        tie(r,c) = (n == sz(S) ?
            mec(S, S[i], i) : mec2(S, a, S[i], i));
    }
    return {r, c};
}

pair<double, P> enclosingCircle(vector<P> S) {
    assert(!S.empty()); auto r = mec(S, S[0], sz(S));
    return {sqrt(r.first), r.second};
} // hash-cpp-all = 9bf427c9626a72f805196e0b7075bda2
```

## 7.3 Polygons

### insidePolygon.h

**Description:** Returns true if p lies within the polygon described by the points between iterators begin and end. If strict false is returned when p is on the edge of the polygon. Answer is calculated by counting the number of intersections between the polygon and a line going from p to infinity in the positive x-direction. The algorithm uses products in intermediate steps so watch out for overflow. If points within epsilon from an edge should be considered as on the edge replace the line "if (onSegment..." with the comment below it (this will cause overflow for int and long long).

**Usage:** typedef Point<int> pi; vector<pi> v; v.push\_back(pi(4,4)); v.push\_back(pi(1,2)); v.push\_back(pi(2,1)); bool in = insidePolygon(v.begin(),v.end(), pi(3,4), false);

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

```
"Point.h", "onSegment.h", "SegmentDistance.h"
14 lines

template<class It, class P>
bool insidePolygon(It begin, It end, const P& p,
    bool strict = true) {
    int n = 0; //number of isects with line from p to (inf,p
        ↪.y)
    for (It i = begin, j = end-1; i != end; j = i++) {
        //if p is on edge of polygon
        if (onSegment(*i, *j, p)) return !strict;
        //or: if (segDist(*i, *j, p) <= epsilon) return !
            ↪strict;
        //increment n if segment intersects line from p
        n += (max(i->y,j->y) > p.y && min(i->y,j->y) <= p.y &&
            ((*j-*i).cross(p-*i) > 0) == (i->y <= p.y));
    }
    return n&1; //inside if odd number of intersections
} // hash-cpp-all = 0cadec56a74f257b8d1b25f56ba7ebad
```

### PolygonArea.h

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

```
"Point.h"
6 lines

template<class T>
T polygonArea2(vector<Point<T>>& v) {
    T a = v.back().cross(v[0]);
    rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
    return a;
} // hash-cpp-all = f123003799a972c1292eb0d8af7e37da
```

### PolygonCenter.h

**Description:** Returns the center of mass for a polygon.

```
"Point.h"
10 lines

typedef Point<double> P;
Point<double> polygonCenter(vector<P>& v) {
    auto i = v.begin(), end = v.end(), j = end-1;
    Point<double> res{0,0}; double A = 0;
    for (; i != end; j=i++) {
        res = res + (*i * j) * j->cross(*i);
        A += j->cross(*i);
    }
    return res / A / 3;
} // hash-cpp-all = d210bd2372832f7d074894d904e548ab
```

### PolygonCut.h

**Description:**

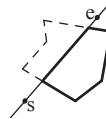
Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.

**Usage:** vector<P> p = ...; p = polygonCut(p, P(0,0), P(1,0));

```
"Point.h", "lineIntersection.h"
15 lines

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

```



```
        lineIntersection(s, e, cur, prev, res.back());
    }
    if (side)
        res.push_back(cur);
}
return res;
} // hash-cpp-all = acf5106be46aa8f6f5d7a8d0ffdaae3c
```

### ConvexHull.h

**Description:**

Returns a vector of indices of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.

**Usage:** vector<P> ps, hull;

trav(i, convexHull(ps)) hull.push\_back(ps[i]);

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

```
"Point.h"
20 lines

typedef Point<ll> P;
pair<vi, vi> ulHull(const vector<P>& S) {
    vi Q(sz(S)), U, L;
    iota(all(Q), 0);
    sort(all(Q), [&S](int a, int b){ return S[a] < S[b]; });
    trav(it, Q) {
        #define ADDP(C, cmp) while (sz(C) > 1 && S[C[sz(C)-2]].
            ↪cross(\
                S[it], S[C.back()]) cmp 0) C.pop_back(); C.push_back(it)
            ↪;
        ADDP(U, <=); ADDP(L, >=);
    }
    return {U, L};
}
```

```
vi convexHull(const vector<P>& S) {
    vi u, l; tie(u, l) = ulHull(S);
    if (sz(S) <= 1) return u;
    if (S[u[0]] == S[u[l]]) return {0};
    l.insert(l.end(), u.rbegin()+1, u.rend()-1);
    return l;
} // hash-cpp-all = cc37090b10272c6be42e22ac943f42c
```

### PolygonDiameter.h

**Description:** Calculates the max squared distance of a set of points.

```
"ConvexHull.h"
19 lines

vector<pii> antipodal(const vector<P>& S, vi& U, vi& L) {
    vector<pii> ret;
    int i = 0, j = sz(L) - 1;
    while (i < sz(U) - 1 || j > 0) {
        ret.emplace_back(U[i], L[j]);
        if (j == 0 || (i != sz(U)-1 && (S[L[j]] - S[L[j-1]]).
            ↪cross(S[U[i+1]] - S[U[i]]) > 0)) ++i;
        else --j;
    }
    return ret;
}
```

