vimrc

```
"Windows: :e $HOME/_vimrc
"Linux:   :e $HOME/.vimrc

set nocompatible
set number
syntax on
filetype plugin indent on
set bs=indent,eol,start
set et
set sw=4
set ts=4
set hls

nnoremap j gj
nnoremap k gk
nnoremap tn :tabnew<Space>
nnoremap <C-l> gt
nnoremap <C-h> gT
nnoremap <C-m> :make<CR>

"set backspace=indent,eol,start
"set expandtab
"set shiftwidth=4
"set tabstop=4
"set hlsearch
```

Algebra.cc

```
// Throughout all following code, it's assumed that inputs are nonnegative.
// However, a signed type is used for two purposes:
// 1. -1 is used as an error code sometimes.
// 2. Some of these (egcd) actually have negative return values.

typedef signed long long int T;
typedef vector<T> VT;
typedef vector<VT> VVT;

// basic gcd
T gcd( T a, T b ) {
    if( a < 0 ) return gcd(-a,b);
    if( b < 0 ) return gcd(a,-b);
    T c;
    while( b ) { c = a % b; a = b; b = c; }
    return a;
}

// basic lcm
T lcm( T a, T b ) {
    if( a < 0 ) return lcm(-a,b);
    if( b < 0 ) return lcm(a,-b);
    return a/gcd(a,b)*b;  // avoids overflow
}

// returns gcd(a,b), and additionally finds x,y such that gcd(a,b) = ax + by
T egcd( T a, T b, T &x, T &y ) {
    if( a < 0 ) {
        T r = egcd(-a,b,x,y);
        x *= -1;
        return r;
    }
    if( b < 0 ) {
        T r = egcd(a,-b,x,y);
        y *= -1;
        return r;
    }
    T u = y = 0,   v = x = 1;
    while( b ) {
        T q = a/b,     r = a % b;
          a = b,       b = r;
        T m = u,       n = v;
          u = x - q*u, v = y - q*v;
          x = m,       y = n;
    }
    return a;
}

// Compute b so that ab = 1 (mod n).
// Returns n if gcd(a,n) != 1, since no such b exists.
T modinv( T a, T n ) {
    T x, y, g = egcd( a, n, x, y );
    if( g != 1 ) return -1;
    x %= n;
    if( x < 0 ) x += n;
    return x;
}

// Find all solutions to ax = b (mod n),
// and push them onto S.
// Returns the number of solutions.
// Solutions exist iff gcd(a,n) divides b.
// If solutions exist, then there are exactly gcd(a,n) of them.
size_t modsolve( T a, T b, T n, VT &S ) {
    T _1,_2, g = egcd(a,n,_1,_2); // modinv uses egcd already
    if( (b % g) == 0 ) {
        T x = modinv( a/g, n/g );
        x = (x * b/g) % (n/g);
        for( T k = 0; k < g; k++ )
            S.push_back( (x + k*(n/g)) % n );
        return (size_t)g;
    }
    return 0;
}

// Chinese remainder theorem, simple version.
// Given a, b, n, m, find z which simultaneously satisfies
//     z = a (mod m)  and  z = b (mod n).
// This z, when it exists, is unique mod lcm(n,m).
// If such z does not exist, then return -1.
// z exists iff a == b (mod gcd(m,n))
T CRT( T a, T m, T b, T n ) {
    T s, t, g = egcd(m, n, s, t);
    T l = m/g*n, r = a % g;
    if( (b % g) != r ) return -1;
    if( g == 1 ) {
        s = s % l; if( s < 0 ) s += l;
        t = t % l; if( t < 0 ) t += l;
        T r1 = (s * b)   % l, r2 = (t * a)   % l;
          r1 = (r1 * m)  % l, r2 = (r2 * n)  % l;
        return (r1 + r2) % l;
    }
    else {
        return g*CRT(a/g, m/g, b/g, n/g) + r;
    }
}

// Chinese remainder theorem, extended version.
// Given a[K] and n[K], find z so that, for every i,
//     z = a[i] (mod n[i])
// The solution is unique mod lcm( n[i] ) when it exists.
// The existence criteria is just the extended version of what it is above.
T CRT_ext( const VT &a, const VT &n ) {
    T ret = a[0], l = n[0];
    FOR(i,1,a.size()) {
        ret = CRT( ret, l,  a[i], n[i]);
        l = lcm( l, n[i] );
        if( ret == -1 ) return -1;
    }
    return ret;
}

// Compute x and y so that ax + by = c.
// The solution, when it exists, is unique up to the transformation
//     x -> x + kb/g
//     y -> y - ka/g
// for integers k, where g = gcd(a,b).
// The solution exists iff gcd(a,b) divides c.
// The return value is true iff the solution exists.
bool linear_diophantine( T a, T b, T c, T &x, T &y ) {
    T s,t, g = egcd(a,b,s,t);
    if( (c % g) != 0 )
        return false;
    x = c/g*s; y = c/g*t;
    return true;
}

// Given an integer n-by-n matrix A and (positive) integer m,
// compute its determinant mod m.
T integer_det( VVT A, const T M ) {
    const size_t n = A.size();
    FOR(i,0,n) FOR(j,0,n) A[i][j] %= M;
    T det = 1 % M;
    FOR(i,0,n) {
        FOR(j,i+1,n) {
            while( A[j][i] != 0 ) {
                T t = A[i][i] / A[j][i];
                FOR(k,i,n) A[i][k] = (A[i][k] - t*A[j][k]) % M;
                swap( A[i], A[j] );
                det *= -1;
            }
        }
        if( A[i][i] == 0 ) return 0;
        det = (det * A[i][i]) % M;
    }
    if( det < 0 ) det += M;
    return det;
}

T mult_mod(T a, T b, T m) {
    T q;
    T r;
    asm(
            "mulq %3;"
            "divq %4;"
            : "=a"(q), "=d"(r)
            : "a"(a), "rm"(b), "rm"(m));
    return r;
}

/* Computes a^b mod m. Assumes 1 <= m <= 2^62-1 and 0^0=1.
 * The return value will always be in [0, m) regardless of the sign of a.
 */
T pow_mod(T a, T b, T m) {
    if (b == 0) return 1 % m;
    if (b == 1) return a < 0 ? a % m + m : a % m;
    T t = pow_mod(a, b / 2, m);
    t = mult_mod(t, t, m);
    if (b % 2) t = mult_mod(t, a, m);
    return t >= 0 ? t : t + m;
}

/* A deterministic implementation of Miller-Rabin primality test.
 * This implementation is guaranteed to give the correct result for n < 2^64
 * by using a 7-number magic base.
 * Alternatively, the base can be replaced with the first 12 prime numbers
 * (prime numbers <= 37) and still work correctly.
 */
bool is_prime(T n) {
    T small_primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
    for (int i = 0; i < 12; ++i)
        if (n > small_primes[i] && n % small_primes[i] == 0)
            return false;
    T base[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
    T d = n - 1;
    int s = 0;
    for (; d % 2 == 0; d /= 2, ++s);
    for (int i = 0; i < 7; ++i) {
        T a = base[i] % n;
        if (a == 0) continue;
        T t = pow_mod(a, d, n);
        if (t == 1 || t == n - 1) continue;
        bool found = false;
        for (int r = 1; r < s; ++r) {
            t = pow_mod(t, 2, n);
            if (t == n - 1) {
                found = true;
                break;
            }
        }
        if (!found)
            return false;
    }
```

```
    return true;
}
```

FFT.cc

```
// Based on implementation at: http://e-maxx.ru/algo/fft_multiply,
// first seen from:
 ↪   http://serjudging.vanb.org/wp-content/uploads/kinversions_hcheng.cc
// Solves UVa Online Judge "Golf Bot" in 2.35 seconds

// Usage:
// f[0...N-1] and g[0..N-1] are numbers, N is a power of 2.
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise product).
// To compute h[] in O(N log N) time, do the following:
//   1. Compute F and G (pass invert=false as the argument).
//   2. Get H by element-wise multiplying F and G
//   3. Get h by taking the inverse FFT (pass invert=true)

#define MAXN (1 << 3)     // must be power of 2

typedef complex<double> T;
typedef vector<T> VT;
const double PI = 4*atan(1);

void FFT(VT &a, bool invert) {
    size_t n = a.size();

    for (int i=1, j=0; i<n; ++i) {
        int bit = n >> 1;
        for (; j>=bit; bit>>=1)
            j -= bit;
        j += bit;
        if (i < j)
            swap(a[i], a[j]);
    }

    for (int len=2; len<=n; len<<=1) {
        double ang = 2*PI/len * (invert ? -1 : 1);
        T wlen (cos(ang), sin(ang));
        for (int i=0; i<n; i+=len) {
            T w(1);
            for (int j=0; j<len/2; ++j) {
                T u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }

    if(invert)
        for (int i=0; i<n; ++i)
            a[i] /= n;
}
```

