Listing 1: C++ Template

```
#include <cstdio>
#include <cstring>
#include <cassert>
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <algorithm>
#include <numeric>
#include <vector>
#include <string>
#include <map>
#include <set>
#include <queue>
#include <ctime>
#include <cctype>
#include <bitset>
#include <cmath>
#include <utility>

using namespace std;

#define GOEMETRY 1
#if GOEMETRY
#include <complex>

typedef complex <double> point;
#define re(x) (point(x, 0))
#define im(x) (point(0, x))
#define pol(r, t) (re(r) * exp(im(t)))
#define rotate(z, t) ((z) * exp(im(t)))
#define trans(z, w) ((z) + (w))
#define unit(z) ((z) / abs(z))
#define X(z) (real(z))
#define Y(z) (imag(z))
#define dot(z, w) (real(conj(z) * (w)))
#define cross(z, w) (imag(conj(z) * (w)))
const double PI = acos(-1);
#endif


#if 1
#define DBG(z) cerr << #z << ": " << (z) << endl
#define NEWL() cerr << endl
#define passert(x, m) {if (!(x)) {cerr << m << "  ::  ";}
    assert(x);}
#define LINE cerr << "DEBUG LINE: " << __LINE__ << endl
#else
#define DBG(z)
#define NEWL()
#endif

#define FOR(i ,n) for (int i = 0; i < (n); ++i)
#define FORL(i, i0, n) for (int i = i0; i < (n); ++i)
#define REP(i, n) for (i = 0; i < (n); ++i)

#define IT(v) (__typeof((v).begin()))
#define mem(f, a) memset(f, a, sizeof(f))
#define all(c) (c).begin(), (c).end()
#define rall(c) (c).rbegin(), (c).rend()
#define for_each(it, v) for (__typeof((v).begin()) it = (v
    ).begin(); it != (v).end(); ++it)
#define next_int() ({int __t; scanf("%d", &__t); __t;})
#define SZ(x) ((int) (x).size())
#define sz size()
#define MP make_pair
#define PB push_back
#define SRAND() srand(time(NULL))
#define RAND(from, to) ((rand() % ((to) - (from) + 1)) + (
    from))

#define eps 1e-9
#define INF 1000000000

typedef long long int64;
```

Listing 2: Number Theory

```
const int __MOBIUS_SIZE = 1000000;
char mu[__MOBIUS_SIZE + 1];
void mobis() {
  int i, j, M = __MOBIUS_SIZE;
  mu[1] = 1;
  for(i = 1; i < M + 1; i++) {
    for (j = 2 * i; j < M + 1; j += i)
      mu[j] -= mu[i];
  }
}


const int __EULER_SIZE = 1000000;
int phi[__EULER_SIZE + 1];

void euler() {
  int i, j, M = __EULER_SIZE + 1;
  for (i = 0; i < M; i++) {
    phi[i] = i;
  }
  for (i = 2; i < M; i++) {
    if (phi[i] == i) {
      for (j = i; j < M; j += i)
        phi[j] -= phi[j] / i;
    }
  }
  for (i = 2; i < M; i++) {
    if (phi[i] == i) {
      phi[i] -= 1;
    }
  }
}

// Check if g is a primitive multiplicative root of p;
    where p is a prime
// P is the list of the prime factors in the factorization
    of phi(p)
bool is_primitive_root(int g, int p) {
  for (int i = 0; i < P.size(); ++i) {
    if (pow(g, (p - 1) / P[i]) == 1) {
      return false;
    }
  }
  return true;
}

// extended Euclidean algorithm
// solves d = ax + by , for x, y; given that d = gcd(a, b)
int ext_gcd(int a, int b, int &x, int &y) {
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }
  int d = ext_gcd(b, a % b, x, y);
  int temp = x - y * (a / b);
  x = y;
  y = temp;
  return d;
}


// Solving linear modular equation ax = b (mod n)
// Returns a vector of the valid solutions
vector <int> solve(int a, int b, int n) {
  int x, y;
  int d = ext_gcd(a, n, x, y);
  if (b % d) {
    return vector <int> ();
  }
  int x0 = (long long) x * (b / d) % n;
  x0 = (x0 + n) % n;
  int t = n / d;
  vector <int> ans(d);
  x = x0;
  for (int i = 0; i < d; ++i) {
    ans[i] = x;
    x = (x + t) % n;
  }
  return ans;
}


// Discrete logarithm such that
// g^x = a (mod p), where p is a prime, g is a primitive
    root of p
// req: sqrt function
int disc_log(int g, int a, int p) {
```

```
  int m = sqrt(p) + 1;
  map <int, int> inv;
  int _a = 1;
  for (int i = 0; i < m; ++i) {
    if (!inv.count(_a))
      inv[_a] = i;
    _a = (long long) _a * g % p;
  }
  int x = a;
  int r = pow(pow(g, m), p - 2);
  for (int i = 0; i < m; ++i) {
    if (inv.count(x)) {
      return i * m + inv[x];
    }
    x = (long long) x * r % p;
  }
}


const int __NUMDIV_SZ = 1000000;
const int __NUMSUM_SZ = 1000000;
int t[__NUMDIV_SZ + 1];
// Precompute the number of divisors for numbers
int d[__NUMSUM_SZ + 1];
// Precompute the sum of divisors for the numbers
void num_sum_of_div() {
  int i, j, M = __NUMDIV_SZ;
  for (i = 1; i < M + 1; ++i) {
    for (j = i; j < M + 1; j += i) {
      t[j]++;
    d[j] += i;
    }
  }
}


// factorize all number less than n
const int MAX = 1000000;
bool is_prime[MAX + 1];
int pr[80000]; // pi[MAX]
int primes = 0;

int p[MAX + 1][7];
char e[MAX + 1][7];
char _cnt[MAX + 1];

void factorize_all() {
  is_prime[0] = is_prime[1] = true;

  for (int i = 2; i < MAX + 1; ++i) {
    if (!is_prime[i]) {
      pr[primes++] = i;
      for (int j = i; j < MAX + 1; j += i) {
        if (j != i)
          is_prime[j] = true;
        p[j][_cnt[j]++] = (i);
        int k = j / i;
        int q;
        for (q = 0; q < _cnt[j]; ++q) {
          if (i == p[k][q])
            break;
        }
        if (q < _cnt[j])
          e[j][_cnt[j] - 1] = (1 + e[k][q]);
        else
          e[j][_cnt[j] - 1] = (1);
      }
    }
  }
}

//solve ax + by == c , d = gcd(a, b)
// X = {x + k * b / d}, Y = {y - k * a / d}
int linear_dipho(int a, int b, int c, int &x, int &y) {
  int d = ext_gcd(a, b, x, y);
  if (d == 0 || c % d != 0) {
    x = y = 0;
    return 0;
  }
  x *= c / d;
  y *= c / d;
  return d;
}
```

Listing 3: Gauss

```
//Kind of a Blur
#include <cstdio>
#include <numeric>