```
pii polygonDiameter(const vector<P>& S) {
    vi U, L; tie(U, L) = ulHull(S);
    pair<ll, pii> ans;
    trav(x, antipodal(S, U, L))
        ans = max(ans, {(S[x.first] - S[x.second]).dist2(), x
            ↪});
    return ans.second;
} // hash-cpp-all = 5596d386362874d2ebcf13cdb142574d
```





### PointInsideHull.h

**Description:** Determine whether a point  $t$  lies inside a given polygon (counter-clockwise order). The polygon must be such that every point on the circumference is visible from the first point in the vector. It returns 0 for points outside, 1 for points on the circumference, and 2 for points inside.

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

```
"Point.h", "sideOf.h", "onSegment.h" 22 lines
typedef Point<ll> P;
int insideHull2(const vector<P>& H, int L, int R, const P&
    ↪ p) {
    int len = R - L;
    if (len == 2) {
        int sa = sideOf(H[0], H[L], p);
        int sb = sideOf(H[L], H[L+1], p);
        int sc = sideOf(H[L+1], H[0], p);
        if (sa < 0 || sb < 0 || sc < 0) return 0;
        if (sb==0 || (sa==0 && L == 1) || (sc == 0 && R == sz(
            ↪ H)))
            return 1;
        return 2;
    }
    int mid = L + len / 2;
    if (sideOf(H[0], H[mid], p) >= 0)
        return insideHull2(H, mid, R, p);
    return insideHull2(H, L, mid+1, p);
}

int insideHull(const vector<P>& hull, const P& p) {
    if (sz(hull) < 3) return onSegment(hull[0], hull.back(),
    ↪ p);
    else return insideHull2(hull, 1, sz(hull), p);
} // hash-cpp-all = 1c16dba23109ced37b95769a3f1d19b7
```

### LineHullIntersection.h

**Description:** Line-convex polygon intersection. The polygon must be ccw and have no colinear points. isct(a, b) returns a pair describing the intersection of a line with the polygon:  $\bullet (-1, -1)$  if no collision,  $\bullet (i, -1)$  if touching the corner  $i$ ,  $\bullet (i, i)$  if along side  $(i, i+1)$ ,  $\bullet (i, j)$  if crossing sides  $(i, i+1)$  and  $(j, j+1)$ . In the last case, if a corner  $i$  is crossed, this is treated as happening on side  $(i, i+1)$ . The points are returned in the same order as the line hits the polygon.

**Time:**  $\mathcal{O}(N + Q \log n)$

```
"Point.h" 63 lines
ll sgn(ll a) { return (a > 0) - (a < 0); }
typedef Point<ll> P;
struct HullIntersection {
    int N;
    vector<P> p;
    vector<pair<P, int>> a;

    HullIntersection(const vector<P>& ps) : N(sz(ps)), p(ps)
    ↪ {
        p.insert(p.end(), all(ps));
        int b = 0;
        rep(i, 1, N) if (P{p[i].y, p[i].x} < P{p[b].y, p[b].x}) b
            ↪ = i;
        rep(i, 0, N) {
            int f = (i + b) % N;
            a.emplace_back(p[f+1] - p[f], f);
        }

        int qd(P p) {
            return (p.y < 0) ? (p.x >= 0) + 2
                : (p.x <= 0) * (1 + (p.y <= 0));
        }
    }
};
```

```
int bs(P dir) {
    int lo = -1, hi = N;
    while (hi - lo > 1) {
        int mid = (lo + hi) / 2;
        if (make_pair(qd(dir), dir.y * a[mid].first.x) <
            make_pair(qd(a[mid].first), dir.x * a[mid].first.y
            ↪ ))
            hi = mid;
        else lo = mid;
    }
    return a[hi%N].second;
}

bool isign(P a, P b, int x, int y, int s) {
    return sgn(a.cross(p[x], b)) * sgn(a.cross(p[y], b))
    ↪ == s;
}

int bs2(int lo, int hi, P a, P b) {
    int L = lo;
    if (hi < lo) hi += N;
    while (hi - lo > 1) {
        int mid = (lo + hi) / 2;
        if (isign(a, b, mid, L, -1)) hi = mid;
        else lo = mid;
    }
    return lo;
}

pii isct(P a, P b) {
    int f = bs(a - b), j = bs(b - a);
    if (isign(a, b, f, j, 1)) return {-1, -1};
    int x = bs2(f, j, a, b)%N,
        y = bs2(j, f, a, b)%N;
    if (a.cross(p[x], b) == 0 &&
        a.cross(p[x+1], b) == 0) return {x, x};
    if (a.cross(p[y], b) == 0 &&
        a.cross(p[y+1], b) == 0) return {y, y};
    if (a.cross(p[f], b) == 0) return {f, -1};
    if (a.cross(p[j], b) == 0) return {j, -1};
    return {x, y};
}

}; // hash-cpp-all = 79decd52fd801714ccebbaa6ab36151e
```

## 7.4 Misc. Point Set Problems

### closestPair.h

**Description:**  $i1, i2$  are the indices to the closest pair of points in the point vector  $p$  after the call. The distance is returned.

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

```
"Point.h" 58 lines
template<class It>
bool it_less(const It& i, const It& j) { return *i < *j; }
template<class It>
bool y_it_less(const It& i, const It& j) { return i->y < j->
    ↪ y; }

template<class It, class IIt> /* IIt = vector<It>::
    ↪ iterator */
double cp_sub(IIt ya, IIt yaend, IIt xa, It &i1, It &i2) {
    typedef typename iterator_traits<It>::value_type P;
    int n = yaend-ya, split = n/2;
    if (n <= 3) { // base case
        double a = (*xa[1]-*xa[0]).dist(), b = 1e50, c = 1e50;
        if (n==3) b = (*xa[2]-*xa[0]).dist(), c = (*xa[2]-*xa[1]).
            ↪ dist();
    }
```