LinearAlgebra.cc

```
// Useful linear algebra routines.
#define FOR(v,l,u) for( size_t v = l; v < u; ++v )
typedef double        T;   // the code below only supports fields
typedef vector<T>      VT;
typedef vector<VT>     VVT;
typedef vector<size_t> VI;
typedef vector<bool>   VB;
typedef vector<long long> VN;
typedef vector<VN> VVN;
typedef long long ll;
// Given an m-by-n matrix A, compute its reduced row echelon form,
// returning a value like the determinant.
// If m = n, the returned value *is* the determinant of A.
// If m != n, the returned value is nonzero iff A has full row rank.
// To compute rank(A), get its RREF, and count the nonzero rows.
T GaussJordan( VVT &A ) {
    const size_t m = A.size(), n = A[0].size();
    T det = 1;
    size_t pj = 0;        // walking pointer for the pivot column
    FOR(k,0,m) {
        size_t pi = k;
        while( pj < n ) {    // find the best row below k to pivot
            FOR(i,k,m) if( fabs(A[i][pj]) > fabs(A[pi][pj]) ) pi = i;
            if( !feq(0.0, A[pi][pj]) ) { // we have our new pivot
                if( pi != k ) {
                    swap( A[pi], A[k] );
                    pi = k;
                    det *= -1;
                }
                break;
            }
            FOR(i,k,m) A[i][pj] = 0; // This column is zeros below row k
            ++pj;                    // So move on to the next column
        }
        if( pj == n ) { det = 0; break; } // we're done early
        T s = 1.0/A[pi][pj];        // scale the pivot row
        FOR(j,pj,n) A[pi][j] *= s;
        det /= s;
        FOR(i,0,m) if( i != pi ) { // subtract pivot from other rows
            T a = A[i][pj];          // multiple of pivot row to subtract
            FOR(j,pj,n) A[i][j] -= a*A[pi][j];
        }
        ++pj;
    }
    return det;
}
// In-place invert A.
void InvertMatrix( VVT &A ) {
    const size_t n = A.size();
    FOR(i,0,n) FOR(j,0,n) A[i].push_back( (i==j) ? 1 : 0 ); // augment
    GaussJordan( A );                            // compute RREF
    FOR(i,0,n) FOR(j,0,n) A[i][j] = A[i][j+n]; // copy A inverse over
    FOR(i,0,n) A[i].resize(n);                 // get rid of cruft
}
// Given m-by-n A and m-by-q b, compute a matrix x with Ax = b.
// This solves q separate systems of equations simultaneously.
// Fix k in [0,q).
// x[*][k] indicates a candidate solution to the jth equation.
// has_sol[k] indicates whether a solution is actually solution.
// The return value is the dimension of the kernel of A.
// Note that this is the dimension of the space of solutions when
// they exist.
size_t SolveLinearSystems( const VVT &A, const VVT &b, VVT &x, VB &has_sol )
 ↪   {
    const size_t m = A.size(), n = A[0].size(), q = b[0].size();
    VVT M = A;                                    // copy
    FOR(i,0,m) FOR(j,0,q) M[i].push_back(b[i][j]); // augment
    GaussJordan( M );                             // RREF
    x = VVT(n, VT(q, 0));
    size_t i = 0, jz = 0;
    while( i < m ) {
        while( jz < n && feq(M[i][jz],0) ) ++jz;
        if( jz == n ) break; // all zero means we're starting the kernel
        FOR(k,0,q) x[jz][k] = M[i][n+k]; // first nonzero is always 1
        ++i;
    }
    size_t kerd = n - i; // i = row rank = column rank
    has_sol = VB(q,true);
    while( i < m ) {
        FOR(k,0,q) if( !feq(M[i][n+k],0) ) has_sol[k] = false;
        ++i;
    }
    return kerd;
}
// Given m-by-n A, compute a basis for the kernel of A.
// The return value is in K, which is interpreted as a length-d array of
// n-dimensional vectors. (So K.size() == dim(Ker(A)))
// The return value is K.size().
size_t KernelSpan( const VVT &A, VVT &K ) {
    const size_t m = A.size(), n = A[0].size();
    VVT M = A;
    GaussJordan(M);
    K = VVT();
    VB all_zero(n,true);
    FOR(i,0,m) {
        size_t jz = 0;
        while( jz < n && feq(M[i][jz],0) ) ++jz;
        if( jz == n ) break; // skip to the easy part of the kernel
        all_zero[jz] = false;
        FOR(j,jz+1,n) if( !feq(M[i][j],0) ) {
            all_zero[j] = false;
            K.push_back( VT(n,0) );
            K.back()[jz] = -1 * M[i][j];
            K.back()[j]  = 1;
        }
    }
    FOR(j,0,n) if( all_zero[j] ) {
        K.push_back( VT(n,0) );
        K.back()[j] = 1;
    }
    return K.size();
}

// code for fast linear recurrence evaluation. Based on blog post at:
// http://fusharblog.com/solving-linear-recurrence-for-programming-contest/
// used for AC answer at: http://codeforces.com/contest/678/problem/D

// compute AB with entries mod M
VVN matrix_mult(VVN A, VVN B, ll M) {
    VVN result(A.size(), VN(B[0].size(), 0));
    for (int i = 0; i < result.size(); ++i) {
        for (int j = 0; j < result[i].size(); ++j) {
            for (int k = 0; k < A[0].size(); ++k) {
                result[i][j] = (result[i][j] + A[i][k] * B[k][j]) % M;
            }
        }
    }
    return result;
}

// compute A^n with entries mod M
// if A is m x m, takes O(m^3 log n) time
VVN matrix_pow(VVN A, ll n, ll M) {
    if (n == 1) {
        return A;
    }
    else if (n % 2 == 0) {
        VVN smaller = matrix_pow(A, n/2, M);
        return matrix_mult(smaller, smaller, M);
    }
    else {
        return matrix_mult(matrix_pow(A, n-1, M), A, M);
    }
}

// computes nth term of f(n) = rec[0]*f(n-1) + rec[1]*f(n-2) + ... +
 ↪   rec[k-1]*f(n-k) + c
// given that f(1) = init[0], f(2) = init[1], ..., f(k) = init[k-1]
// in O(k^3 log n) time
ll eval_rec(VN rec, VN init, T n, T M, T c = 0) {
    size_t k = rec.size();
    VVN mat(k+1, VN(k+1, 0));

    for (int i = 0; i < k; ++i) {
        mat[i][i+1] = 1;
    }
    for (int i = 0; i < k; ++i) {
        mat[k-1][i] = rec[k-1-i];
    }
    mat[k][k] = 1;

    if (n == 1)
        return init[0];

    VVN mat_new = matrix_pow(mat, n-1, M);
    VVN vect(k+1, VN(1));
    for (int i = 0; i < k; ++i)
        vect[i][0] = init[i];
    vect[k][0] = c;
```

```
    VVN vect_new = matrix_mult(mat_new, vect, M);
    return vect_new[0][0];
}
```

Simplex.cc

```
// Ripped from http://web.stanford.edu/~liszt90/acm/notebook.html#file17
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

// BEGIN CUT
#define ACM_assert(x) {if(!(x))*((long *)0)=666;}
//#define TEST_LEAD_OR_GOLD
#define TEST_HAPPINESS
// END CUT
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
    int m, n;
    VI B, N;
    VVD D;
    LPSolver(const VVD &A, const VD &b, const VD &c) :
        m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
        for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
        for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }
        for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m+1][n] = 1;
    }
    void Pivot(int r, int s) {
        for (int i = 0; i < m+2; i++) if (i != r)
            for (int j = 0; j < n+2; j++) if (j != s)
                D[i][j] -= D[r][j] * D[i][s] / D[r][s];
        for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
        for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];
        D[r][s] = 1.0 / D[r][s];
        swap(B[r], N[s]);
    }
    bool Simplex(int phase) {
        int x = phase == 1 ? m+1 : m;
        while (true) {
            int s = -1;
            for (int j = 0; j <= n; j++) {
                if (phase == 2 && N[j] == -1) continue;
                if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
            }
            if (D[x][s] >= -EPS) return true;
            int r = -1;
            for (int i = 0; i < m; i++) {
                if (D[i][s] <= 0) continue;
                if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
                        D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;
            }
            if (r == -1) return false;
            Pivot(r, s);
        }
    }

    DOUBLE Solve(VD &x) {
        int r = 0;
        for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
        if (D[r][n+1] <= -EPS) {
            Pivot(r, n);
            if (!Simplex(1) || D[m+1][n+1] < -EPS) return -numeric_limits<DOUBLE>::infinity();
            for (int i = 0; i < m; i++) if (B[i] == -1) {
                int s = -1;
                for (int j = 0; j <= n; j++)
                    if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
                Pivot(i, s);
            }
        }
        if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
        x = VD(n);
        for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
        return D[m][n+1];
    }
    // BEGIN CUT
    void Print() {
        cout << "N = "; for (int i = 0; i < N.size(); i++) printf("%8d", N[i]); cout << endl;
        cout << "B = "; for (int i = 0; i < B.size(); i++) printf("%8d", B[i]); cout << endl;
        cout << endl;
        for (int i = 0; i < D.size(); i++) {
            for (int j = 0; j < D[i].size(); j++) {
                printf("%8.2f", double(D[i][j]));
            }
            printf("\n");
        }
        printf("\n");
    }
    // END CUT
};
// BEGIN CUT
#ifdef TEST_HAPPINESS
int main() {
    int n, m;
    while (cin >> n >> m) {
        ACM_assert(3 <= n && n <= 20);
        ACM_assert(3 <= m && m <= 20);
        VVD A(m, VD(n));
        VD b(m), c(n);
        for (int i = 0; i < n; i++) {
            cin >> c[i];
            ACM_assert(c[i] >= 0);
            ACM_assert(c[i] <= 10);
        }
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++)
                cin >> A[i][j];
            cin >> b[i];
            ACM_assert(b[i] >= 0);
            ACM_assert(b[i] <= 1000);
        }
        LPSolver solver(A, b, c);
        VD sol;
        DOUBLE primal_answer = m * solver.Solve(sol);

        VVD AT(A[0].size(), VD(A.size()));
        for (int i = 0; i < A.size(); i++)
            for (int j = 0; j < A[0].size(); j++)
                AT[j][i] = -A[i][j];
        for (int i = 0; i < c.size(); i++)
            c[i] = -c[i];
        for (int i = 0; i < b.size(); i++)
            b[i] = -b[i];
        LPSolver solver2(AT, c, b);
        DOUBLE dual_answer = -m * solver2.Solve(sol);
        ACM_assert(fabs(primal_answer - dual_answer) < 1e-10);
        int primal_rounded_answer = (int) ceil(primal_answer);
        int dual_rounded_answer = (int) ceil(dual_answer);
        // The following assert fails b/c of the input data.
        // ACM_assert(primal_rounded_answer == dual_rounded_answer);
        cout << "Nasa can spend " << primal_rounded_answer << " taka." << endl;
    }
}
#else
#ifdef TEST_LEAD_OR_GOLD
int main() {
    int n;
    int ct = 0;
    while (cin >> n) {
        if (n == 0) break;
        VVD A(6, VD(n));
        VD b(6), c(n, -1);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < 3; j++) {
                cin >> A[j][i]; A[j+3][i] = -A[j][i];
            }
        }
        for (int i = 0; i < 3; i++) {
            cin >> b[i]; b[i+3] = -b[i];
        }
        if (ct > 0) cout << endl;
        cout << "Mixture " << ++ct << endl;
        LPSolver solver(A, b, c);
        VD x;
        double obj = solver.Solve(x);
        if (isfinite(obj)) {
            cout << "Possible" << endl;
        } else {
            cout << "Impossible" << endl;
        }
    }
    return 0;
}
#else
// END CUT
int main() {
    const int m = 4;
    const int n = 3;
    DOUBLE _A[m][n] = {
        { 6, -1, 0 },
        { -1, -5, 0 },
        { 1, 5, 1 },
        { -1, -5, -1 }
    };
    DOUBLE _b[m] = { 10, -4, 5, -5 };
    DOUBLE _c[n] = { 1, -1, 0 };

    VVD A(m);
    VD b(_b, _b + m);
    VD c(_c, _c + n);
    for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);

    LPSolver solver(A, b, c);
    VD x;
    DOUBLE value = solver.Solve(x);

    cerr << "VALUE: "<< value << endl;
    cerr << "SOLUTION:";
    for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
    cerr << endl;
    return 0;
}
// BEGIN CUT
#endif
#endif
// END CUT
```

Rational.cc

```
// Rational struct. Uses lcm to keep in simplified form.
// Simply replace "double" (or "int") with "rat" and use
// contructor to initialize constants. Rat class handles
// everything else.
//
// Written in such a way as to avoid overflow if possible.

struct rat {
    T n, d;

    rat(T n, T d) {
        T k = gcd(n, d);
        this->n = n/k;
```

```
        this->d = d/k;
    }

    rat(T n) : n(n), d(1) {}
};

rat operator + (const rat &a, const rat &b) {
    T new_d = lcm(a.d, b.d); // if overflow occurs, this may be 0
                             // causing floating point exception
    T a_scale = new_d / a.d;
    T b_scale = new_d / b.d;

    return rat(a.n*a_scale + b.n*b_scale, new_d);
}

rat operator * (const T s, const rat &a) {
    return rat(a.n * s, a.d);
}

rat operator * (const rat &a, const T s) {
    return s*a;
}

rat operator - (const rat &a, const rat &b) {
    return a + (-1 * b);
}

rat operator * (const rat &a, const rat &b) {
    return rat(a.n*b.n, a.d*b.d);
}

rat operator / (const rat &a, const rat &b) {
    return a * rat(b.d, b.n);
}
```