#include <cassert>
#include <algorithm>
#include <utility>

using namespace std;

const double eps = 1e-7;

#define Z(x) ((x) > eps || -(x) > eps)

double a[100][101];
double x[10][10];
int w, h, d, n;

void gauss(void);

int main() {
  bool first = true;
  while (true) {
    scanf("%d %d %d", &h, &w, &d);
    n = w * h;
    if (w == 0 && h == 0 && d == 0) break;
    if (!first) puts("");
    first = false;
    for (int i = 0; i < w; ++i)
      for (int j = 0; j < h; ++j)
        scanf("%lf", &x[i][j]);

    int eq = 0;
    for (int i = 0; i < w; ++i) {
      for (int j = 0; j < h; ++j) {
        for (int r = 0; r <= d; ++r) {
          for (int k = 0; k <= d - r; ++k) {
            for (int msk = 0; msk < 4; ++msk) {
              int di = r, dj = k;
              if (msk & 1) di = -di;
              if (msk & 2) dj = -dj;
              if (i + di < w && i + di >= 0 && j + dj < h
                   && j + dj >= 0) {
                a[eq][(i + di) * h + j + dj] = 1;
              }
            }
          }
        }
        int cnt = accumulate(a[eq], a[eq] + n, 0);
        a[eq][n] = cnt * x[i][j];
        ++eq;
      }
    }
    gauss();
  }
}


void gauss() {
  double k = 0;
  for(int m = 0; m < n; ++m) {
    if (!Z(a[m][m])) {
      for (int j = m + 1; j < n; ++j)
        if (Z(a[j][m])) {
          for (int k = 0; k < n + 1; ++k) {
            swap(a[j][k], a[m][k]);
          }
          break;
        }
    }
    for(int i = m + 1; i < n; ++i) {
      if (Z(a[m][m])) {
        k = a[i][m] / a[m][m];
        for(int j = m; j < n + 1; ++j) {
          a[i][j] -= a[m][j] * k;
        }
      }
    }
  }
  for(int i = n - 1; i >= 0; --i) {
    if (Z(a[i][i])) {
      a[i][n] /= a[i][i];
      for(int j = i - 1; j >= 0; --j) {
        a[j][n] -= a[i][n] * a[j][i];
```

```
        a[j][i] = 0;
      }
      a[i][i] = 1;
    }
  }
  int r = 0;
  for (int i = 0; i < w; ++i) {
    for (int j = 0; j + 1 < h; ++j) {
      printf("%8.2f", a[r++][n]);
    }
    printf("%8.2f\n", a[r++][n]);
  }
  for (int i = 0; i < 100; ++i)
    for (int j = 0; j < 101; ++j)
      a[i][j] = 0;
}
```

Listing 4: PollardRho

```
import java.math.BigInteger;
import java.security.SecureRandom;
import java.io.*;
import java.util.*;

public class Main {
    final static BigInteger ZERO = new BigInteger("0");
    final static BigInteger ONE  = new BigInteger("1");
    final static BigInteger TWO  = new BigInteger("2");
    final static SecureRandom random = new SecureRandom();
    static HashMap<BigInteger, Integer> map;
    static BigInteger divisor, c, x, xx;

    static BigInteger rho(BigInteger N) {
        c  = new BigInteger(N.bitLength(), random);
        x  = new BigInteger(N.bitLength(), random);
        xx = x;
        if (N.mod(TWO).compareTo(ZERO) == 0) return TWO;
        do {
            x  =  x.multiply(x).mod(N).add(c).mod(N);
            xx = xx.multiply(xx).mod(N).add(c).mod(N);
            xx = xx.multiply(xx).mod(N).add(c).mod(N);
            divisor = x.subtract(xx).gcd(N);
        } while((divisor.compareTo(ONE)) == 0);
        return divisor;
    }

    static void factor(BigInteger N) {
        if (N.compareTo(ONE) == 0) return;
        if (N.isProbablePrime(   20   )) {
            Integer t = map.get(N);
            if (t == null) map.put(N, 1);
            else map.put(N, t + 1);
            return;
        }
        BigInteger divisor = rho(N);
        factor(divisor);
        factor(N.divide(divisor));
    }


    public static void main(String[] args) throws
         Exception {
        BufferedReader in = new BufferedReader(new
             InputStreamReader(System.in));
        StringBuffer sb = new StringBuffer();
        while (true) {
           BigInteger N = new BigInteger(in.readLine().
                trim());
           if (N.compareTo(ZERO) == 0) break;
           map = new HashMap<BigInteger, Integer>();
           factor(N);
         BigInteger[] keys = map.keySet().toArray(new
              BigInteger[0]);
         Arrays.sort(keys);
         sb.append(keys[0] + "^" + map.get(keys[0]));
         for (int i = 1; i < keys.length; i++)
                sb.append(" " + keys[i] + "^" + map.get(
                     keys[i]));
         sb.append("\n");
        }
        System.out.print(sb);
    }
}
```

Listing 5: Geometry

```
/**
 *   ////////////////////////
 *   // GEOMETRIC ROUTINES //
 *   ////////////////////////
 *
 * CONTENTS:
 *  - struct P (point)
 *  - struct L (canonical line with integer parameters)
 *  - left turn
 *  - point inside triangle
 *  - polar angle
 *  - point inside polygon
 *  - distance from a point to a line
 *  - distance from a point to a line segment
 *  - line intersection
 *  - line segment intersection
 *  - circle through 3 points
 *  - circle of a given radius through 2 points
 *  - cut polygon (cut a convex polygon by a half-plane)
 *  - triangle area from median lengths
 *
 * LAST MODIFIED:
 *      November 30, 2004
 *
 * This file is part of my library of algorithms found
     here:
 *      http://www.palmcommander.com:8081/tools/
 * LICENSE:
 *      http://www.palmcommander.com:8081/tools/LICENSE.
     html
 * Copyright (c) 2004
 * Contact author:
 *      igor at cs.ubc.ca
 **/

/*********
 * Point *
 *********
 * A simple point class used by some of the routines below
     .
 * Anything else that supports .x and .y will work just as
 * well. There are 2 variants - double and int.