```
if (a <= b) { i1 = xa[1];
    if (a <= c) return i2 = xa[0], a;
    else return i2 = xa[2], c;
} else { i1 = xa[2];
    if (b <= c) return i2 = xa[0], b;
    else return i2 = xa[1], c;
} }
vector<It> ly, ry, strip;
P splitp = *xa[split];
double splitx = splitp.x;
for (IIt i = ya; i != yaend; ++i) { // Divide
    if (*i != xa[split] && (*i-splitp).dist2() < 1e-12)
        return i1 = *i, i2 = xa[split], 0; // nasty special
    ↪ case!
    if ((*i < splitp) ly.push_back(*i);
    else ry.push_back(*i);
} // assert((signed)lefty.size() == split)
It j1, j2; // Conquer
double a = cp_sub(ly.begin(), ly.end(), xa, i1, i2);
double b = cp_sub(ry.begin(), ry.end(), xa+split, j1, j2
    ↪ );
if (b < a) a = b, i1 = j1, i2 = j2;
double a2 = a*a;
for (IIt i = ya; i != yaend; ++i) { // Create strip (y-
    ↪ sorted)
    double x = (*i)->x;
    if (x >= splitx-a && x <= splitx+a) strip.push_back(*i
    ↪ );
}
for (IIt i = strip.begin(); i != strip.end(); ++i) {
    const P &p1 = *i;
    for (IIt j = i+1; j != strip.end(); ++j) {
        const P &p2 = *j;
        if (p2.y-p1.y > a) break;
        double d2 = (p2-p1).dist2();
        if (d2 < a2) i1 = *i, i2 = *j, a2 = d2;
    }
}
return sqrt(a2);
}
```

```
template<class It> // It is random access iterators of
    ↪ point<T>
double closestpair(It begin, It end, It &i1, It &i2) {
    vector<It> xa, ya;
    assert(end-begin >= 2);
    for (It i = begin; i != end; ++i)
        xa.push_back(i), ya.push_back(i);
    sort(xa.begin(), xa.end(), it_less<It>);
    sort(ya.begin(), ya.end(), y_it_less<It>);
    return cp_sub(ya.begin(), ya.end(), xa.begin(), i1, i2);
} // hash-cpp-all = 42735b8e08701a3b73504ac0690e31df
```

### kdTree.h

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

```
"Point.h" 63 lines
typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();

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

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



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

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

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

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

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

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

    // find nearest point to a point, and its squared
    ↪distance
    // (requires an arbitrary operator< for Point)
    pair<T, P> nearest(const P& p) {
        return search(root, p);
    }
}; // hash-cpp-all = bac5b0409b201c3b040301344a40dc31
```

DelaunayTriangulation.h

**Description:** Computes the Delaunay triangulation of a set of points. Each circumcircle contains none of the input points. If any three points are colinear or any four are on the same circle, behavior is undefined. **Time:**  $\mathcal{O}(n^2)$

```
"Point.h", "3dHull.h" 10 lines
template<class P, class F>
void delaunay(vector<P>&& ps, F trifun) {
```

```
    if (sz(ps) == 3) { int d = (ps[0].cross(ps[1], ps[2]) <
        ↪0);
        trifun(0,1+d,2-d); }
    vector<P3> p3;
    trav(p, ps) p3.emplace_back(p.x, p.y, p.dist2());
    if (sz(ps) > 3) trav(t, hull3d(p3)) if ((p3[t.b]-p3[t.a
        ↪]).
        cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
        trifun(t.a, t.c, t.b);
} // hash-cpp-all = d173fc69317d23d87be99189086af6d2
```

FastDelaunay.h

**Description:** Fast Delaunay triangulation. There must be no duplicate points. If all points are on a line, no triangles will be returned. Should work for doubles as well, though there may be precision issues in 'circ'. Returns triangles in order {t[0][0], t[0][1], t[0][2], t[1][0], ...}, all counter-clockwise. **Time:**  $\mathcal{O}(n \log n)$

```
"Point.h" 90 lines
```

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

```
struct Quad {
    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 rot->r()->o->rot; }
};
```

```
bool circ(P p, P a, P b, P c) { // is p in the
    ↪circumcircle?
    lll p2 = p.dist2(), A = a.dist2()-p2,
        B = b.dist2()-p2, C = c.dist2()-p2;
    return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B
        ↪> 0;
```

```
Q makeEdge(P orig, P dest) {
    Q q0 = new Quad{0,0,0,orig}, q1 = new Quad{0,0,0,arb},
        q2 = new Quad{0,0,0,dest}, q3 = new Quad{0,0,0,arb};
    q0->o = q0; q2->o = q2; // 0-0, 2-2
    q1->o = q3; q3->o = q1; // 1-3, 3-1
    q0->rot = q1; q1->rot = q2;
    q2->rot = q3; q3->rot = q0;
    return q0;
}
```

```
void splice(Q a, Q b) {
    swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
```

```
Q connect(Q a, Q b) {
    Q q = makeEdge(a->F(), b->p);
    splice(q, a->next());
    splice(q->r(), b);
    return q;
}
```

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