SegmentTree.cc

```
// Segment Tree with lazy propagation. This solves
// AhoyPirates on UVa Online Judge in 1.172 seconds.
// To use, you only need to change SegmentNode data,
// the merge function, update, and initialization;
// the functions are written in such a way as to handle everything else.
// If range updates are not necessary,
// ignore the updateVal and rangeUpdate function.
// The following implementation is an example for RMQ.

#define MAXN 1000 // the maximum input length of the sequnece, good idea to
 ↪   add 10 or so to this

typedef signed long long int T; // the type of the underlying sequence
typedef vector<size_t> VI;
typedef vector<T> VT;

struct SegmentNode {
    // segment data
    T maxVal;

    // update data
    T updateVal;
};

SegmentNode st[MAXN*4];
T A[MAXN];

size_t left(size_t cur) { return cur << 1; }
size_t right(size_t cur) { return (cur << 1) + 1; }

// create merging function here
void merge(SegmentNode &left, SegmentNode &right, SegmentNode &result) {
    result.maxVal = max(left.maxVal, right.maxVal);
}

// only use if range update is needed
void updateChildren(size_t cur, size_t L, size_t R);

// merge handles all querying, no changes needed here
// note that when calling any segment funcitons in main,
// the first three parameters will be 1, 0, length-1
SegmentNode query(size_t cur, size_t L, size_t R, size_t LQ, size_t RQ) {
    if (L >= LQ && R <= RQ)
        return st[cur];

    updateChildren(cur, L, R);
    size_t M = (L+R)/2;

    if (M < LQ)
        return query(right(cur), M+1, R, LQ, RQ);

    if (M+1 > RQ)
        return query(left(cur), L, M, LQ, RQ);

    SegmentNode leftResult = query(left(cur), L, M, LQ, RQ);
    SegmentNode rightResult = query(right(cur), M+1, R, LQ, RQ);
    SegmentNode result;

    merge(leftResult, rightResult, result);

    return result;
}

// only implement if necessary
void update(size_t cur, size_t L, size_t R, size_t idx, T val) {
    if (L == idx && R == idx) {
        // write update of single value here
        st[cur].maxVal = val;
    }
    else if (L <= idx && R >= idx) {
        size_t M = (L+R)/2;
        update(left(cur), L, M, idx, val);
        update(right(cur), M+1, R, idx, val);

        merge(st[left(cur)], st[right(cur)], st[cur]);
    }
}

// only implement if necessary
void rangeUpdate(size_t cur, size_t L, size_t R, size_t Lbound, size_t
 ↪   Rbound, T val) {
    if (L >= Lbound && R <= Rbound) {
        // implement range update here
        st[cur].maxVal += val;

        // set update vals here
        st[cur].updateVal += val;
    }
    else if (L <= Rbound && R >= Lbound) {
        updateChildren(cur, L, R);

        size_t M = (L+R)/2;
        rangeUpdate(left(cur), L, M, Lbound, Rbound, val);
        rangeUpdate(right(cur), M+1, R, Lbound, Rbound, val);

        merge(st[left(cur)], st[right(cur)], st[cur]);
    }
}

void updateChildren(size_t cur, size_t L, size_t R) {
    rangeUpdate(left(cur), L, R, L, R, st[cur].updateVal);
    rangeUpdate(right(cur), L, R, L, R, st[cur].updateVal);

    // reset update vals
    st[cur].updateVal = 0;
}

void build(size_t cur, size_t L, size_t R) {
    // initialize update vals
    st[cur].updateVal = 0;

    if (L == R) {
        // initialize single valiue here
        st[cur].maxVal = A[L];
    }
    else {
        size_t M = (L+R)/2;
        build(left(cur), L, M);
        build(right(cur), M+1, R);

        merge(st[left(cur)], st[right(cur)], st[cur]);
    }
}
```

BIT.cc

```
// T is a type with +/- operations and identity element '0'.

// Least significant bit of a. Used throughout.
int LSB( int a ) { return a ^ (a & (a-1)); }

// To use it, instantiate it as 'BIT(n)' where n is the size of the
 ↪   underlying
// array. The BIT then assumes a value of 0 for every element. Update each
// index individually (with 'add') to use a different set of values.
//
// Note that it is assumed that the underlying array has size a power of 2!
// This mostly just simplifies the implementation without any loss in speed.
// Just use the closest power of 2 larger than the max input size. Even if
 ↪   some test
// cases do not test this high, initialization is extremely quick.
//
// The comments below make reference to an array 'arry'. This is the
 ↪   underlying
// array. (A is the data stored in the actual tree.)
struct BIT {
    int N;
    VT A;
    BIT( int n ) : N(n), A(N+1,0) {} // n must be a power of 2
    // add v to arry[idx]
    void add( int idx, T v ) {
        for( int i = idx+1; i <= N; i += LSB(i) ) A[i] += v;
    }
    // get sum( arry[0..idx] )
    T sum( int idx ) {
        T ret = 0;
        for( int i = idx+1; i > 0; i -= LSB(i) ) ret += A[i];
        return ret;
    }
    // get sum( arry[l..r] )
    T sum_range( int l, int r ) { return sum(r) - sum(l-1); }
    // Find largest r so that sum( arry[0..r] ) <= thresh
    // This assumes arry[i] >= 0 for all i > 0, for monotonicity.
    // This takes advantage of the specific structure of LSB() to simplify
     ↪   the
    // binary search.
    int largest_at_most( T thresh ) {
        int r = 0, del = N;
        while( del && r <= N ) {
            int q = r + del;
            if( A[q] <= thresh ) {
                r = q;
                thresh -= A[q];
            }
            del /= 2;
        }
        return r-1;
    }
};

// A 'range-add'/'index query' BIT
struct BIT_flip {
    BIT A;
    BIT_flip( int n ) : A(n) {}
    // add v to arry[l,r]
    void add( int l, int r, T v ) {
        A.add(l,v);
        A.add(r+1,-v);
    }
    // get arry[idx]
    T query( int idx ) {
        return A.sum(idx);
    }
};
```

```
// A 'range-add'/'range-query' data structure that uses BITs.
struct BIT_super {
    int N;
    BIT_flip m, b; // linear coefficient, constant coefficient
    BIT_super( int n ) : N(n), m(n), b(n) {}
    // add v to arry[l..r]
    void add( int l, int r, T v ) {
        m.add(l,r,v);         // add slope on active interval
        b.add(l,N,l*(-v));    // subtract contribution from pre-interval
        b.add(r+1,N,(r+1)*v);       // add total contribution to
         ↪   after-interval
    }
    // get sum( arry[0..r] )
    T query( int r ) {
        ++r;
        return m.query(r)*r + b.query(r);
    }
    // get sum( arry[l..r] )
    T query_range( int l, int r ) {
        return query(r) - query(l-1);
    }
};

// A 2-dimensional specialization of BITd. (see below)
// What took 'nlogn' before now takes 'nlog^2(n)'.
struct BIT2 {
    int N1;
    int N2;
    vector<BIT> A;
    BIT2( int n1, int n2 ) : N1(n1), N2(n2) {
        A.resize(N1+1, BIT(n2));
    }
    // add v to arry[x][y]
    void add( int x, int y, T v ) {
        for( int i = x+1; i <= N1; i += LSB(i) ) A[i].add(y,v);
    }
    // get sum( arry[0..x][0..y] ).
    T sum( int x, int y ) {
        T ret = 0;
        for( int i = x+1; i > 0; i -= LSB(i) ) ret += A[i].sum(y);
        return ret;
    }
    // get sum( arry[xL..xH)[yL..yH) ).
    T sum_range( int xL, int yL, int xH, int yH ) {
        return sum(xH,yH) + sum(xL-1,yL-1) - sum(xH,yL-1) - sum(xL-1,yH);
    }
};

// A d-dimensional binary indexed tree
// What took 'nlogn' before now takes 'nlog^d(n)'.
//
// To construct it, set dims to be the vector of dimensions, and pass
// d <- dims.size().
typedef vector<int> VI;
struct BITd {
    int N;
    int D;
    vector<BITd> A;
    T V;
    BITd( const VI &dims, int d ) : N(dims[d-1]), D(d) {
        if( D == 0 ) V = 0;
        else         A.resize( N+1, BITd( dims, D-1 ) );
    }
    void add( const VI &idx, T v ) {
        if( D == 0 ) V += v;
        for( int i = idx[D-1]+1; i <= N; i += LSB(i) ) A[i].add(idx,v);
    }
    T sum( const VI &idx ) {
        if( D == 0 ) return V;
        T ret = 0;
        for( int i = idx[D-1]+1; i > 0; i -= LSB(i) ) ret += A[i].sum(idx);
        return ret;
    }
    T sum_range( VI lo, VI hi ) {
        FOR(i,0,D) --lo[i];
        // In higher dimensions, we have to use inclusion-exclusion
        int BD = ((int)1) << D;
        T ret = 0;
        FOR(S,0,BD) {
            int sign = 1;
            VI q(lo);
            FOR(b,0,BD) if( (S >> b) & 1 ) {
                q[b] = hi[b];
                sign *= -1;
            }
            ret += sign * sum(q);
        }
        return ret;
    }
};
```

KMP.cc

```
// An implemention of Knuth-Morris-Pratt substring-finding.
// The table constructed with KMP_table may have other uses.
typedef vector<size_t> VI;
// In the KMP table, T[i] is the *length* of the longest *prefix*
// which is also a *proper suffix* of the first i characters of w.
void KMP_table( string &w, VI &T ) {
    T = VI( w.size()+1 );
    size_t i = 2, j = 0;
    T[1] = 0; // T[0] is undefined
    while( i <= w.size() ) {
        if(  w[i-1] == w[j]  ) { T[i] = j+1; ++i; ++j; } // extend previous
        else if( j > 0 )       { j = T[j]; }             // fall back
        else                   { T[i] = 0; ++i; }        // give up
    }
}
// Search for first occurrence of q in s in O(|q|+|s|) time.
size_t KMP( string &s, string &q ) {
    size_t m, z;   m = z = 0;   // m is the start, z is the length so far
    VI T; KMP_table(q, T);      // init the table
    while( m+z < s.size() ) {   // while we're not running off the edge...
        if( q[z] == s[m+z] ) {  // next character matches
            ++z;
            if( z == q.size() ) return m; // we're done
        }
        else if( z > 0 ) {      // fall back to the next best match
            m += z - T[z];   z = T[z];
        }
        else {                  // go back to start
            m += 1;         z = 0;
        }
    }
    return s.size();
}
```