 **/
struct P { double x, y; P() {}; P( double q, double w ) :
     x( q ), y( w ) {} };
struct P { int x, y; P() {}; P( int q, int w ) : x( q ), y
     ( w ) {} };

/*******************************
 * Line with integer parameters *
 *******************************
 * Represents a line through two lattice points as an
 * implicit equation:
 *      ax + by = c
 * Stores a, b and c in lowest terms with a unique
 * representation (positive a; if a is 0, then positive b)
     .
 * Create a line by giving either (a, b, c) or a pair
 * of points, p and q (p == q is ok, but expect a == b ==
     0).
 * Perfect for testing whether 3 or more points are
     collinear -
 * simply compute lines through all pairs of points and
     store
 * them in a set or a map (see UVA184).
 * REQUIRES:
 * struct P
 * from number.cpp: gcd(int, int)
 * FIELD TESTING:
 *      Passed UVA 184
 **/
struct L
{
    int a, b, c;

    void init( int A, int B, int C )
    {
        if( A < 0 || A == 0 && B < 0 ) { A = -A; B = -B; C
             = -C; }
        int d = A ?
            gcd( gcd( abs( A ), abs( B ) ), C ) :
            ( B || C ? gcd( abs( B ), C ) : 1 );
        a = A / d;
```

```
        b = B / d;
        c = C / d;
    }

    // constructors
    L() {}
    L( int A, int B, int C ) { init( A, B, C ); }
    L( P p, P q ) { init( q.y - p.y, p.x - q.x, p.x * q.y
        - p.y * q.x ); }

    bool operator<( const L &l ) const
    {
        return a < l.a || a == l.a && ( b < l.b || b == l.
            b && c < l.c );
    }
};

/************
 * Distance *
 ************
 * Computes the distance between two points a and b.
 * #include <math.h>
 **/
double dist( double ax, double ay, double bx, double by )
{
    return sqrt( ( ax - bx ) * ( ax - bx ) + ( ay - by ) *
         ( ay - by ) );
}

/********************
 * Squared distance *
 ********************
 * ... between two points.
 */
__typeof( P().x ) dist2( P p, P q )
{
    return ( p.x - q.x ) * ( p.x - q.x ) - ( p.y - q.y ) *
         ( p.y - q.y );
}

/*************
 * Left turn *
 *************
 * Returns true iff the sequence v1->v2->v3 is a left turn
      in
 * the plane. Straight line is not a left turn (change to
     ">= -EPS").
 * #define EPS 1e-7
 **/
bool leftTurn( double x1, double y1, double x2, double y2,
     double x3, double y3 )
{
    return ( x2 - x1 ) * ( y3 - y1 ) - ( y2 - y1 ) * ( x3
        - x1 ) > EPS;
}

/*************
 * Left turn * (this one works with integers)
 *************
 * Returns true iff the sequence v1->v2->v3 is a left turn
      in
 * the plane. Straight line is not a left turn (change to
     ">= -C( EPS )").
 * #define EPS ... (1e-7 for doubles, 0 for ints)
 **/
template< class C >
bool leftTurn( C x1, C y1, C x2, C y2, C x3, C y3 )
{
    return ( x2 - x1 ) * ( y3 - y1 ) - ( y2 - y1 ) * ( x3
        - x1 ) > C( EPS );
}

/*************
 * Left turn * (for either of the two P (point) structs)
 *************
 * Returns true iff the sequence v1->v2->v3 is a left turn
      in
 * the plane. Straight line is not a left turn (change to
     ">= -C( EPS )").
 * #define EPS ...
 **/
bool leftTurn( P a, P b, P c )
{
    return ( b.x - a.x ) * ( c.y - b.y ) - ( b.y - a.y ) *
         ( c.x - b.x ) > EPS;
}

/************************
 * Point inside triangle *
 ************************
 * Returns true iff point (xx,yy) is inside the counter-
     clockwise
 * triangle (x[3],y[3])
 * REQUIRES: leftTurn()
 **/
bool pointInsideTriangle( double x[], double y[], double
    xx, double yy )
{
    return leftTurn( x[0], y[0], x[1], y[1], xx, yy )
        && leftTurn( x[1], y[1], x[2], y[2], xx, yy )
        && leftTurn( x[2], y[2], x[0], y[0], xx, yy );
}

/***************
 * Polar angle *
 ***************
 * Returns an angle in the range [0, 2*Pi) of a given
     Cartesian point.
 * If the point is (0,0), -1.0 is returned.
 * REQUIRES:
 * #include <math.h>
 * #define EPS 0.000000001 // or your choice
 * P has members x and y.
 **/
double polarAngle( P p )
{
    if( fabs( p.x ) <= EPS && fabs( p.y ) <= EPS ) return
         -1.0;
    if( fabs( p.x ) <= EPS ) return ( p.y > EPS ? 1.0 :
         3.0 ) * acos( 0 );
    double theta = atan( 1.0 * p.y / p.x );
    if( p.x > EPS ) return( p.y >= -EPS ? theta : ( 4 *
         acos( 0 ) + theta ) );
    return( 2 * acos( 0 ) + theta );
}

/************************
 * Point inside polygon *
 ************************
 * Returns true iff p is inside poly.
 * PRE: The vertices of poly are ordered (either clockwise
      or
 *      counter-clockwise.
 * POST: Modify code inside to handle the special case of
     "on an edge".
 * REQUIRES:
 * polarAngle()
 * #include <math.h>
 * #include <vector>
 * #define EPS 0.000000001 // or your choice
 **/
bool pointInPoly( P p, vector< P > &poly )
{
    int n = poly.size();
    double ang = 0.0;
    for( int i = n - 1, j = 0; j < n; i = j++ )
    {
        P v( poly[i].x - p.x, poly[i].y - p.y );
        P w( poly[j].x - p.x, poly[j].y - p.y );
        double va = polarAngle( v );
        double wa = polarAngle( w );
        double xx = wa - va;
        if( va < -0.5 || wa < -0.5 || fabs( fabs( xx ) - 2
             * acos( 0 ) ) < EPS )
        {
            // POINT IS ON THE EDGE
            assert( false );
            ang += 2 * acos( 0 );
            continue;
        }
        if( xx < -2 * acos( 0 ) ) ang += xx + 4 * acos( 0
             );
        else if( xx > 2 * acos( 0 ) ) ang += xx - 4 * acos
             ( 0 );
        else ang += xx;
    }
    return( ang * ang > 1.0 );
}
```

```
/**********************************
 * Distance from a point to a line *
 **********************************
 * Returns the distance from p to the line defined by {a,
     b}.
 * The closest point on the line is returned through (cpx,
     cpy).