```
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r()
        ↪};
}
```

```
#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
    Q A, B, ra, rb;
    int half = (sz(s) + 1) / 2;
    tie(ra, A) = rec({s.begin(), s.begin() + half});
    tie(B, rb) = rec({s.begin() + half, s.end()});
    while ((B->p.cross(H(A)) < 0 && (A = A->next()) ||
        (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
    Q base = connect(B->r(), A);
    if (A->p == ra->p) ra = base->r();
    if (B->p == rb->p) rb = base;
```

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

```
vector<P> triangulate(vector<P> pts) {
    sort(all(pts)); assert(unique(all(pts)) == pts.end());
    if (sz(pts) < 2) return {};
    Q e = rec(pts).first;
    vector<Q> q = {e};
    int qi = 0;
    while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
    #define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->
        ↪p); \
        q.push_back(c->r()); c = c->next(); } while (c != e); }
    ADD; pts.clear();
    while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
    return pts;
} // hash-cpp-all = 9f490f2e6d7aa19210d75ee72a867313
```

7.5 3D

PolyhedronVolume.h

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

```
6 lines
```

```
template<class V, class L>
double signed_poly_volume(const V& p, const L& trilst) {
    double v = 0;
    trav(i, trilst) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
    return v / 6;
} // hash-cpp-all = 1ec4d393ab307cedc3866534eaa83a0e
```

Point3D.h

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

```
32 lines
```

```
template<class T> struct Point3D {
    typedef Point3D P;
```

```

typedef const P& R;
T x, y, z;
explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z)
    ↪ {}
bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z); }
bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z); }
P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z);
    ↪ }
P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z);
    ↪ }
P operator*(T d) const { return P(x*d, y*d, z*d); }
P operator/(T d) const { return P(x/d, y/d, z/d); }
T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
}
T dist2() const { return x*x + y*y + z*z; }
double dist() const { return sqrt((double)dist2()); }
//Azimuthal angle (longitude) to x-axis in interval [-pi
    ↪ pi]
double phi() const { return atan2(y, x); }
//Zenith angle (latitude) to the z-axis in interval [0,
    ↪ pi]
double theta() const { return atan2(sqrt(x*x+y*y),z); }
P unit() const { return *this/(T)dist(); } //makes dist
    ↪ ()=1
//returns unit vector normal to *this and p
P normal(P p) const { return cross(p).unit(); }
//returns point rotated 'angle' radians ccw around axis
P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit
        ↪ ();
    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
}
}; // hash-cpp-all = 8058aeda36daf3cba079c7bb0b43dcea

```

### 3dHull.h

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

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

"Point3D.h" 49 lines

```
typedef Point3D<double> P3;
```

```

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

```

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

```

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

```

```

    FS.push_back(f);
};
rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);

rep(i,4,sz(A)) {
    rep(j,0,sz(FS)) {
        F f = FS[j];
        if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
            E(a,b).rem(f.c);
            E(a,c).rem(f.b);
            E(b,c).rem(f.a);
            swap(FS[j-], FS.back());
            FS.pop_back();
        }
    }
    int nw = sz(FS);
    rep(j,0,nw) {
        F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i,
        ↪ f.c);
        C(a, b, c); C(a, c, b); C(b, c, a);
    }
    trav(it, FS) if ((A[it.b] - A[it.a]).cross(
        A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
    return FS;
}; // hash-cpp-all = bc89cf3055f5f62e7f5dec78d412986c

```

### sphericalDistance.h

**Description:** Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 ( $\phi_1$ ) and f2 ( $\phi_2$ ) from x axis and zenith angles (latitude) t1 ( $\theta_1$ ) and t2 ( $\theta_2$ ) from z axis. All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx\*radius is then the difference between the two points in the x direction and d\*radius is the total distance between the points.

8 lines

```

double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
    double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
    double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
    double dz = cos(t2) - cos(t1);
    double d = sqrt(dx*dx + dy*dy + dz*dz);
    return radius*2*asin(d/2);
} // hash-cpp-all = 611f0797307c583c66413c2dd5b3ba28

```

## Strings (8)

### KMP.h

**Description:** pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.

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

16 lines

```

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

```

```

vi match(const string& s, const string& pat) {
    vi p = pi(pat + '\0' + s), res;
    rep(i,sz(p)-sz(s),sz(p))
        if (p[i] == sz(pat)) res.push_back(i - 2 * sz(pat));
    return res;
} // hash-cpp-all = d4375c5f06b664278b2df96136a588d9

```

### extended-KMP.h

**Description:** extended KMP S[i] stores the maximum common prefix between s[i] and t; T[i] stores the maximum common prefix between t[i] and t for i>0;

33 lines

```

int S[N], T[N];

void extKMP(const string&s, const string &t) {
    int m = t.size();
    T[0] = 0;
    int maT = 0;
    for (int i = 1; i < m; i++) {
        if (maT + T[maT] >= i) {
            T[i] = min(T[i - maT], maT + T[maT] - i);
        } else {
            T[i] = 0;
        }
        while (T[i] + i < m && t[T[i]] == t[T[i] + i])
            T[i]++;
        if (i + T[i] > maT + T[maT])
            maT = i;
    }
    int maS = 0;
    int n = s.size();
    for (int i = 0; i < n; i++) {
        if (maS + S[maS] >= i) {
            S[i] = min(T[i - maS], maS + S[maS] - i);
        } else {
            S[i] = 0;
        }
        while (S[i] < m && i + S[i] < n && t[S[i]] == s[S[i] +
            ↪ i])
            S[i]++;
        if (i + S[i] > maS + S[maS])
            maS = i;
    }
} // hash-cpp-all = 40cf01c6dd1669aaac6106a10af35b35

```

### Manacher.h

**Description:** For each position in a string, computes p[0][i] = half length of longest even palindrome around pos i, p[1][i] = longest odd (half rounded down).

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

11 lines