AhoCorasick.cc

```
// An implementation of Aho-Corasick dictionary matching algorithm
// Taken directly from the paper

typedef vector<size_t> VI;
typedef vector<string> VS;

struct node {
    unordered_map<char, node*> g;
    node* f;
    node* output = NULL;
    bool isWord = false;
    size_t num;

    node(size_t num) : num(num) {}
    node() {}
};

void enter(string S, node *root, size_t &n) {
    node *state = root;
    size_t j = 0;
    while (state->g.count(S[j]) != 0) {
        state = state->g[S[j]];
        ++j;
    }
    for (; j < S.size(); ++j) {
        state->g[S[j]] = new node(n++);
        state = state->g[S[j]];
    }
    state->isWord = true;
}

void construct_f(node *root) {
    queue<node*> q;
    for (auto it = root->g.begin(); it != root->g.end(); ++it) {
        q.push(it->second);
        it->second->f = root;
    }
    while (!q.empty()) {
        node *r = q.front(); q.pop();
        for (auto it = r->g.begin(); it != r->g.end(); ++it) {
            q.push(it->second);
            node *state = r->f;
            while (state->g.count(it->first) == 0 && state->num != 0)
                state = state->f;
            if (state->g.count(it->first) != 0)
                it->second->f = state->g[it->first];
            else
                it->second->f = root;
            if (it->second->f->isWord)
                it->second->output = it->second->f;
            else   // may assign NULL pointer
                it->second->output = it->second->f->output;
        }
    }
}

node* ConstructAutomaton(VS &dictionary) {
    size_t n = 0;
    node *root = new node(n++);
    for (size_t i = 0; i < dictionary.size(); ++i) {
        enter(dictionary[i], root, n);
    }
    construct_f(root);
    return root;
}

void PrintMatches(node *root, string x) {
    node *state = root;
    for (size_t i = 0; i < x.size(); ++i) {
        while (state->g.count(x[i]) == 0 && state->num != 0) state =
         ↪   state->f;
        if (state->g.count(x[i]) != 0)
            state = state->g[x[i]];
        node *out = state;
        while (true) {
            if (out->isWord) cout << i << " " << out->num << " " << endl;
            if (out->output == NULL) break;
            out = out->output;
        }
    }
}
```

SuffixArray.cc

```
// A prefix-doubling suffix array construction implementation.
#define FOR(v,l,u) for( size_t v = l; v < u; ++v )

typedef vector<size_t> VI;
typedef pair<int, int> II;

#define MAX_N 100010
int RA[MAX_N], tempRA[MAX_N];
int SA[MAX_N], tempSA[MAX_N];
int c[MAX_N];

// uses Radix Sort as a subroutine to sort in O(n)
void CountingSort(int n, int k) {
    int i, sum, maxi = max(300, n);
    memset(c, 0, sizeof c);
    for (i = 0; i < n; i++)
```

```cpp
        c[i+k < n ? RA[i+k] : 0]++;
    for (i = sum = 0; i < maxi; i++) {
        int t = c[i]; c[i] = sum; sum += t;
    }
    for (i = 0; i < n; i++)
        tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
    for (i = 0; i < n; i++)
        SA[i] = tempSA[i];
}

// Construct SA in O(n log n) time. Solves UVa Online Judge "Glass Beads" in
 ↪   .5 seconds
void ConstructSA(string T) {
    int i, k, r, n = T.size();
    for (i = 0; i < n; i++) RA[i] = T[i];
    for (i = 0; i < n; i++) SA[i] = i;
    for (k = 1; k < n; k <<= 1) {
        CountingSort(n, k);
        CountingSort(n, 0);
        tempRA[SA[0]] = r = 0;
        for (i = 1; i < n; i++)
            tempRA[SA[i]] =
                (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ?
                 ↪   r : ++r;
        for (i = 0; i < n; i++)
            RA[i] = tempRA[i];
        if (RA[SA[n-1]] == n-1) break;
    }
}

// Given a "string" w, and suffix array SA, compute the array LCP for which
// the suffix starting at SA[i] matches SA[i+1] for exactly LCP[i]
 ↪   characters
// It is assumed that the last character of w is the unique smallest-rank
// character in w. Runs in O(n).
void LongestCommonPrefix( const string &w, VI &LCP ) {
    const size_t N = w.size();   VI rk(N);
    FOR(i,0,N) rk[ SA[i] ] = i;
    LCP = VI(N-1);   size_t k = 0;
    FOR(i,0,N) {
        if( rk[i] == N-1 ) continue;
        size_t j = SA[ rk[i]+1 ];
        while( w[ i+k ] == w[ j+k ] ) ++k;
        LCP[ rk[i] ] = k;
        if( k > 0 ) --k;
    }
}
// Finds the smallest and largest i such that the prefix of suffix SA[i]
 ↪   matches
// the pattern string P. Returns (-1, -1) if P is not found in T. Runs in O(m
 ↪   log n).
II StringMatching(const string &T, const string P) {
    int n = T.size(), m = P.size();
    int lo = 0, hi = n-1, mid = lo;
    while (lo < hi) {
        mid = (lo + hi) / 2;
        int res = T.compare(SA[mid], m, P);
        if (res >= 0) hi = mid;
        else          lo = mid+1;
    }
    if (T.compare(SA[lo], m, P) != 0) return II(-1, -1);
    II ans; ans.first = lo;
    lo = 0; hi = n-1; mid = lo;
    while (lo < hi) {
        mid = (lo + hi) / 2;
        int res = T.compare(SA[mid], m, P);
        if (res > 0) hi = mid;
        else         lo = mid+1;
    }
    if (T.compare(SA[hi], m, P)) hi--;
    ans.second = hi;
    return ans;
}
```

ArticulationPoint.cc

```cpp
// This is code for computing articulation points of graphs,
// ie points whose removal increases the number of components in the graph.
// This works when the given graph is not necessarily connected, too.
typedef vector<size_t> VI;
typedef vector<VI>     VVI;
typedef vector<bool>   VB;

struct artpt_graph {
    size_t N;     VVI adj;        // basic graph stuff
    VI  parent, n_children, rank; // dfs tree
    VB  is_art;   VI reach;       // articulation points
    artpt_graph( size_t N ) : N(N), adj(N), is_art(N) {}
    void add_edge( size_t s, size_t t ) {
        adj[s].push_back(t);
        adj[t].push_back(s);
    }
    size_t dfs_artpts( size_t rt, VB &visited, size_t R ) {
        visited[rt] = true;
        rank[rt] = R++;
        reach[rt] = rank[rt]; // reach[rt] <= rank[rt] always.
        FOR(i,0,adj[rt].size()) {
            size_t v = adj[rt][i];
            if( v == parent[rt] ) continue;
            if( visited[v] )
                reach[rt] = min(reach[rt], rank[v]);
            else {
                ++n_children[rt];
                parent[v] = rt;
                R = dfs_artpts( v, visited, R );
                reach[rt] = min(reach[rt], reach[v]);
            }
        }
        if( reach[rt] < rank[rt] || n_children[rt] == 0 )
            is_art[rt] = false;
        return R;
    }
    void comp_articulation_points() {
        is_art = VB(N, true);  reach = VI(N);
        parent = VI(N,N);      rank = VI(N);      n_children = VI(N,0);
        VB visited(N,false);   size_t R = 0;
        FOR(i,0,N) {
            if( visited[i] ) continue;
            R = dfs_artpts(i, visited, R); // this is not right on i
            is_art[i] = (n_children[i] >= 2); // but we can fix it!
        }
    }
};
```

BellmanFord.cc

```cpp
// A Bellman-Ford implementation.
// bellmanford(S) computes the shortest paths from S to all other nodes.
// It returns true if there are no negative cycles in the graph,
// and false otherwise.
// D[v] is set to the shortest path from S to v (when it exists).
// P[v] is set to the parent of v in the shortest-paths tree,
// or N (for which there is no index) if v is not reachable from S.
#define FOR(v,l,u) for( size_t v = l; v < u; ++v )
typedef signed long long int T;
typedef vector<T>      VT;
typedef vector<VT>     VVT;
typedef vector<bool>    VB;
typedef vector<VB>     VVB;
typedef vector<size_t>  VI;
typedef vector<VI>     VVI;
const T UNBOUNDED = numeric_limits<T>::min(); // -infinity for doubles
const T INFINITY  = numeric_limits<T>::max(); // infinity for doubles

struct bellmanford_graph {
    size_t N; // number of nodes
    VVI    A; // adjacency list
    VVT    W; // weight of edges
    VT     D; // shortest distance
    VI     P; // parent in the shortest path tree
    bellmanford_graph( size_t N ) : N(N), A(N), W(N) {}
    void add_edge( size_t s, size_t t, T w ) {
        A[s].push_back(t);
        W[s].push_back(w);
    }
    bool bellmanford( size_t S ) {
        D = VT(N, INFINITY); D[S] = 0; P = VI(N,N);
        FOR(k,0,N)
        FOR(s,0,N)
        FOR(i,0,A[s].size()) {
            size_t t = A[s][i];
            if( D[s] == INFINITY ) continue;
            if( D[t] > D[s] + W[s][i] ) {
                if( k == N-1 ) {
                    D[t] = UNBOUNDED;
                }
                else {
                    D[t] = D[s] + W[s][i];
                    P[t] = s;
                }
            }
        }
        FOR(v,0,N) if( D[v] == UNBOUNDED ) return false;
        return true;
    }
};
```

FloydWarshall.cc

```cpp
// Floyd-Warshall implementation with negative cycle detection.
// This will modify the graph, computing its transitive closure.
//
// If there is an upper bound for any simple path length,
// then create a constant INF equal to that,
// and set W[i][j] = INF when there is no edge i->j.
// You can then remove all reference to A.
// Notable generalizations:
// - Finding paths with maximum minimum-capacity-along-path
// - Transitive closure (done with A below)
#define FOR(v,l,u) for( size_t v = l; v < u; ++v )
typedef signed long long int T; // anything with <, +, and 0
typedef vector<T>      VT;
typedef vector<VT>     VVT;
typedef vector<size_t> VI;  // only if you want the actual paths
typedef vector<VI>     VVI; // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
typedef vector<bool>   VB;  // only if you don't have an upper bound
typedef vector<VB>     VVB; // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

struct floydwarshall_graph {
    size_t N;   // Number of nodes
    VVB    A;   // [i][j] is true iff there exists an edge i -> j
    VVT    W;   // [i][j] is the weight of the edge i -> j.
    VVI    P;   // [i][j] is the next node in shortest path i -> j
    floydwarshall_graph( size_t n ) :
        N(n),  A(n,VB(n,false)),  W(n,VT(n,0)),  P(n,VI(n,n)) {}
    void add_edge( size_t s, size_t t, T w ) {
        A[s][t] = true;     W[s][t] = w;    P[s][t] = t;
    }
    bool floydwarshall() {
        FOR(k,0,N)    // We've computed paths using only {0, 1, ..., k-1}
        FOR(i,0,N)    // Now compute the shortest path from i -> j
        FOR(j,0,N) {  // when considering a path using k.
            if( !A[i][k] || !A[k][j] ) continue;     // skip invalid
            if( !A[i][j] ) {                         // first time
                A[i][j] = true;
                W[i][j] = W[i][k] + W[k][j];
                P[i][j] = P[i][k];
            }
            if( W[i][k] + W[k][j] < W[i][j] ) {      // future times
                P[i][j] = P[i][k];
                W[i][j] = W[i][k] + W[k][j];
            }
        }
        FOR(i,0,N) if( W[i][i] < 0 ) return false; // negative cycle.
        return true; // no negative cycle.
    }
};
```