 * Does not work for degenerate lines (when answer is
     undefined).
 * REQUIRES:
 * #include <math.h>
 * #define EPS ...
 * dist()
 **/
double distToLine(
    double ax, double ay,
    double bx, double by,
    double px, double py,
    double *cpx, double *cpy )
{
    //Formula: cp = a + (p-a).(b-a) / |b-a| * (b-a)
    double proj = ( ( px - ax ) * ( bx - ax ) + ( py - ay
        ) * ( by - ay ) ) /
                  ( ( bx - ax ) * ( bx - ax ) + ( by - ay
                      ) * ( by - ay ) );
    *cpx = ax + proj * ( bx - ax );
    *cpy = ay + proj * ( by - ay );
    return dist( px, py, *cpx, *cpy );
}

/*****************************************
 * Distance from a point to a line segment *
 *****************************************
 * Returns the distance from p to the line segment ab.
 * The closest point on ab is returned through (cpx, cpy).
 * Works correctly for degenerate line segments (a == b).
 * REQUIRES:
 * #include <math.h>
 * #define EPS ...
 * dist()
 * distToLine()
 **/
double distToLineSegment(
    double ax, double ay,
    double bx, double by,
    double px, double py,
    double *cpx, double *cpy )
{
    if( ( bx - ax ) * ( px - ax ) + ( by - ay ) * ( py -
        ay ) < EPS )
    {
        *cpx = ax;
        *cpy = ay;
        return dist( ax, ay, px, py );
    }

    if( ( ax - bx ) * ( px - bx ) + ( ay - by ) * ( py -
        by ) < EPS )
    {
        *cpx = bx;
        *cpy = by;
        return dist( bx, by, px, py );
    }

    return distToLine( ax, ay, bx, by, px, py, cpx, cpy );
}

/*********************
 * Line intersection *
 *********************
 * Returns the point of intersection of two lines:
 * (x[0],y[0])-(x[1],y[1]) and (x[2],y[2])-(x[3],y[3]).
 * Puts the result (x, y) into (r[0], r[1]) and returns
     true.
 * If there is no intersection, return false;
 * USED BY: circle3pts
 * #include <math.h>
 * #define EPS ...
 **/
bool lineIntersect( double x[], double y[], double r[] )
{
    double n[2]; n[0] = y[3] - y[2]; n[1] = x[2] - x[3];
    double denom = n[0] * ( x[1] - x[0] ) + n[1] * ( y[1]
        - y[0] );
    if( fabs( denom ) < EPS ) return false;
    double num = n[0] * ( x[0] - x[2] ) + n[1] * ( y[0] -
        y[2] );
    double t = -num / denom;
    r[0] = x[0] + t * ( x[1] - x[0] );
    r[1] = y[0] + t * ( y[1] - y[0] );
    return true;
}

/*********************
 * Line intersection * (P version)
 *********************
 * Returns the point of intersection of two lines:
 * (x[0],y[0])-(x[1],y[1]) and (x[2],y[2])-(x[3],y[3]).
 * Puts the result (x, y) into (r[0], r[1]) and returns
     true.
 * If there is no intersection, return false;
 * #include <math.h>
 * #define EPS ...
 **/
bool lineIntersect( P a, P b, P c, P d, P &r )
{
    P n; n.x = d.y - c.y; n.y = c.x - d.x;
    double denom = n.x * ( b.x - a.x ) + n.y * ( b.y - a.y
        );
    if( fabs( denom ) < EPS ) return false;
    double num = n.x * ( a.x - c.x ) + n.y * ( a.y - c.y )
        ;
    double t = -num / denom;
    r.x = a.x + t * ( b.x - a.x );
    r.y = a.y + t * ( b.y - a.y );
    return true;
}

/*****************************
 * Line segment intersection *
 *****************************
 * Returns true iff two line segments:
 * (x[0],y[0])-(x[1],y[1]) and (x[2],y[2])-(x[3],y[3])
 * intersect. Call lineIntersect( x, y ) to get the point
     of intersection.
 * WARNING: Does not work for collinear line segments!
 * #include <vector>
 **/
template< class T >
bool lineSegIntersect( vector< T > &x, vector< T > &y )
{
    double ucrossv1 = ( x[1] - x[0] ) * ( y[2] - y[0] ) -
        ( y[1] - y[0] ) * ( x[2] - x[0] );
    double ucrossv2 = ( x[1] - x[0] ) * ( y[3] - y[0] ) -
        ( y[1] - y[0] ) * ( x[3] - x[0] );
    if( ucrossv1 * ucrossv2 > 0 ) return false;
    double vcrossu1 = ( x[3] - x[2] ) * ( y[0] - y[2] ) -
        ( y[3] - y[2] ) * ( x[0] - x[2] );
    double vcrossu2 = ( x[3] - x[2] ) * ( y[1] - y[2] ) -
        ( y[3] - y[2] ) * ( x[1] - x[2] );
    return( vcrossu1 * vcrossu2 <= 0 );
}

/***************************
 * Circle through 3 points *
 ***************************
 * Computes the circle containing the 3 given points.
 * The 3 points are
 *      (x[0], y[0]), (x[1], y[1]) and (x[2], y[2]).
 * The centre of the circle is returned as (r[0], r[1]).
 * The radius is returned normally. If the circle is
 * undefined (the points are collinear), -1.0 is returned.
 * #include <math.h>
 * REQUIRES: lineIntersect
 * FIELD TESTING: Passed UVA 190
 **/
double circle3pts( double x[], double y[], double r[] )
{
    double lix[4], liy[4];
    lix[0] = 0.5 * ( x[0] + x[1] ); liy[0] = 0.5 * ( y[0]
        + y[1] );
    lix[1] = lix[0] + y[1] - y[0];  liy[1] = liy[0] + x[0]
        - x[1];
    lix[2] = 0.5 * ( x[1] + x[2] ); liy[2] = 0.5 * ( y[1]
        + y[2] );
    lix[3] = lix[2] + y[2] - y[1];  liy[3] = liy[2] + x[1]
        - x[2];
```

```
    if( !lineIntersect( lix, liy, r ) ) return -1.0;
    return sqrt(
        ( r[0] - x[0] ) * ( r[0] - x[0] ) +
        ( r[1] - y[0] ) * ( r[1] - y[0] ) );
}

/*******************************************
 * Circle of a given radius through 2 points *
 *******************************************
 * Computes the center of a circle containing the 2 given
 * points. The circle has the given radius. The returned
 * center is never to the right of the vector
 * (x1, y1)-->(x2, y2).
 * If this is possible, returns true and passes the center
 * through the ctr array. Otherwise, returns false.