```

void manacher(const string& s) {
    int n = sz(s);
    vi p[2] = {vi(n+1), vi(n)};
    rep(z,0,2) for (int i=0,l=0,r=0; i < n; i++) {
        int t = r-i+!z;
        if (i<r) p[z][i] = min(t, p[z][l+t]);
        int L = i-p[z][i], R = i+p[z][i]-!z;
        while (L>=l && R+1<n && s[L-1] == s[R+1])
            p[z][i]++, L--, R++;
        if (R>r) l=L, r=R;
    } // hash-cpp-all = d9436881723eb8d866ac15aa011523db
}

```

## MinRotation.h

**Description:** Finds the lexicographically smallest rotation of a string.

**Usage:** rotate(v.begin(), v.begin()+min\_rotation(v), v.end());

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

8 lines

```
int min_rotation(string s) {
    int a=0, N=sz(s); s += s;
    rep(b,0,N) rep(i,0,N) {
        if (a+i == b || s[a+i] < s[b+i]) { b += max(0, i-1);
            ↪break; }
        if (s[a+i] > s[b+i]) { a = b; break; }
    }
    return a;
} // hash-cpp-all = 358164768a20176868eba20757681e19
```

## string-sa+lcp.cpp

**Description:** SA + LCP

**Usage:** da(str, sa, strlen(str)+1, 256);

calheight(str, sa, strlen(str));

31 lines

```
int wa[maxn],wb[maxn],wv[maxn],ws[maxn];
int cmp(int *r,int a,int b,int l) { // hash-cpp-1
    return r[a]==r[b]&&r[a+l]==r[b+l];
}
void da(int *r,int *sa,int n,int m) {
    int i,j,p,*x=wa,*y=wb,*t;
    for(i=0;i<m;i++) ws[i]=0;
    for(i=0;i<n;i++) ws[x[i]=r[i]]++;
    for(i=1;i<m;i++) ws[i]+=ws[i-1];
    for(i=n-1;i>=0;i--) sa[--ws[x[i]]]=i;
    for(j=1,p=1;p<n;j*=2,m=p) {
        for(p=0,i=n-j;i<n;i++) y[p++]=i;
        for(i=0;i<n;i++)
            if(sa[i]>=j) y[p++]=sa[i]-j;
        for(i=0;i<n;i++) wv[i]=x[y[i]];
        for(i=0;i<m;i++) ws[i]=0;
        for(i=0;i<n;i++) ws[wv[i]]++;
        for(i=1;i<m;i++) ws[i]+=ws[i-1];
        for(i=n-1;i>=0;i--) sa[--ws[wv[i]]]=y[i];
        for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
            x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
    }
} // hash-cpp-1 = ce2b3946ed8dab557ac57271351047a5
//height[i]: lcp(sa[i],sa[i-1])
int rank[maxn],height[maxn];
void calheight(int *r,int *sa,int n) { // hash-cpp-2
    int i,j,k=0;
    for(i=1;i<=n;i++) rank[sa[i]]=i;
    for(i=0;i<n;height[rank[i++]]=k)
        for(k?k--:0,j=sa[rank[i]-1]; r[i+k]==r[j+k]; k++);
} // hash-cpp-2 = 29b5645cc1aca9a59ff90adec1d537e5
```

## string-SAM.cpp

**Description:** Suffix Automaton (SAM)

37 lines

```
int n,i,init,L,len,ll,q,h,ch,p,last[1700000],n1[1700000],
    ↪du[1700000],s[1700000],fa[800001],l[1700000],son
    ↪[1700000][3],par[1700000];
char S[8000001],k;
long long ans,sum[1600001];
void ins(int p,int ss,int k)
{
    int np=++len,q,nq;
    l[np]=l[p]+1;
    s[np]=l;
    while (p&&!son[p][k]) son[p][k]=np,p=par[p];
    if (!p) par[np]=1;
```

```
else {
    q=son[p][k];
    if (l[p]+1==l[q]) par[np]=q;
    else {
        nq=++len;
        l[nq]=l[p]+1;
        s[nq]=0;
        memset(son[nq], son[q], sizeof son[q]);
        par[nq]=par[q];
        par[q]=nq;
        par[np]=nq;
        while (p&&son[p][k]==q) son[p][k]=nq,p=par[p];
    }
    last[ss]=np;
}
int main()
{
    read(n);
    last[1]=init=len=1;
    for (i=2;i<=n;i++)
    {
        read(fa[i]);
        for (k=getchar();k<=32;k=getchar());
        ins(last[fa[i]],i,k-'a');
    }
} // hash-cpp-all = 6de1ae4723820c6fbc161c9e51574990
```

## string-dc3.cpp

**Description:** Linear-time SA+LCP+Tree

108 lines