MaxCardBipartiteMatching.cc

```
// This code performs maximum (cardinality) bipartite matching.
// Does not support weighted edges.
//
// Running time: O(|E| |V|) -- often much faster in practice
//
//   INPUT: adj_list[i][j] = edge between row node i and column node
 ↪   adj_list[i][j]
//          mr[i] = vector of size #rows, initialized to -1
//          mc[j] = vector of size #columns, initialized to -1
//
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef vector<bool> VB;

bool FindMatch(int i, const VVI &adj_list, VI &mr, VI &mc, VB &seen) {
  for (int j = 0; j < adj_list[i].size(); j++) {
    int item = adj_list[i][j];
    if (!seen[item]) {
      seen[item] = true;
      if (mc[item] < 0 || FindMatch(mc[item], adj_list, mr, mc, seen)) {
        mr[i] = item;
        mc[item] = i;
        return true;
      }
    }
  }
  return false;
}

// mr should be a vector of size number of row items, initialized to -1
// mc should be a vector of size number of column items, initialized to -1
int BipartiteMatching(const VVI &adj_list, VI &mr, VI &mc) {
  int ct = 0;
  for (int i = 0; i < adj_list.size(); i++) {
    VB seen(mc.size(), false);
    if (FindMatch(i, adj_list, mr, mc, seen)) ct++;
  }
  return ct;
}
```

MaximumFlow-Dinic.cc

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).
// Taken from Stanford ACM:
 ↪   http://stanford.edu/~liszt90/acm/notebook.html#file1

const int INF = 2000000000;

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Dinic {
    int N;
    vector<vector<Edge> > G;
    vector<Edge *> dad;
    vector<int> Q;

    Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

    void AddEdge(int from, int to, int cap) {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

    long long BlockingFlow(int s, int t) {
        fill(dad.begin(), dad.end(), (Edge *) NULL);
        dad[s] = &G[0][0] - 1;

        int head = 0, tail = 0;
        Q[tail++] = s;
        while (head < tail) {
            int x = Q[head++];
            for (int i = 0; i < G[x].size(); i++) {
                Edge &e = G[x][i];
                if (!dad[e.to] && e.cap - e.flow > 0) {
                    dad[e.to] = &G[x][i];
                    Q[tail++] = e.to;
                }
            }
        }
        if (!dad[t]) return 0;

        long long totflow = 0;
        for (int i = 0; i < G[t].size(); i++) {
            Edge *start = &G[G[t][i].to][G[t][i].index];
            int amt = INF;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
                if (!e) { amt = 0; break; }
                amt = min(amt, e->cap - e->flow);
            }
            if (amt == 0) continue;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
                e->flow += amt;
                G[e->to][e->index].flow -= amt;
            }
            totflow += amt;
        }
        return totflow;
    }

    long long GetMaxFlow(int s, int t) {
        long long totflow = 0;
        while (long long flow = BlockingFlow(s, t))
            totflow += flow;
        return totflow;
    }
};
```

MaximumFlow-EdmondsKarp.cc

```
typedef double T; // also works for doubles, but use feq, etc
const T INFTY = numeric_limits<T>::max(); // ::infinity() for doubles

typedef vector<size_t> VI;
typedef vector<VI>     VVI;
typedef vector<T>      VT;
typedef vector<VT>     VVT;

// Edmonds-Karp algorithm for max-flow
// Runs in O(VE^2). Alternate complexity: O(f(V+E)),
// where f is the value of the maximum flow
struct edmondskarp_graph {
    size_t N;
    VVI A;
    VVT C, F;  // references to F can be removed if you don't want flows
    edmondskarp_graph( size_t _N ) : N(_N), A(N), C(N,VT(N,0)), F(N,VT(N,0))
     ↪   {}
    void add_capacity( size_t s, size_t t, T cap ) {
        if( cap == 0 ) return;
        if( C[s][t] == 0 && C[t][s] == 0 ) {
            A[s].push_back(t);
            A[t].push_back(s);
        }
        C[s][t] += cap;
        // If you subtract capacities, and want to remove edges with cap 0,
        // do so here, or afterward.
    }
    T Augment( const VI &P ) {
        T amt = INFTY;
        FOR(i,0,P.size()-1) amt = min(amt, C[ P[i] ][ P[i+1] ]);
        FOR(i,0,P.size()-1) {
            size_t u = P[i], v = P[i+1];
            C[u][v] -= amt;
            C[v][u] += amt;
            if( F[v][u] > amt ) {
                F[v][u] -= amt;
            }
            else {
                F[u][v] += amt - F[v][u];
                F[v][u] = 0;
            }
        }
        return amt;
    }
    bool bfs( size_t s, size_t t, VI &P ) {
        P = VI(N,N);
        VI Q(N);  size_t qh=0, qt=0;
        P[  Q[qt++] = s  ] = s;
        while( qh < qt && P[t] == N ) {
            size_t c = Q[qh++];
            FOR(i,0,A[c].size()) {
                size_t u = A[c][i];
                if( C[c][u] == 0 ) continue;
                if( P[u] != N ) continue;
                P[  Q[qt++] = u  ] = c;
            }
        }
        return P[t] != N;
    }
    T ComputeMaxFlow( size_t s, size_t t ) {
        T flow = 0;
        VI P;
        while( bfs(s,t,P) ) {
            VI path(1,t);
            size_t z = t;
            while( z != P[z] ) path.push_back( z = P[z] );
            path = VI(path.rbegin(), path.rend());
            flow += Augment(path);
        }
        return flow;
    }
};
```

MinCostBipartiteMatching.cc

```
///////////////////////////////////////////////////////////////////////////
// Min cost bipartite matching VIa shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.
///////////////////////////////////////////////////////////////////////////

typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
    int n = int(cost.size());
```

```
    // construct dual feasible solution
    VD u(n);
    VD v(n);
    for (int i = 0; i < n; i++) {
        u[i] = cost[i][0];
        for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
    }
    for (int j = 0; j < n; j++) {
        v[j] = cost[0][j] - u[0];
        for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
    }

    // construct primal solution satisfying complementary slackness
    Lmate = VI(n, -1);
    Rmate = VI(n, -1);
    int mated = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (Rmate[j] != -1) continue;
            if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
                Lmate[i] = j;
                Rmate[j] = i;
                mated++;
                break;
            }
        }
    }

    VD dist(n);
    VI dad(n);
    VI seen(n);

    // repeat until primal solution is feasible
    while (mated < n) {

        // find an unmatched left node
        int s = 0;
        while (Lmate[s] != -1) s++;

        // initialize Dijkstra
        fill(dad.begin(), dad.end(), -1);
        fill(seen.begin(), seen.end(), 0);
        for (int k = 0; k < n; k++)
        dist[k] = cost[s][k] - u[s] - v[k];

        int j = 0;
        while (true) {

            // find closest
            j = -1;
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                if (j == -1 || dist[k] < dist[j]) j = k;
            }
            seen[j] = 1;

            // termination condition
            if (Rmate[j] == -1) break;

            // relax neighbors
            const int i = Rmate[j];
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
                if (dist[k] > new_dist) {
                    dist[k] = new_dist;
                    dad[k] = j;
                }
            }
        }

        // update dual variables
        for (int k = 0; k < n; k++) {
            if (k == j || !seen[k]) continue;
            const int i = Rmate[k];
            v[k] += dist[k] - dist[j];
            u[i] -= dist[k] - dist[j];
        }
        u[s] += dist[j];

        // augment along path
        while (dad[j] >= 0) {
            const int d = dad[j];
            Rmate[j] = Rmate[d];
            Lmate[Rmate[j]] = j;
            j = d;
        }
        Rmate[j] = s;
        Lmate[s] = j;

        mated++;
    }

    double value = 0;
    for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

    return value;
}
```

MinCostMaxFlow.cc

```
// Min-cost Maximum Flow. The implementation has a few stages, some of which
// can be outright ignored if the graph is guaranteed to satisfy certain
// conditions. These are documented below.
#define FOR(v,l,u) for( size_t v = l; v < u; ++v )
typedef signed long long int T;   // the basic type of costs and flow.
typedef vector<T>           VT;
struct edge {
    size_t s, t;
    T cap, flow, cost;  // Note: cap is *residual* capacity.
    size_t di; // index of dual in t's edgelist.
    edge *dual; // the actual dual (see "compile_edges")
};
typedef vector<edge>   VE;
typedef vector<VE>     VVE;
typedef vector<size_t> VI;
typedef pair<T,size_t> DijkP;  // Dijkstra PQ element.
typedef priority_queue<DijkP, vector<DijkP>, std::greater<DijkP> >  DijkPQ;