 * #include <math.h>
 * FIELD TESTING:
 *      - Valladolid 10136: Chocolate Chip Cookies
 **/
bool circle2ptsRad( double x1, double y1, double x2,
     double y2, double r, double ctr[2] )
{
    double d2 = ( x1 - x2 ) * ( x1 - x2 ) + ( y1 - y2 ) *
         ( y1 - y2 );
    double det = r * r / d2 - 0.25;
    if( det < 0.0 ) return false;
    double h = sqrt( det );
    ctr[0] = ( x1 + x2 ) * 0.5 + ( y1 - y2 ) * h;
    ctr[1] = ( y1 + y2 ) * 0.5 + ( x2 - x1 ) * h;
    return true;
}

/***************
 * Cut Polygon *
 ***************
 * Intersects a given convex polygon with a half-plane.
      The
 * half-plane is defined as the one on the left side of
      the
 * directed line a-->b. The polygon 'poly' is modified.
 * The half-plane is considered open, so if only one
      vertex
 * of the polygon remains after the cut, it is eliminated.
 * REQUIRES:
 *      - Pt must have members x and y.
 *      - lineIntersect( Pt, Pt, Pt, Pt )
 *      - dist2( Pt, Pt )
 * FIELD TESTING:
 *      - Valladolid 137: Polygon
 *      - Valladolid 10117: Nice Milk
 **/
template< class Pt >
void cutPoly( list< Pt > &poly, Pt a, Pt b )
{
    if( !poly.size() ) return;

    // see if the last point of the polygon is inside
    bool lastin = leftTurn( a, b, poly.back() );

    // find the boundary points
    __typeof( poly.begin() ) fi = poly.end(), la = fi, fip
         = fi, lan = fi;
    for( __typeof( fi ) i = --poly.end(), j = poly.begin()
         ;
        j != poly.end(); i = j++ )
    {
        int thisin = leftTurn( a, b, *j );

        if( lastin && !thisin ) { la = i; lan = j; }
        if( !lastin && thisin ) { fi = j; fip = i; }

        lastin = thisin;
    }

    // see if we have crossed the line at all
    if( fi == poly.end() )
    {
        if( !lastin ) poly.clear();
        return;
    }

    // if we cut off a corner, insert a new point
    if( lan == fip )
    {
        poly.insert( lan, *lan );
        --lan;
    }

    // compute intersection points
    Pt r;
    lineIntersect( *la, *lan, a, b, r );
    *lan = r;
    lineIntersect( *fip, *fi, a, b, r );
    *fip = r;

    // erase the part that disappears
    __typeof( fi ) i = lan; ++i;
    while( i != fip )
    {
        if( i == poly.end() ) { i = poly.begin(); if( i ==
             fip ) break; }
        poly.erase( i++ );
    }

    // clean up duplicate points
    if( dist2( *lan, *fip ) < EPS ) poly.erase( fip );
}

/*****************************
 * Triangle Area from Medians *
 *****************************
 * Given the lengths of the 3 medians of a triangle,
 * returns the triangle's area, or -1 if it impossible.
 * WARNING: Deal with the case of zero area carefully.
 * #include <math.h>
 * FIELD TESTING:
 *      - Valladolid 10347: Medians
 **/
double triAreaFromMedians( double ma, double mb, double mc
     )
{
    double x = 0.5 * ( ma + mb + mc );
    double a = x * ( x - ma ) * ( x - mb ) * ( x - mc );
    if( a < 0.0 ) return -1.0;
    else return sqrt( a ) * 4.0 / 3.0;
}

/****************
 * Great Circle *
 ****************
 * Given two pairs of (latitude, longitude), returns the
 * great circle distance between them.
 * FIELD TESTING
 *      - Valladolid 535: Globetrotter
 **/
double greatCircle( double laa, double loa, double lab,
    double lob )
{
    double PI = acos( -1.0 ), R = 6378.0;
    double u[3] = { cos( laa ) * sin( loa ), cos( laa ) *
         cos( loa ), sin( laa ) };
    double v[3] = { cos( lab ) * sin( lob ), cos( lab ) *
         cos( lob ), sin( lab ) };
    double dot = u[0]*v[0] + u[1]*v[1] + u[2]*v[2];
    bool flip = false;
    if( dot < 0.0 )
    {
        flip = true;
        for( int i = 0; i < 3; i++ ) v[i] = -v[i];
    }
    double cr[3] = { u[1]*v[2] - u[2]*v[1], u[2]*v[0] - u
         [0]*v[2], u[0]*v[1] - u[1]*v[0] };
    double theta = asin( sqrt( cr[0]*cr[0] + cr[1]*cr[1] +
          cr[2]*cr[2] ) );
    double len = theta * R;
    if( flip ) len = PI * R - len;
    return len;
}
```

Listing 6: BIT

```
const int MAX = 100010;
int T[100020];

void update(int i, int v) {
  while (i <= MAX) {
    T[i] += v;
    i += i & -i;
```

```
  }
}
int read(int i) {
  int sum = 0;
  while (i > 0) {
    sum += T[i];
    i -= i & -i;
  }
  return sum;
}

// next 2 functions work, if the accumulated values are
    the values of a given
// sequence
void update_range(int i, int j, int v) {
  update(i, v);
  update(j + 1, -v);
}
void update_sum_element(int i, int v) {
  update(i, -v);
  update(i + 1, v);
}


int readSingle(int idx) {
  int sum = tree[idx]; // sum will be decreased
  if (idx > 0) { // special case
    int z = idx - (idx & -idx); // make z first
  idx--; // idx is no important any more, so instead y,
      you can use idx
  while (idx != z) { // at some iteration idx (y) will
      become z
    sum -= tree[idx];
    // substruct tree frequency which is between y and "
        the same path"
    idx -= (idx & -idx);
  }
  }
  return sum;
}

// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// some of them (we do not know which one)

// bitMask - initialy, it is the greatest bit of MaxVal
// bitMask store interval which should be searched
int find(int cumFre){
  int idx = 0; // this var is result of function

  while ((bitMask != 0) && (idx < MaxVal)){ // nobody
      likes overflow :)
    int tIdx = idx + bitMask; // we make midpoint of
        interval
    if (cumFre == tree[tIdx]) // if it is equal, we just
        return idx
      return tIdx;
    else if (cumFre > tree[tIdx]){
            // if tree frequency "can fit" into cumFre,
            // then include it
      idx = tIdx; // update index
      cumFre -= tree[tIdx]; // set frequency for next loop
    }
    bitMask >>= 1; // half current interval
  }
  if (cumFre != 0) // maybe given cumulative frequency
      doesn't exist
    return -1;
  else
    return idx;
}



// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// the greatest one
int findG(int cumFre){
  int idx = 0;

  while ((bitMask != 0) && (idx < MaxVal)){
    int tIdx = idx + bitMask;
    if (cumFre >= tree[tIdx]){
            // if current cumulative frequency is equal to
                cumFre,
            // we are still looking for higher index (if
                exists)
      idx = tIdx;
      cumFre -= tree[tIdx];
    }
    bitMask >>= 1;
  }
  if (cumFre != 0)
    return -1;
  else
    return idx;
}
```

Listing 7: Directed MST

```
// Reference:
// http://www.ce.rit.edu/~sjyeec/dmst.html

// Finds only the cost of directed MST.