```
const int N=1000010;
char s[N];
int *h;

namespace SuffixArray {

const int N=1000010;

int sa[N],rk[N],ht[N];
bool t[N<<1];

bool islms(const int i,const bool *t) { // hash-cpp-1
    return i>0&&t[i]&&t[i-1];
} // hash-cpp-1 = 5ca6c1c830ec37aed73de79822fb6c8e

template<class T>
inline void sort(T s,int *sa,const int len,const int sz,
    ↪const int sigma,
    bool *t,int *b,int *cb,int *p) { // hash-cpp-2
    memset(b,0,sizeof(int)*sigma);
    memset(sa,-1,sizeof(int)*len);
    rep(i,0,len) b[(int)s[i]]++;
    cb[0]=b[0];
    rep(i,1,sigma) cb[i]=cb[i-1]+b[i];
    per(i,0,sz) sa[--cb[(int)s[p[i]]]]=p[i];
    rep(i,1,sigma) cb[i]=cb[i-1]+b[i-1];
    rep(i,0,len) if (sa[i]>0&&t[sa[i]-1]) sa[cb[(int)s[sa[i]
        ↪]-1]]+=sa[i]-1;
    cb[0]=b[0];
    rep(i,1,sigma) cb[i]=cb[i-1]+b[i];
    per(i,0,len) if (sa[i]>0&&t[sa[i]-1]) sa[--cb[(int)s[sa[i]
        ↪]-1]]+=sa[i]-1;
} // hash-cpp-2 = 88f5a486e24125b363a4fdb671376629

template<class T>
```

```
inline void sais(T s,int *sa,const int len,bool *t,int *b,
    ↪int *bl,
    const int sigma) { // hash-cpp-3
    int p=-1,*cb=b+sigma;
    t[len-1]=1;
    per(i,0,len-1) t[i]=s[i]<s[i+1]||(s[i]==s[i+1]&&t[i+1]);
    int sz=0,cnt=0;
    rep(i,1,len) if (t[i]&&!t[i-1]) bl[sz++]=i;
    sort(s,sa,len,sz,sigma,t,b,cb,bl);
    sz=0;
    rep(i,0,len) if (islms(sa[i],t)) sa[sz++]=sa[i];
    rep(i,sz,len) sa[i]=-1;
    rep(i,0,sz) {
        int x=sa[i];
        rep(j,0,len) {
            if (p==-1||s[x+j]!=s[p+j]||t[x+j]!=t[p+j]) {
                cnt++; p=x;
                break;
            } else if (j>0&&(islms(x+j,t)||islms(p+j,t))) {
                break;
            }
        }
        sa[sz+(x>=1)]=cnt-1;
    }
    for (int i=len-1,j=len-1;i>=sz;i--) if (sa[i]>=0) sa[j
        ↪--]=sa[i];
    int *sl=sa+len-sz,*b2=bl+sz;
    if (cnt<sz) sais(sl,sa,sz,t+len,b,bl+sz,cnt);
    else rep(i,0,sz) sa[sl[i]]=i;
    rep(i,0,sz) b2[i]=bl[sa[i]];
    sort(s,sa,len,sz,sigma,t,b,cb,b2);
} // hash-cpp-3 = 06c63b43c0de339e2fbc000178dc4084
```

```
template<class T>
inline void getHeight(T s,int n) { // hash-cpp-4
    rep(i,1,n+1) rk[sa[i]]=i;
    int j=0,k=0;
    for (int i=0;i<n;ht[rk[i++]]=k)
        for (k?k--:0,j=sa[rk[i]-1];s[i+k]==s[j+k];k++);
} // hash-cpp-4 = d171edf9c242a8cdb65bbca53aab75dd
```

```
template<class T>
inline void init(T s,const int len,const int sigma) { //
    ↪hash-cpp-5
    sais(s,sa,len,t,rk,ht,sigma);
} // hash-cpp-5 = e90e73297525a28516de9c2d1653b256
```

```
inline void solve(char *s,int len) {
    init(s,len+1,124);
    getHeight(s,len);
}
} // namespace SuffixArray
```

int n;

int stk[N],top,a[N],l[N],r[N],sz[N],par[N];

```
void build() { // hash-cpp-6
    int top=0;
    h=SuffixArray::ht+1;
    rep(i,1,n) l[i]=r[i]=par[i]=0;
    rep(i,1,n) {
        int k=top;
        while (k>0&&h[stk[k-1]]>h[i]) --k;
        if (k) r[stk[k-1]]=i;
        if (k<top) l[i]=stk[k];
        stk[k++]=i;
        top=k;
```

```

}
int t=0,rt=stk[0];
int *q=stk;
q[t++]=rt;
rep(i,0,t) {
    int u=q[i]; sz[u]=1;
    if (l[u]) q[t++]=l[u],par[l[u]]=u;
    if (r[u]) q[t++]=r[u],par[r[u]]=u;
}
} // hash-cpp-6 = 496cf09518bc84e0fc8000c0f7adf03d

```

## SuffixTree.h

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

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

50 lines

```

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

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

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

```

// example: find longest common substring (uses ALPHA = 28)

```

pii best;
int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node])
        <=> 0;
    rep(c,0,ALPHA) if (t[node][c] != -1)
        mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
        best = max(best, {len, r[node] - len});
    return mask;
}

```

```

}
static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' +
        <=> 2));
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
}
}; // hash-cpp-all = aae0b8bb2efccb834b9a439b63d92f53

```

## Hashing.h

**Description:** Various self-explanatory methods for string hashing, 45 lines