struct mcmf_graph {
    size_t N;   VVE adj;  VT pot;
    mcmf_graph( size_t N ) : N(N), adj(N), pot(N,0) {}
    void add_edge( size_t s, size_t t, T cap, T cost ) {
        edge f, r;
        f.s = s; f.t = t; f.cap = cap; f.flow = 0; f.cost =  cost;
        r.s = t; r.t = s; r.cap =   0; r.flow = 0; r.cost = -cost;
        f.di = adj[t].size(); r.di = adj[s].size();
        adj[s].push_back(f);  adj[t].push_back(r);
    }
    void compile_edges() {
        FOR(v,0,N) FOR(i,0,adj[v].size()) { // This has to be done after all
            edge &e = adj[v][i];      // edges are added because vectors can
            e.dual = &adj[e.t][e.di]; // resize and move their contents.
        }
    }
    T Augment( const VE &path ) {
        T push = path[0].cap;
        FOR(i,0,path.size()) push = min(push, path[i].cap);
        FOR(i,0,path.size()) {
            edge &e = *(path[i].dual->dual); // the actual edge, not a copy
            e.cap -= push;  e.dual->cap += push;
            if( e.dual->flow >= push ) e.dual->flow -= push;
            else { e.flow += push - e.dual->flow;  e.dual->flow = 0; }
        }
        return push;
    }
    void ApplyPotential( const VT &delta ) {
        FOR(v,0,N) {
            FOR(i,0,adj[v].size()) {
                adj[v][i].cost += delta[v];
                adj[v][i].dual->cost -= delta[v];
            }
            pot[v] += delta[v];
        }
    }
    /* The following, down to "CancelNegativeCycles", are unnecessary if the
     * graph is guaranteed to have no negative cycles.
     * Alternatively, if you compute any maxflow, you can include these, and
     * run CancelNegativeCycles to find a cost-optimal maxflow. */
    bool dfs_negcycle_r( const size_t rt, VI &par, VE &cycle ) {
        FOR(i,0,adj[rt].size()) {
            edge &e = adj[rt][i];
            if( e.cap == 0 || e.cost >= 0 ) continue;
            size_t v = e.t;
            if( par[v] < N ) { // found a negative cycle!
                size_t fr = 0; while( cycle[fr].s != v ) ++fr;
                cycle = VE( cycle.begin()+fr, cycle.end() );
                cycle.push_back(e);
                return true;
            }
            else if( par[v] == N ) { // unvisited node
                par[v] = rt;  cycle.push_back(e);
                if( dfs_negcycle_r(v,par,cycle) ) return true;
                par[v] = N+1; cycle.pop_back();
            }
        }
        return false;
    }
    bool dfs_negcycle( VE &cycle ) {
        cycle.clear();   VI par(N,N);
        FOR(v,0,N) if( par[v] == N && dfs_negcycle_r(v,par,cycle) ) return
         ↪   true;
        return false;
    }
    void CancelNegativeCycles() { // only if the graph has negative cycles
        VE cycle;
        while( dfs_negcycle(cycle) )
            Augment(cycle);
    }
    /* The following is unnecessary if the graph is guaranteed to have no
     * negative-cost edges with positive capacity before MCMF is run. */
    void FixNegativeEdges( size_t SRC ) {
        VT W(N);   VI P(N,N);   P[ SRC ] = 0;
        FOR(kk,0,N-1) {
            FOR(v,0,N) FOR(i,0,adj[v].size()) {
                if( adj[v][i].cap == 0 ) continue;
                size_t u = adj[v][i].t;  T w = adj[v][i].cost;
                if( P[u] == N || W[v]+w < W[u] ) {
                    W[u] = W[v]+w;
                    P[u] = v;
                }
            }
        }
        ApplyPotential( W );
    }
    /* The following form the crux of min-cost max-flow, unless you go with
     ↪   a
     * pure cycle-canceling approach by precomputing a maxflow. */
    void shortest_paths( size_t S, VE &P, VT &W ) {
        DijkPQ Q;  P = VE(N);  W = VT(N,0); // DO init everything to 0!
        FOR(i,0,N) P[i].s = N;    edge x; x.s = x.t = S;
        Q.push( DijkP(0,0) );  VE trv;  trv.push_back(x);
        while( !Q.empty() ) {
            T wt = Q.top().first;   edge e = trv[Q.top().second];
            size_t v = e.t;         Q.pop();
            if( P[v].s != N ) continue;
            W[v] = wt;   P[v] = e;
            FOR(i,0,adj[v].size()) {
                edge &f = adj[v][i];
                if( f.cap == 0 ) continue;
                Q.push( DijkP( W[v]+f.cost, trv.size() ) );
                trv.push_back(f);
            }
        }
    }
    // Note that this returns the total *maximum flow*, not its cost. Use
    // "Cost()" after calling this for that.
    T ComputeMinCostMaxFlow( size_t SRC, size_t DST ) {
        compile_edges();  // we have to do this after all edges are added
        CancelNegativeCycles();     // Only if necessary!
        FixNegativeEdges( SRC );    // Ditto!
```

```
        T flow = 0;  VE P;  VT W;   shortest_paths( SRC, P, W );
        while( P[DST].s != N ) { // while there is a path S->T
            VE ap;
            for( size_t v = DST; v != SRC; v = P[v].s ) ap.push_back(P[v]);
            ap = VE( ap.rbegin(), ap.rend() ); // I love C++ sometimes
            flow += Augment( ap );
            ApplyPotential( W ); // This eliminates negative cycles from ^
            shortest_paths( SRC, P, W );
        }
        return flow;
    }
    T Cost() {
        T c = 0;
        FOR(v,0,N) FOR(i,0,adj[v].size()) {
            edge &e = adj[v][i];
            c += e.flow * (e.cost - pot[e.s] + pot[e.t]);
        }
        return c;
    }
};
```

SCC.cc

```
// An implementation of Kosaraju's algorithm for strongly-connected
 ↪   components
// This includes code which constructs a "meta" graph with one node per SCC.
#define FOR(v,l,u) for( size_t v = l; v < u; ++v )
typedef vector<size_t> VI;
typedef vector<VI>     VVI;
typedef vector<bool>   VB;
struct graph {
    size_t N;
    VVI    A; // Adjacency lists.
    VVI    B; // Reversed adjacency lists.
    VI     scc;    // scc[i] is the component to which i belongs
    size_t n_sccs; // the number of components
    graph( size_t n ) : N(n), A(n), B(n), scc(n) {}
    void add_edge( size_t s, size_t t ) {
        A[s].push_back(t);
        B[t].push_back(s);
    }
    bool has_edge( size_t s, size_t t ) { // only for compute_scc_graph
        FOR(i,0,A[s].size()) if( A[s][i] == t ) return true;
        return false;
    }
    void dfs_order( size_t rt, VB &Vis, VI &order ) {
        Vis[rt] = true;
        FOR(i,0,A[rt].size()) {
            size_t v = A[rt][i];
            if( Vis[v] ) continue;
            dfs_order( v, Vis, order );
        }
        order.push_back(rt);
    }
    void dfs_label( size_t rt, VB &Vis, size_t lbl, VI &out ) {
        Vis[rt] = true;
        out[rt] = lbl;
        FOR(i,0,A[rt].size()) {
            size_t v = A[rt][i];
            if( Vis[v] ) continue;
            dfs_label( v, Vis, lbl, out );
        }
    }
    void compute_sccs() {
        VB visited(N,false);   VI order;
        FOR(v,0,N) if( !visited[v] ) dfs_order(v, visited, order);
        swap(A,B);
        visited = VB(N,false); n_sccs = 0;
        FOR(i,0,N) {
            size_t v = order[N-1-i];
            if( !visited[v] ) dfs_label(v, visited, n_sccs++, scc);
        }
        swap(A,B);
    }
    void compute_scc_graph( graph &H ) {
        H = graph(n_sccs);
        FOR(v,0,N) {
            FOR(i,0,A[v].size()) {
                size_t u = A[v][i];
                size_t vv = scc[v], uu = scc[u];
                if( vv != uu && !H.has_edge(vv,uu) )
                    H.add_edge(vv,uu);
            }
        }
    }
};
```

UnionFind.cc

```
// UnionFind data structure that implements
// path compression, stolen from stanfordacm
// should not stackoverflow in correcty configured environment
// (ulimit -s BIGNUMBER)

typedef vector<size_t> VI;

// Union find is now a vector of integers, C[i] = parent(i).
// Initialize with C[i] = i

int find(VI &C, size_t x) { return (C[x] == x) ? x : C[x] = find(C, C[x]); }
void merge(VI &C, size_t x, size_t y) { C[find(C, x)] = find(C, y); }
```

Kruskal.cc

```
// Kruskal's algorithm to return MST using
// Union-Find data structure.

typedef pair<size_t, size_t> ii;
typedef pair<double, ii> dii;
typedef vector<dii> vdii;

// edges is a list of all edges in the graph, n is number
// of vertices in the graph
double kruskal(vdii &edges, size_t n) {
    sort(edges.begin(), edges.end());

    VI uf(n);
    for (size_t i = 0; i < n; ++i)
        uf[i] = i;

    double cost = 0;
    for (size_t i = 0; i < edges.size(); ++i) {
        size_t u = edges[i].second.first;
        size_t v = edges[i].second.second;

        if (find(uf, u) != find(uf, v)) {
            merge(uf, u, v);
            cost += edges[i].first;
            // if MST edges used are needed, add them here
        }
    }

    return cost;
}
```

LCA.cc

```
// Lowest Common Ancestor on a weighted, rooted tree.
// Solves UVa Online Judge "AntsColony" in 2.04 seconds.
// O(n log n) time and space precomputation, O(log n) LCA query.

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef pair<int, int> II;
typedef vector<II> VII;
typedef vector<VII> VVII;

// computes an integer upper bound of the binary logarithm of n
int log2(int n) {
    int j;
    for (j = 0; (1 << j) < n; ++j);
    return j;
}

// computes the distance from the root to every node (not necessary for LCA)
// and the level of each node in the tree (necessary for LCA)
// expects an adjacency list of the tree, where each entry is (idx, weight)
void get_dist(VVII &tree, VI &dist, VI &L) {
    stack<int> s;
    s.push(0);
    dist[0] = 0;
    L[0] = 0;
    while (!s.empty()) {
        int cur = s.top(); s.pop();
        for (int i = 0; i < tree[cur].size(); ++i) {
            dist[tree[cur][i].first] = dist[cur] + tree[cur][i].second;
            s.push(tree[cur][i].first);
            L[tree[cur][i].first] = L[cur]+1;
        }
    }
}

// uses dynamic programming to preprocess the P table
void preprocess(VVI &P, int N) {
    for (int j = 1; (1 << j) < N; ++j) {
        for (int i = 0; i < N; ++i) {
            if (P[i][j-1] != -1) {
                P[i][j] = P[P[i][j-1]][j-1];
            }
        }
    }
}

// comptues the LCA of p and q, given P, L, and N
int LCA(int p, int q, VVI &P, VI &L, int N) {
    if (L[p] < L[q])
        swap(p, q);

    for (int i = log2(N)-1; i >= 0; --i) {
        if (P[p][i] != -1 && L[P[p][i]] >= L[q])
            p = P[p][i];
    }

    if (p == q)
        return p;

    for (int i = log2(N)-1; i >= 0; --i) {
        if (P[p][i] != P[q][i]) {
            p = P[p][i], q = P[q][i];
        }
    }

    return P[p][0];
}
```

FloatCompare.cc

```
// Short function for comparing floating point numbers.

const double EPS_ABS    = 1e-10;  // for values near 0.0. Keep small.
const double EPS_REL    = 1e-8;   // for values NOT near 0.0. Balance.

bool feq( double a, double b ) {
    double d = fabs(b-a);
    if( d <= EPS_ABS ) return true;
    if( d <= max(fabs(a),fabs(b))*EPS_REL ) return true;
    return false;
}

bool flt( double a, double b ) {
    return !feq(a,b) && a < b;
}
```

Vector.cc

```
// A simple library used elsewhere in the notebook.
// Provides basic vector/point operations.
typedef double T;

struct Pt {
    T x, y;
    Pt() {}
    Pt( T x, T y ) : x(x), y(y) {}
    Pt( const Pt &h ) : x(h.x), y(h.y) {}
};

Pt operator + ( const Pt &a, const Pt &b ) { return Pt(a.x+b.x, a.y+b.y); }
Pt operator - ( const Pt &a, const Pt &b ) { return Pt(a.x-b.x, a.y-b.y); }
Pt operator * ( const T s, const Pt &a )   { return Pt(s*a.x, s*a.y); }
Pt operator * ( const Pt &a, const T s )   { return s*a; }
Pt operator / ( const Pt &a, const T s )   { return Pt(a.x/s,a.y/s); }
// Note the kind of division that occurs when using integer types.
// Use rationals if you want this to work right.
bool operator == ( const Pt &a, const Pt &b ) {
    return feq(a.x,b.x) && feq(a.y,b.y);
}
bool operator != ( const Pt &a, const Pt &b ) { return !(a == b); }