// UVA 11183
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <cctype>
#include <cmath>
#include <cassert>
#include <algorithm>
#include <vector>
#include <iostream>
#include <sstream>
using namespace std;

int INF = 0x3fffffff;

struct UnionFind {
        int head[1024], next[1024], size[1024], cc;

        void clear(int n) {
                for (int i = 0; i < n; i++) {
                        head[i] = i;
                        next[i] = -1;
                        size[i] = 1;
                }
                cc = n;
        }

        inline int operator()(int x) const { return head[x
            ]; }

        void merge(int x, int y) {
                x = head[x];
                y = head[y];
                if (x == y) return;
                if (size[x] < size[y]) swap(x, y);

                size[x] += size[y];
                cc--;
                while (y >= 0) {
                        int yy = next[y];
                        next[y] = next[x];
                        next[x] = y;
                        head[y] = x;
                        y = yy;
                }
        }
} uf;

struct Edge { int x, y, c; };
vector<Edge> in[1024];
int N, parent[1024], parentCost[1024], answer;

// Chooses least cost incoming edge to component s
void choose(int s) {
        s = uf(s);
        parent[s] = -1;
        parentCost[s] = INF;
        if (uf(s) == uf(0)) return;
        for (int x = s; x >= 0; x = uf.next[x]) {
                for (int j = 0; j < (int)in[x].size(); j
                    ++) {
                        if (uf(in[x][j].x) != uf(s) && in[
                            x][j].c < parentCost[s]) {
                                parent[s] = in[x][j].x;
```

```
                                parentCost[s] = in[x][j].c
                                    ;
                        }
                }
        }
}

// Contracts a cycle into a single component
void contract(int s) {
        // add costs of edges in the cycle to the answer,
        // reduce costs of incoming edges
        for (int x = uf(s);;) {
                answer += parentCost[x];

                for (int a = x; a >= 0; a = uf.next[a]) {
                        for (int i = 0; i < (int)in[a].
                             size(); i++)
                                in[a][i].c -= parentCost[x
                                    ];
                }

                x = uf(parent[x]);
                if (x == uf(s)) break;
        }

        // merge cycle into a single component
        for (int x = uf(s);;) {
                int y = uf(parent[x]);
                if (uf(x) == uf(y)) break;
                uf.merge(x, y);
                x = y;
        }

        s = uf(s);
        choose(s);
}

int solve() {
        uf.clear(N);
        for (int i = 0; i < N; i++) choose(i);

        int seen[1024], tick = 1;
        memset(seen, 0, sizeof(seen));

        answer = 0;
        for (int s = 0; s < N; s++) {
                if (uf(s) != s || uf(s) == uf(0))
                        continue;

                tick++;
                for (int y = uf(s); uf(y) != uf(0);) {
                        if (seen[y] == tick) {
                                contract(y);
                                y = uf(y);
                        }

                        seen[y] = tick;
                        if (parent[y] < 0)
                                return INF;
                        y = uf(parent[y]);
                }

                for (int y = uf(s); uf(y) != uf(0);) {
                        answer += parentCost[y];
                        uf.merge(y, 0);
                        y = uf(parent[y]);
                }
        }

        return answer;
}

int main() {
        int T, M;
        scanf("%d", &T);

        for (int cs = 1; cs <= T && scanf("%d %d", &N, &M)
             == 2; cs++) {
                for (int i = 0; i < N; i++) in[i].clear();
                for (int i = 0; i < M; i++) {
                        Edge e;
                        scanf("%d %d %d", &e.x, &e.y, &e.c
                            );
                        in[e.y].push_back(e);
                }

                int res = solve();
                printf("Case #%d: ", cs);
                if (res >= INF)
                        printf("Possums!\n");
                else
                        printf("%d\n", res);
        }
}
```

Listing 8: Fast Matching

```
//fastmatching
#include <cstdio>
#include <vector>
#include <cstring>
using namespace std;
const int MAX_N = 50001;
int N, M, E;
vector<int> W[MAX_N];
bool V[MAX_N], match[MAX_N];
int D[MAX_N];
bool dfs(int i) {
  if (!V[i]) {
    V[i] = true;
    int len = W[i].size(), j, k;
    for (k = 0; k < len; k++) {
      j = W[i][k];
      if (D[j] == 0) {
        D[j] = i;
        match[i] = j;
        return true;
      }
    }
    for (k = 0; k < len; k++) {
      j = W[i][k];
      if (dfs(D[j])) {
        D[j] = i;
        match[i] = j;
        return true;
      }
    }
  }
  return false;
}
int matching() {
  bool done;
  do {
    done = true;
    memset(V, 0, sizeof V);
    for (int i = 1; i <= N; i++)
      if (match[i] == 0 && dfs(i))
        done = false;
  } while (!done);

  int n = 0;
  for (int i = 1; i <= N; ++i) {
    if (match[i]) {
      ++n;
    }
  }
  return n;
}

int main() {
  scanf("%d%d%d", &N, &M, &E);
  for (int c = 0; c < E; ++c) {
    int a, b;
    scanf("%d%d", &a, &b);
    W[a].push_back(b);
  }

    int n = matching();

  printf("%d\n", n);

/*
  for (int i = 1; i <= N; ++i) {
    if (D[i]) {
      printf("%d %d\n", i, D[i]);
    }
  }
*/
```

```
}
```

Listing 9: Max Flow (Adj. Matrix)

```
/****************
 * Maximum flow * (Dinic's on an adjacency list + matrix)
 ****************
 * Takes a weighted directed graph of edge capacities as
     an adjacency
 * matrix 'cap' and returns the maximum flow from s to t.
 *
 * PARAMETERS:
 *      - cap (global): adjacency matrix where cap[u][v]
     is the capacity
 *          of the edge u->v. cap[u][v] is 0 for non-
     existent edges.