```

// Arithmetic mod 2^64-1. 2x slower than mod 2^64 and more
// code, but works on evil test data (e.g. Thue-Morse,
    <=> where
// ABBA... and BAAB... of length 2^10 hash the same mod
    <=> 2^64).
// "typedef ull H;" instead if you think test data is
    <=> random,
// or work mod 10^9+7 if the Birthday paradox is not a
    <=> problem.
struct H {
    typedef uint64_t ull;
    ull x; H(ull x=0) : x(x) {}
    #define OP(O,A,B) H operator O(H o) { ull r = x; asm \
        (A "addq %%rdx, %0\n adcq $0,%0" : "+a"(r) : B); return
        <=> r; }
    OP(+, "d"(o.x)) OP(*, "mul %l\n", "r"(o.x) : "rdx")
    H operator-(H o) { return *this + ~o.x; }
    ull get() const { return x + !~x; }
    bool operator==(H o) const { return get() == o.get(); }
    bool operator<(H o) const { return get() < o.get(); }
};
static const H C = (1ll)1e11+3; // (order ~ 3e9; random
    <=> also ok)

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

```

```

vector<H> getHashes(string& str, int length) {
    if (sz(str) < length) return {};
    H h = 0, pw = 1;
    rep(i,0,length)
        h = h * C + str[i], pw = pw * C;
    vector<H> ret = {h};
    rep(i,length,sz(str)) {
        ret.push_back(h = h * C + str[i] - pw * str[i-length])
            <=> ;
    }
    return ret;
}

```

```

H hashString(string& s) { H h{}; trav(c,s) h=h*C+c; return
    <=> h; }
// hash-cpp-all = ca8497b9a5d82dca62bc1210bb2b3c4c

```

## AhoCorasick.h

**Description:** Aho-Corasick tree is used for multiple pattern matching. Initialize the tree with create(patterns). find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll(., word) finds all words (up to  $N\sqrt{N}$  many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. To find the longest words that start at each position, reverse all input.

**Time:** Function create is  $\mathcal{O}(26N)$  where  $N$  is the sum of length of patterns. find is  $\mathcal{O}(M)$  where  $M$  is the length of the word. findAll is  $\mathcal{O}(NM)$ .

67 lines

```

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

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

```

```
vector<vi> findAll(vector<string>& pat, string word) {
    vi r = find(word);
    vector<vi> res(sz(word));
    rep(i,0,sz(word)) {
        int ind = r[i];
        while (ind != -1) {
            res[i - sz(pat[ind]) + 1].push_back(ind);
            ind = backp[ind];
        }
    }
    return res;
}; // hash-cpp-all = 716ac4cbf4109c8b0ba0795702a8bfe1
```

## Various (9)

### 9.1 Intervals

#### IntervalContainer.h

**Description:** Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).

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

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

```
void removeInterval(set<pii>& is, int L, int R) {
    if (L == R) return;
    auto it = addInterval(is, L, R);
    auto r2 = it->second;
    if (it->first == L) is.erase(it);
    else (int&)it->second = L;
    if (R != r2) is.emplace(R, r2);
}; // hash-cpp-all = edce47664ed34a95a513b699a9b796e2
```

#### IntervalCover.h

**Description:** Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add || R.empty(). Returns empty set on failure (or if G is empty).

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

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

```
            mx = max(mx, make_pair(I[S[at]].second, S[at]));
            at++;
        }
        if (mx.second == -1) return {};
        cur = mx.first;
        R.push_back(mx.second);
    }
    return R;
}; // hash-cpp-all = 9e9d8de75aadfadfe513b17b1c746dbe
```

#### ConstantIntervals.h

**Description:** Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a call-back g for each such interval.

**Usage:** constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});

**Time:**  $\mathcal{O}(k \log \frac{n}{k})$

```
19 lines
template<class F, class G, class T>
void rec(int from, int to, F f, G g, int& i, T& p, T q) {
    if (p == q) return;
    if (from == to) {
        g(i, to, p);
        i = to; p = q;
    } else {
        int mid = (from + to) >> 1;
        rec(from, mid, f, g, i, p, f(mid));
        rec(mid+1, to, f, g, i, p, q);
    }
}
template<class F, class G>
void constantIntervals(int from, int to, F f, G g) {
    if (to <= from) return;
    int i = from; auto p = f(i), q = f(to-1);
    rec(from, to-1, f, g, i, p, q);
    g(i, to, q);
}; // hash-cpp-all = 792e7d94c54ab04f9efdb6834b12fec
```

### 9.2 Misc. algorithms

#### Karatsuba.h

**Description:** Faster-than-naive convolution of two sequences:  $c[x] = \sum a[i]b[x-i]$ . Uses the identity  $(aX+b)(cX+d) = acX^2 + bd + ((a+c)(b+d) - ac - bd)X$ . Doesn't handle sequences of very different length well. See also FFT, under the Numerical chapter.

**Time:**  $\mathcal{O}(N^{1.6})$

```
1 lines
// hash-cpp-all = d41d8cd98f00b204e9800998ecf8427e
```

### 9.3 Dynamic programming

#### DivideAndConquerDP.h

**Description:** Given  $a[i] = \min_{lo(i) \leq k < hi(i)} (f(i, k))$  where the (minimal) optimal  $k$  increases with  $i$ , computes  $a[i]$  for  $i = L..R - 1$ .

**Time:**  $\mathcal{O}((N + (hi - lo)) \log N)$

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

    void rec(int L, int R, int LO, int HI) {
        if (L >= R) return;
        int mid = (L + R) >> 1;
        pair<ll, int> best(LLONG_MAX, LO);
        rep(k, max(LO, lo(mid)), min(HI, hi(mid)))
```

```
        best = min(best, make_pair(f(mid, k), k));
        store(mid, best.second, best.first);
        rec(L, mid, LO, best.second+1);
        rec(mid+1, R, best.second, HI);
    }
    void solve(int L, int R) { rec(L, R, INT_MIN, INT_MAX);
        ↪ }
}; // hash-cpp-all = d38d2b272d60f6d5cead54745a551b99
```

#### KnuthDP.h

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

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

```
1 lines
// hash-cpp-all = d41d8cd98f00b204e9800998ecf8427e
```

### 9.4 Debugging tricks

- signal(SIGSEGV, [](int) { \_Exit(0); }) ; converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions). \_GLIBCXX\_DEBUG 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).