T dot( const Pt &a, const Pt &b )   { return a.x*b.x + a.y*b.y; }
T cross( const Pt &a, const Pt &b ) { return a.x*b.y - a.y*b.x; }
T norm2( const Pt &a )              { return a.x*a.x + a.y*a.y; } //
 ↪   dot(a,a)
T norm( const Pt &a )               { return sqrt(a.x*a.x + a.y*a.y); }
T dist2( const Pt &a, const Pt &b ) { // dot(a-b,a-b)
    T dx = a.x - b.x, dy = a.y - b.y;
    return dx*dx + dy*dy;
}
T dist( const Pt &a, const Pt &b )  { // sqrt(dot(a-b,a-b))
    T dx = a.x - b.x, dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}

bool lex_cmp_xy( const Pt &lhs, const Pt &rhs ) {
    if( !feq(lhs.x,rhs.x) ) return lhs.x < rhs.x;
    if( !feq(lhs.y,rhs.y) ) return lhs.y < rhs.y;
    return false;
}
```

PlaneGeometry.cc

```
// Some routines for basic plane geometry.
// Depends on Vector.cc.
typedef vector<Pt> VP;

int isLeft( Pt a, Pt b,  Pt c ) {
    T z = cross(b-a,c-a);
    if( feq(z,0) ) return  0; // c is on the line ab
    else if( z > 0 )    return  1; // c is left of the line ab
    else                return -1; // c is right of the line ab
}
Pt RotateCCW90( Pt p ) { return Pt(-p.y,p.x); }
Pt RotateCW90( Pt p )  { return Pt(p.y,-p.x); }
Pt RotateCCW( Pt p, T t ) { // This only makes sense for T=double
    return Pt(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}
// Project the point c onto line ab
// This assumes a != b, so check that first.
Pt ProjectPointLine( const Pt &a, const Pt &b,  const Pt &c ) {
    return a + (b-a) * dot(b-a,c-a)/norm2(b-a);
}
// "Project" the point c onto segment ab
// Nicely paired with dist: dist(c, ProjectPointSegment(a,b,c))
Pt ProjectPointSegment( Pt a, Pt b,  Pt c ) {
    T r = dist2(a,b);
    if( feq(r,0) ) return a;
    r = dot(c-a, b-a)/r;
    if( r < 0 ) return a;
    if( r > 1 ) return b;
    return a + (b-a)*r;
}
// Compute the distance between point (x,y,z) and plane ax+by+cz=d
T DistancePointPlane( T x, T y, T z,   T a, T b, T c, T d ) {
    return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}
// Decide if lines ab and cd are parallel.
// If a=b or c=d, then this will return true.
bool LinesParallel( Pt a, Pt b, Pt c, Pt d ) {
    return feq( cross(b-a,c-d), 0 );
}
// Decide if lines ab and cd are the same line
// If a=b and c=d, then this will return true.
// If a=b xor c=d, (wlog a=b), then this is true iff a is on cd.
bool LinesColinear( Pt a, Pt b, Pt c, Pt d ) {
    return LinesParallel(a,b, c,d)
        && isLeft(a,b, c) == 0
        && isLeft(c,d, a) == 0; // to make a=b, c=d cases symmetric
}
// Determine if the segment ab intersects with segment cd
// Use line-line intersection (below) to find it.
// This *will* do the right thing if a=b, c=d, or both!
bool SegmentsIntersect( Pt a, Pt b,   Pt c, Pt d ) {
    if(  LinesColinear(a,b, c,d)  ) {
        if( a==c || a==d || b==c || b==d ) return true;
        if( dot(a-c,b-c) > 0 && dot(a-d,b-d) > 0 && dot(c-b,d-b) > 0 )
            return false;
        return true;
    }
    if( isLeft(a,b, d) * isLeft(a,b, c) > 0 ) return false;
    if( isLeft(c,d, a) * isLeft(c,d, b) > 0 ) return false;
    return true;
}
// Determine if c is on the segment ab
bool PointOnSegment( Pt a, Pt b,  Pt c )
    { return SegmentsIntersect(a,b,c,c); }
// Compute the intersection of lines ab and cd.
// ab and cd are assumed to be *NOT* parallel
Pt ComputeLineIntersection( Pt a, Pt b,   Pt c, Pt d ) {
    b=b-a; d=d-c; c=c-a; // translate to a, set b,d to directions
    return a + b*cross(c,d)/cross(b,d); // solve s*b = c + t*d by Cramer
}
// Compute the center of the circle uniquely containing three points.
// It's assume the points are *NOT* colinear, so check that first.
Pt ComputeCircleCenter(Pt a, Pt b, Pt c) {
    b=(b-a)/2;   c=(c-a)/2; // translate to a=origin, shrink to midpoints
    return a+ComputeLineIntersection(b,b+RotateCW90(b),c,c+RotateCW90(c));
}
// Compute intersection of line ab with circle at c with radius r.
// This assumes a!=b.
VP CircleLineIntersection( Pt a, Pt b, Pt c, T r ) {
    VP ret;
    b = b-a;   a = a-c;   // translate c to origin, make b the direction
    T A = dot(b,b);       // Let P(t) = a + t*b, and Px, Py projections
    T B = dot(a,b);       // Solve Px(t)^2 + Py(t)^2 = r^2
    T C = dot(a,a) - r*r; // Get A*t^2 + 2B*t + C = 0
    T D = B*B - A*C;      // 4*D is the discriminant^
    if( flt(D,0) ) return ret;
    D = sqrt( max((T)0,D) );
    ret.push_back( c+a + b*(-B + D)/A );
    if( feq(D,0) ) return ret;
    ret.push_back( c+a + b*(-B - D)/A );
    return ret;
}
// Compute intersection of circle at a with radius r
// with circle at b with radius s.
// This assumes the circles are distinct, ie (a,r)!=(b,s)
VP CircleCircleIntersection( Pt a, T r, Pt b, T s ) {
    VP ret;
    T d = dist(a, b);
    if( d > r+s || d+min(r,s) < max(r,s) ) return ret; // empty
    T x = (d*d-s*s+r*r)/(2*d);    // The rest of this is magic.
    T y = sqrt(r*r-x*x);          // (It's actually basic geometry.)
    Pt v = (b-a)/d;
    ret.push_back(a+v*x + RotateCCW90(v)*y);
    if( !feq(y,0) )
        ret.push_back(a+v*x - RotateCCW90(v)*y);
    return ret;
}
```

Polygon.cc

```
// Basic routines for polygon-related stuff.
// Uses Vector.cc and PlaneGeometry.cc.
// Polygons are just vector<Pt>'s.
#define FOR(v,l,u) for( size_t v = l; v < u; ++v )
typedef vector<Pt> VP;

// These generalize to higher-dimensional polyhedra, provided you represent
// them as a collection of facets
// Just replace "cross" with the suitable determinant, and adjust any
 ↪   scaling
// factors.
T ComputeSignedArea( const VP &p ) {
    T area = 0;
    for( size_t i = 0; i < p.size(); i++ ) {
        size_t z = (i + 1) % p.size();
        area += cross( p[i], p[z] );
    }
    return area / 2.0;
}
T ComputeArea( const VP &p ) {
    return fabs(ComputeSignedArea(p));
}
T ComputePerimeter( const VP &p ) {
    T perim = 0.0;
    for( size_t i = 0; i < p.size(); ++i )
        perim += dist(p[i], p[(i+1) % p.size()]);
    return perim;
}
Pt ComputeCentroid( const VP &p ) {
    Pt c(0,0); T scale = 6.0 * ComputeSignedArea(p);
    for( size_t i = 0; i < p.size(); i++ ) {
        size_t j = (i + 1) % p.size();
        c = c + cross(p[i],p[j]) * (p[i]+p[j]);
    }
    return c / scale;
}
bool IsSimple( const VP &p ) {
    for( size_t i = 0; i < p.size(); ++i )
    for( size_t k = i+1; k < p.size(); ++k ) {
        size_t j = (i + 1) % p.size();
        size_t l = (k + 1) % p.size();
        if( i == l || j == k ) continue;
        if( SegmentsIntersect(p[i],p[j], p[k],p[l]) )
            return false;
    }
    return true;
}
// Determine the winding number of a point. This is the number of
// times the polygon goes around the given point.
// It is 0 exactly when the point is outside.
// A signed type is used intermediately so that we don't have to
// detect CW versus CCW, but the absolute value is taken in the end.
// If q is *on* the polygon, then the results are not well-defined,
// since it depends on whether q is on an "up" or "down" edge.
size_t WindingNumber( const VP &p, Pt q) {
    int wn = 0;   vector<int> state(p.size()); // state decides up/down
    FOR(i,0,p.size())
        if( feq(p[i].y, q.y) )  state[i] =  0; // break ties later
        else if( p[i].y < q.y ) state[i] = -1; // we'll use nearest
        else                    state[i] =  1; // neighbor (either)
    FOR(i,1,p.size())   if( state[i] == 0 ) state[i] = state[i-1];
    if( state[0] == 0 ) state[0] = state.back();
    FOR(i,1,p.size())   if( state[i] == 0 ) state[i] = state[i-1];
    FOR(i,0,p.size()) {
        size_t z = (i + 1) % p.size();
        if( state[z] == state[i] ) continue; // only interested in changes
        else if( state[z] == 1 && isLeft(p[i],p[z],q) > 0 ) ++wn;
        else if( state[i] == 1 && isLeft(p[i],p[z],q) < 0 ) --wn;
    }
    return (size_t)(wn < 0 ? -wn : wn);
}
// A complement to the above.
bool PointOnPolygon( const VP &p, Pt q ) {
    for( size_t i = 0; i < p.size(); i++ ) {
        size_t z = (i + 1) % p.size();
        if( PointOnSegment(p[i],p[z],q) )
```