 *      - n: the number of vertices ([0, n-1] are
     considered as vertices).
 *      - s: source vertex.
 *      - t: sink.
 * RETURNS:
 *      - the flow
 *      - prev contains the minimum cut. If prev[v] == -1,
      then v is not
 *          reachable from s; otherwise, it is reachable.
 * RUNNING TIME:
 *      - O(n^3)
 * #include <string.h>
 **/

#include <string.h>
#include <stdio.h>

// the maximum number of vertices
#define NN 1024

// adjacency matrix (fill this up)
// If you fill adj[][] yourself, make sure to include both
     u->v and v->u.
int cap[NN][NN], deg[NN], adj[NN][NN];

// BFS stuff
int q[NN], prev[NN];

int dinic( int n, int s, int t )
{
    int flow = 0;

    while( true )
    {
        // find an augmenting path
        memset( prev, -1, sizeof( prev ) );
        int qf = 0, qb = 0;
        prev[q[qb++] = s] = -2;
        while( qb > qf && prev[t] == -1 )
            for( int u = q[qf++], i = 0, v; i < deg[u]; i
                 ++ )
                if( prev[v = adj[u][i]] == -1 && cap[u][v]
                     )
                    prev[q[qb++] = v] = u;

        // see if we're done
        if( prev[t] == -1 ) break;

        // try finding more paths
        for( int z = 0; z < n; z++ ) if( cap[z][t] && prev
             [z] != -1 )
        {
            int bot = cap[z][t];
            for( int v = z, u = prev[v]; u >= 0; v = u, u
                 = prev[v] )
                bot <?= cap[u][v];
            if( !bot ) continue;

            cap[z][t] -= bot;
            cap[t][z] += bot;
            for( int v = z, u = prev[v]; u >= 0; v = u, u
                 = prev[v] )
            {
                cap[u][v] -= bot;
                cap[v][u] += bot;
            }
            flow += bot;
        }
    }

    return flow;
}

//----------------- EXAMPLE USAGE -----------------
int main()
{
    // read a graph into cap[][]
    memset( cap, 0, sizeof( cap ) );
    int n, s, t, m;
    scanf( " %d %d %d %d", &n, &s, &t, &m );
    while( m-- )
    {
        int u, v, c; scanf( " %d %d %d", &u, &v, &c );
        cap[u][v] = c;
    }

    // init the adjacency list adj[][] from cap[][]
    memset( deg, 0, sizeof( deg ) );
    for( int u = 0; u < n; u++ )
        for( int v = 0; v < n; v++ ) if( cap[u][v] || cap[
             v][u] )
            adj[u][deg[u]++] = v;

    printf( "%d\n", dinic( n, s, t ) );
    return 0;
}
```

Listing 10: Max Flow (Adj. List)

```
//unfair play (max flow, dinic, by K.A.D.R)
#include <cassert>
#include <cstdio>
#include <cstring>
#include <cmath>
#include <ctime>
#include <cstdlib>
#include <cctype>
#include <string>
#include <sstream>
#include <iostream>
#include <vector>
#include <set>
#include <queue>
#include <map>
#include <queue>
#include <algorithm>
#define FOR(i,n) for(int i = 0 ; i < n ; i++)
#define FORL(i,i0,n) for(int i = i0 ; i < n ; i++)
#define FORIT(x) for(__typeof(x.begin()) it = x.begin() ;
     it != x.end() ; it++)
#define ALL(x) x.begin(), x.end()
#define SZ(x) int(x.size())
#define LEN(x) int(x.length())
#define PB push_back
#define MP make_pair
#define FST(x) x.first
#define SEC(x) x.second
#define LL long long
#define mem(x, n) memset(x, n, sizeof(x))
#if 0
#define DBG(z) cerr << #z << ": " << (z) << endl
#else
#define DBG(z)
#endif
using namespace std;

const int MAX_E = 10000;
const int MAX_V = 2000;
const int INF = 10000000;

int ver[MAX_E], nx[MAX_E], cap[MAX_E];
int last[MAX_V], ds[MAX_V], st[MAX_V], now[MAX_V];
int edge_count, SRC, SINK, V;

inline void addedge(const int v, const int w, const int
     capacity, const int reverse_capacity) {
  ver[edge_count] = w;
  cap[edge_count] = capacity;
  nx[edge_count] = last[v];
  last[v] = edge_count++;
  ver[edge_count] = v;
  cap[edge_count] = reverse_capacity;
```

```
  nx[edge_count] = last[w];
  last[w] = edge_count++;
}

inline bool bfs() {
  FOR(i, V)
    ds[i] = -1;
  int a, b;
  a = b = 0;
  st[0] = SINK;
  ds[SINK] = 0;
  while (a <= b) {
    int v = st[a++];
    DBG(v);
    for (int w = last[v]; w >= 0; w = nx[w]) {
      if (cap[w ^ 1] > 0 && ds[ver[w]] == -1) {
        st[++b] = ver[w];
        ds[ver[w]] = ds[v] + 1;
      }
    }
  }
  return ds[SRC] >= 0;
}

int dfs(int v, int cur) {
  if (v == SINK)
    return cur;
  for (int &w = now[v]; w >= 0; w = nx[w]) {
    if (cap[w] > 0 && ds[ver[w]] == ds[v] - 1) {
      int d = dfs(ver[w], min(cur, cap[w]));
      if (d) {
        cap[w] -= d;
        cap[w ^ 1] += d;
        return d;
      }
    }
  }
  return 0;
}

inline int flow() {
  int res = 0;
  while (bfs()) {
    DBG(res);
    for (int i = 0; i < V; i++)
      now[i] = last[i];
    while (1) {
      int tf = dfs(SRC, INF);
      DBG(tf);
      if (!tf)
        break;
      res += tf;
    }
    DBG(res);
  }
  return res;
}

int points[200];

int main() {
  while (1) {
    int n, m, u, v, match = 0, win = 0;
    mem(nx, -1);
    mem(last, -1);
    edge_count = 0;
    scanf("%d", &n);
    if (n < 0)
      break;
    scanf("%d", &m);
    FOR(i, n)
      scanf("%d", &points[i]);
    win += points[n - 1];
    while (m--) {
      scanf("%d%d", &u, &v);
      if (u == n || v == n) {
        win += 2;
        continue;
      }
      addedge(n + match, --u, 2, 0);
      addedge(n + match, --v, 2, 0);
      match++;
    }
    bool can = true;
    FOR(i, n-1)
      if (points[i] >= win)
        can = false;
    if (!can) {
      printf("NO\n");
    }
    else {
      SRC = match + n;
      SINK = SRC + 1;
      V = SINK + 1;

      FOR(i, n-1)
        addedge(i, SINK, max(win - points[i], 0), 0);
      FOR(i, match)
        addedge(SRC, i + n, 2, 0);

      DBG(edge_count);

      int f = flow();
      if (f >= 2 * match)
        printf("YES\n");
      else
        printf("NO\n");
    }
  }
  return 0;
}
```

Listing 11: Min Cost Max Flow (Dense Graphs)

```
/**
 *   //////////////////////
 *   // MIN COST MAX FLOW //
 *   //////////////////////
 *
 *   Authors: Frank Chu, Igor Naverniouk
 **/

/*********************
 * Min cost max flow * (Edmonds-Karp relabelling +
     Dijkstra)
 *********************
 * Takes a directed graph where each edge has a capacity
     ('cap') and a
 * cost per unit of flow ('cost') and returns a maximum
     flow network
 * of minimal cost ('fcost') from s to t. USE THIS CODE
     FOR (MODERATELY)
 * DENSE GRAPHS; FOR VERY SPARSE GRAPHS, USE mcmf4.cpp.