### 9.5 Optimization tricks

#### 9.5.1 Bit hacks

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

#### 9.5.2 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.
- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).

BumpAllocator.h

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

```
9 lines
// Either globally or in a single class:
static char buf[450 << 20];
void* operator new(size_t s) {
    static size_t i = sizeof buf;
    assert(s < i);
    return (void*)&buf[i -= s];
}
void operator delete(void*) {}
// hash-cpp-all = 745db225903de8f3cdfa051660956100
```

SmallPtr.h

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

```
10 lines
"BumpAllocator.h"
template<class T> struct ptr {
    unsigned ind;
    ptr(T* p = 0) : ind(p ? unsigned((char*)p - buf) : 0) {
        assert(ind < sizeof buf);
    }
    T& operator*() const { return *(T*)(buf + ind); }
    T* operator->() const { return &*this; }
    T& operator[](int a) const { return (&this)[a]; }
    explicit operator bool() const { return ind; }
}; // hash-cpp-all = 2dd6c9773f202bd47422e255099f4829
```

BumpAllocatorSTL.h

**Description:** BumpAllocator for STL containers.

**Usage:** vector<vector<int, small<int>>> ed(N);

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

template<class T> struct small {
    typedef T value_type;
    small() {}
    template<class U> small(const U&) {}
    T* allocate(size_t n) {
        buf_ind -= n * sizeof(T);
        buf_ind &= 0 - alignof(T);
        return (T*)(buf + buf_ind);
    }
    void deallocate(T*, size_t) {}
}; // hash-cpp-all = bb66d4225a1941b85228ee92b9779d4b
```

Unrolling.h

```
6 lines
#define F {...; ++i;}
int i = from;
while (i&3 && i < to) F // for alignment, if needed
while (i + 4 <= to) { F F F F }
while (i < to) F
// hash-cpp-all = 69ac737ad5a50f5688d5720fb6fce39f
```

SIMD.h

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

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

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

// High-level/specific methods:
// load(u)?_si256, store(u)?_si256, setzero_si256,
//   ↳ _mm_malloc
// blendv_(epi8|ps|pd) (z?y:x), movemask_epi8 (hibits of
//   ↳ bytes)
// i32gather_epi32(addr, x, 4): map addr[] over 32-b parts
//   ↳ of x
// sad_epu8: sum of absolute differences of u8, outputs 4
//   ↳ xi64
// maddubs_epil6: dot product of unsigned i7's, outputs 16
//   ↳ xi15
// madd_epil6: dot product of signed i16's, outputs 8xi32
// extractf128_si256(i) (256->128), cvtsil128_si32 (128->
//   ↳ lo32)
// permute2f128_si256(x,x,1) swaps 128-bit lanes
// shuffle_epi32(x, 3*64+2*16+1*4+0) == x for each lane
// shuffle_epi8(x, y) takes a vector instead of an imm

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

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

ll example_filteredDotProduct(int n, short* a, short* b) {
    int i = 0; ll r = 0;
    mi zero = _mm256_setzero_si256(), acc = zero;
    while (i + 16 <= n) {
        mi va = L(a[i]), vb = L(b[i]); i += 16;
        va = _mm256_and_si256(_mm256_cmpgt_epi16(vb, va), va);
        mi vp = _mm256_madd_epil6(va, vb);
        acc = _mm256_add_epi64(_mm256_unpacklo_epi32(vp, zero)
            ↳ ,
            _mm256_add_epi64(acc, _mm256_unpackhi_epi32(vp, zero
                ↳ )));
    }
    union {ll v[4]; mi m;} u; u.m = acc; rep(i,0,4) r += u.v
        ↳ [i];
    for (;i<n;++i) if (a[i] < b[i]) r += a[i]*b[i]; // <-
        ↳ equiv
    return r;
} // hash-cpp-all = 22e525308475fcaf994390db2fedf94
```

Hashmap.h

**Description:** Faster/better hash maps, taken from CF

```
14 lines
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
gp_hash_table<int, int> table;

struct custom_hash {
    size_t operator()(uint64_t x) const {
        x += 48;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
};
gp_hash_table<int, int, custom_hash> safe_table;
// hash-cpp-all = e62eb2668aee2263b6d72043f3652fb2
```

9.6 Other languages

Main.java

**Description:** Basic template/info for Java

```
14 lines
import java.util.*;
import java.math.*;
import java.io.*;
public class Main {
    public static void main(String[] args) throws Exception
        ↳ {
        BufferedReader br = new BufferedReader(new
            ↳ InputStreamReader(System.in));
        PrintStream out = System.out;
        StringTokenizer st = new StringTokenizer(br.readLine()
            ↳ );
        assert st.hasMoreTokens(); // enable with java -ea
            ↳ main
        out.println("v=" + Integer.parseInt(st.nextToken()));
        ArrayList<Integer> a = new ArrayList<>();
        a.add(1234); a.get(0); a.remove(a.size()-1); a.clear()
            ↳ ;
        }
}
```