```
            return true;
    }
    return false;
}
// Convex hull.
// This *will* modify the given VP. To save your points, do
// {   VP hull(p.begin(),p.end());    ConvexHull(hull);   }
// This *will* keep redundant points on the polygon border.
// To ignore those, change the isLeft's < and > to <= and >=.
void ConvexHull( VP &Z ) {
    sort( Z.begin(), Z.end(), lex_cmp_xy);
    Z.resize( unique(Z.begin(),Z.end()) - Z.begin() );
    if( Z.size() < 2 ) return;
    VP up, dn;
    for( size_t i = 0; i < Z.size(); i++ ) {
        while(up.size() > 1 && isLeft(up[up.size()-2],up.back(),Z[i]) > 0)
            up.pop_back();
        while(dn.size() > 1 && isLeft(dn[dn.size()-2],dn.back(),Z[i]) < 0)
            dn.pop_back();
        up.push_back(Z[i]);
        dn.push_back(Z[i]);
    }
    Z = dn;
    for( size_t i = up.size() - 2; i >= 1; i-- ) Z.push_back(up[i]);
}
// Implementation of Sutherland-Hodgman algorithm:
// https://en.wikipedia.org/wiki/Sutherland-Hodgman_algorithm
// Computes the intersection of polygon subject and polygon clip.
// Polygons points must be given in clockwise order. Clip must be convex.
// May return repeated points, especially if intersection is a single point
 ↪   or line segment.
// If no intersection occurs, will return an empty vector.
// Undefined behavior if intersection consists of multiple polygons.
VP ConvexClipPolygon( const VP &subject, const VP &clip ) {
    VP output = subject;
    for (size_t i = 0; i < clip.size(); ++i) {
        size_t ip1 = (i+1)%clip.size();
        Pt EdgeStart = clip[i];
        Pt EdgeEnd = clip[ip1];
        VP input = output;
        output.clear();
        Pt S = input.back();
        for (size_t j = 0; j < input.size(); ++j) {
            Pt E = input[j];
            if (isLeft(EdgeStart, EdgeEnd, E) <= 0) {
                if (isLeft(EdgeStart, EdgeEnd, S) > 0) {
                    output.push_back(ComputeLineIntersection(EdgeStart,
                     ↪   EdgeEnd,  S, E));
                }
                output.push_back(E);
            }
            else if (isLeft(EdgeStart, EdgeEnd, S) <= 0) {
                output.push_back(ComputeLineIntersection(EdgeStart, EdgeEnd,
                 ↪   S, E));
            }
            S = E;
        }
    }
    return output;
}
```

KDtree.cc

```
// Fully dynamic n-dimensional sledgehammer kd-tree.
// Constructs 100,000 point kd tree in about 2 seconds
// Handles 100,000 2D NN searches on 100,000 points in about 2 seconds
// Handles 1,000 2D range queries of ranges ~O(sqrt(n)) points
// on 50,000 points in about 2 seconds
// May degenerate when points are not uniformly distributed
// I hope you don't implement the whole thing.

// use to change data structure of pts in kdtree
typedef int T;
typedef vector<T> VT;
typedef vector<VT*> VVT;
typedef vector<VVT> VVVT;
struct kdnode {
    size_t d;
    kdnode *left;
    kdnode *right;

    // number of alive nodes in subtree rooted at this node,
    // including this node if alive
    size_t nAlive;

    // number of flagged dead nodes in subtree rooted at this node,
    // including this node if dead
    size_t nDead;

    // is flagged or not
    bool isAlive;
    VT *pt;

    // computes distance in n-dimensional space
    double dist(VT &pt1, VT &pt2) {
        double retVal = 0;

        for (size_t i = 0; i < pt1.size(); ++i) {
            retVal += (pt1[i]-pt2[i]) * (pt1[i]-pt2[i]);
        }

        return sqrt(retVal);
    }

    // returns closer point to qpt
    VT * minPt(VT &qpt, VT *pt1, VT *pt2) {
        if (pt1 == NULL) return pt2;
        if (pt2 == NULL) return pt1;
        return dist(*pt1, qpt) < dist(*pt2, qpt) ? pt1 : pt2;
    }

    // find median based on chosen algorithm
    T findMed(VVT &pts, size_t d) {
        VT arr(pts.size());

        for (size_t i = 0; i < pts.size(); ++i)
            arr[i] = (*pts[i])[d];

        sort(arr.begin(), arr.end());
        return arr[arr.size()/2];
    }

    void printPt(VT &pt) {
        for (size_t i = 0; i < pt.size(); ++i)
            cout << pt[i] << " ";
    }

    // intersects orthogonal region with left or right
    // of orthogonal halfspace on dth dimension
    VT region_intersect(VT region, T line, size_t d, bool goLeft) {
        if (goLeft) {
            region[d*2+1] = line;
        }
        else {
            region[d*2] = line;
        }

        return region;
    }

    // returns true iff the entire region is contained in the given range
    bool region_contained(VT &region, VT &range) {
        for (size_t i = 0; i < region.size(); ++i) {
            if (i % 2 == 0) {
                if (region[i] < range[i])
                    return false;
            }
            else {
                if (region[i] > range[i])
                    return false;
            }
        }

        return true;
    }

    // returns true if point is in range
    bool pt_contained(VT *pt, VT &range) {
        for (size_t i = 0; i < pt->size(); ++i) {
            if ((*pt)[i] < range[i*2] || (*pt)[i] > range[i*2+1])
                return false;
        }

        return true;
    }

    // creates "infinite" region, unbounded on all dimensions
    VT infRegion(size_t d) {
        VT region(d*2);

        for (size_t i = 0; i < region.size(); ++i) {
            if (i % 2 == 0)
                region[i] = -INF;
            else
                region[i] = INF;
        }

        return region;
    }

    void build_tree(VVT &pts, size_t d, size_t num_d) {
        pt = NULL;
        this->d = d;
        nAlive = pts.size();
        nDead = 0;
        isAlive = true;

        VVT leftV, rightV;
        T med = findMed(pts, d);

        for (size_t i = 0; i < pts.size(); ++i) {
            if ((*pts[i])[d] == med && pt == NULL)
                pt = pts[i];
            else if ((*pts[i])[d] <= med)
                leftV.push_back(pts[i]);
            else
                rightV.push_back(pts[i]);
        }

        left = leftV.empty() ? NULL : new kdnode(leftV, (d+1)%num_d, num_d);
        right = rightV.empty() ? NULL : new kdnode(rightV, (d+1)%num_d,
         ↪   num_d);
    }

    // constructs kd tree
    kdnode(VVT &pts, size_t d, size_t num_d) {
        build_tree(pts, d, num_d);
    }

    // adds pt to tree
    void addPt(VT *newPt) {
        ++nAlive;

        bool goLeft = (*newPt)[d] <= (*pt)[d];
        kdnode *child = goLeft ? left : right;
        size_t childCt = (child == NULL ? 0 : child->nAlive) + 1;

        // rebuild
        if (childCt > (1+ALPHA)/2 * nAlive) {
            VVT allPts;
            addPtToResult(allPts);
            allPts.push_back(newPt);

            delete left; delete right;

            build_tree(allPts, d, pt->size());
        }
        else if (child == NULL) {
            // add node
            VVT ptV(1, newPt);
```

```
            if (goLeft)
                left = new kdnode(ptV, (d+1)%pt->size(), pt->size());
            else
                right = new kdnode(ptV, (d+1)%pt->size(), pt->size());
        }
        else {
            // recurse
            child->addPt(newPt);
        }
    }

    // deletes existing point from kd-tree, rebalancing if necessary
    // returns the number of dead nodes removed from this subtree,
    // and bool for if pt found both are necessary in this implementation
    // to retain proper balancing invariants
    pair<size_t, bool> deletePt(VT *oldPt) {
        ++nDead;
        --nAlive;

        // need to reconstruct - last part is to avoid
        // an empty tree construction. Will get picked up by parent later
        if (nAlive < (1.0-ALPHA) * (nAlive + nDead) && nAlive > 0) {
            VVT allPts;
            addPtToResult(allPts);

            bool found = false;
            for (size_t i = 0; i < allPts.size(); ++i) {
                if (*allPts[i] == *oldPt) {
                    found = true;
                    allPts.erase(allPts.begin() + i);
                    break;
                }
            }

            delete left; delete right;

            size_t deadRemoved = nDead;

            build_tree(allPts, d, pt->size());

            return make_pair(deadRemoved, found);
        }
        else if (*pt == *oldPt) { // base case, point found
            isAlive = false;

            return make_pair(0, true);
        }
        else {
            bool goLeft = (*oldPt)[d] <= (*pt)[d];
            kdnode *child = goLeft ? left : right;

            size_t deadRemoved = 0;
            bool found = false;
            if (child != NULL) {
                // recurse
                pair<size_t, bool> result = child->deletePt(oldPt);
                deadRemoved = result.first;
                found = result.second;
            }

            // point may not have been found
            if (!found)
                ++nAlive;

            nDead -= deadRemoved;

            return make_pair(deadRemoved, found);
        }
    }

    // returns points in orthogonal range in O(n^((d-1)/d) + k)
    // where k is the number of points returned
    VVT range_query(VT &range) {
        VVT result;
        int dummy = 0;
        VT region = infRegion(pt->size());

        range_query(range, region, result, dummy, false);

        return result;
    }

    // counts number of queries in range, runs in O(n^((d-1)/d))
    int count_query(VT &range) {
        int numPts = 0;
        VVT dummy;
        VT region = infRegion(pt->size());

        range_query(range, region, dummy, numPts, true);

        return numPts;
    }

    void range_query(VT &range, VT &region, VVT &result, int &numPts, bool
     ↪   count) {
        if (region_contained(region, range)) {
            if (count)
                //by only adding size, we get rid of parameter
                //k in output sensitive O(n^((d-1)/d) + k) analysis
                numPts += nAlive;
            else
                addPtToResult(result);
            return;
        }
        else if (isAlive && pt_contained(pt, range)) {
            if (count)
                ++numPts;
            else
                result.push_back(pt);
        }

        // are parts of the range to the right of splitting line?
        if ((*pt)[d] <= range[d*2+1] && right != NULL) {
            VT newRegion = region_intersect(region, (*pt)[d], d, false);
            right->range_query(range, newRegion, result, numPts, count);
        }

        // are parts of the range to the left of splitting line?
        if ((*pt)[d] >= range[d*2] && left != NULL) {
            VT newRegion = region_intersect(region, (*pt)[d], d, true);
            left->range_query(range, newRegion, result, numPts, count);
        }
    }

    // adds point to vector result and recursively calls addPt on children
    void addPtToResult(VVT &result) {
        if (isAlive)
            result.push_back(pt);

        if (left != NULL)
            left->addPtToResult(result);
        if (right != NULL)
            right->addPtToResult(result);
    }

    // overloaded for first call with no current best
    VT * NN(VT &qpt) {
        VT *result = NN(qpt, NULL);
        return result;
    }

    // performs NN query
    VT * NN(VT &qpt, VT *curBest) {
        bool goLeft = qpt[d] <= (*pt)[d];
        kdnode *child = goLeft ? left : right;

        if (isAlive)
            curBest = minPt(qpt, pt, curBest);

        if (child != NULL)
            curBest = child->NN(qpt, curBest);

        double curDist = curBest == NULL ? INF : dist(*curBest, qpt);

        // need to check other subtree
        if (curDist + EP > abs((*pt)[d] - qpt[d])) {
            kdnode *oppChild = goLeft ? right : left;

            if (oppChild != NULL) {
                curBest = oppChild->NN(qpt, curBest);
            }
        }

        return curBest;
    }

    // prints tree somewhat nicely
    void print_tree() {
        printf("((%c%d", isAlive ? 'A' : 'D', (*pt)[0]);

        for (size_t i = 1; i < pt->size(); ++i) {
            printf(" %d", (*pt)[i]);
        }

        cout << endl;

        if (left != NULL) left->print_tree();
        printf(", ");
        if (right != NULL) right->print_tree();

        printf(")");
    }

    ~kdnode() {
        delete left;
        delete right;
    }
};
```