 *
 * PARAMETERS:
 *      - cap (global): adjacency matrix where cap[u][v]
     is the capacity
 *          of the edge u->v. cap[u][v] is 0 for non-
     existent edges.
 *      - cost (global): a matrix where cost[u][v] is the
     cost per unit
 *          of flow along the edge u->v. If cap[u][v] ==
     0, cost[u][v] is
 *          ignored. ALL COSTS MUST BE NON-NEGATIVE!
 *      - n: the number of vertices ([0, n-1] are
     considered as vertices).
 *      - s: source vertex.
 *      - t: sink.
 * RETURNS:
 *      - the flow
 *      - the total cost through 'fcost'
 *      - fnet contains the flow network. Careful: both
     fnet[u][v] and
 *          fnet[v][u] could be positive. Take the
     difference.
 * COMPLEXITY:
 *      - Worst case: O(n^2*flow  <?  n^3*fcost)
 * FIELD TESTING:
 *      - Valladolid 10594: Data Flow
 * REFERENCE:
 *      Edmonds, J., Karp, R.  "Theoretical Improvements
     in Algorithmic
 *          Efficieincy for Network Flow Problems".
 *      This is a slight improvement of Frank Chu's
     implementation.
 **/

#include <string.h>
```

```
#include <limits.h>
using namespace std;

// the maximum number of vertices + 1
#define NN 1024

// adjacency matrix (fill this up)
int cap[NN][NN];

// cost per unit of flow matrix (fill this up)
int cost[NN][NN];

// flow network and adjacency list
int fnet[NN][NN], adj[NN][NN], deg[NN];

// Dijkstra's successor and depth
int par[NN], d[NN];        // par[source] = source;

// Labelling function
int pi[NN];

#define CLR(a, x) memset( a, x, sizeof( a ) )
#define Inf (INT_MAX/2)

// Dijkstra's using non-negative edge weights (cost +
    potential)
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra( int n, int s, int t )
{
    for( int i = 0; i < n; i++ ) d[i] = Inf, par[i] = -1;
    d[s] = 0;
    par[s] = -n - 1;

    while( 1 )
    {
        // find u with smallest d[u]
        int u = -1, bestD = Inf;
        for( int i = 0; i < n; i++ ) if( par[i] < 0 && d[i
            ] < bestD )
            bestD = d[u = i];
        if( bestD == Inf ) break;

        // relax edge (u,i) or (i,u) for all i;
        par[u] = -par[u] - 1;
        for( int i = 0; i < deg[u]; i++ )
        {
            // try undoing edge v->u
            int v = adj[u][i];
            if( par[v] >= 0 ) continue;
            if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u]
                )
                d[v] = Pot( u, v ) - cost[v][u], par[v] =
                    -u-1;

            // try edge u->v
            if( fnet[u][v] < cap[u][v] && d[v] > Pot(u,v)
                + cost[u][v] )
                d[v] = Pot(u,v) + cost[u][v], par[v] = -u
                    - 1;
        }
    }

    for( int i = 0; i < n; i++ ) if( pi[i] < Inf ) pi[i]
        += d[i];

    return par[t] >= 0;
}
#undef Pot

int mcmf3( int n, int s, int t, int &fcost )
{
    // build the adjacency list
    CLR( deg, 0 );
    for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        if( cap[i][j] || cap[j][i] ) adj[i][deg[i]++] = j;

    CLR( fnet, 0 );
    CLR( pi, 0 );
    int flow = fcost = 0;

    // repeatedly, find a cheapest path from s to t
    while( dijkstra( n, s, t ) )
    {
        // get the bottleneck capacity
        int bot = INT_MAX;
        for( int v = t, u = par[v]; v != s; u = par[v = u]
            )
            bot <?= fnet[v][u] ? fnet[v][u] : ( cap[u][v]
                - fnet[u][v] );

        // update the flow network
        for( int v = t, u = par[v]; v != s; u = par[v = u]
            )
            if( fnet[v][u] ) { fnet[v][u] -= bot; fcost -=
                bot * cost[v][u]; }
            else { fnet[u][v] += bot; fcost += bot * cost[
                u][v]; }

        flow += bot;
    }

    return flow;
}

//----------------- EXAMPLE USAGE -----------------
#include <iostream>
#include <stdio.h>
using namespace std;

int main()
{
  int numV;
  //  while ( cin >> numV && numV ) {
  cin >> numV;
    memset( cap, 0, sizeof( cap ) );

    int m, a, b, c, cp;
    int s, t;
    cin >> m;
    cin >> s >> t;

    // fill up cap with existing capacities.
    // if the edge u->v has capacity 6, set cap[u][v] = 6.
    // for each cap[u][v] > 0, set cost[u][v] to  the
    // cost per unit of flow along the edge i->v
    for (int i=0; i<m; i++) {
      cin >> a >> b >> cp >> c;
      cost[a][b] = c; // cost[b][a] = c;
      cap[a][b] = cp; // cap[b][a] = cp;
    }

    int fcost;
    int flow = mcmf3( numV, s, t, fcost );
    cout << "flow: " << flow << endl;
    cout << "cost: " << fcost << endl;

    return 0;
}
```

Listing 12: Min Cost Max Flow (Sparse Graphs)

```
/**
 *   ///////////////////////
 *   // MIN COST MAX FLOW //
 *   ///////////////////////
 *
 *   Authors: Frank Chu, Igor Naverniouk
 **/

/*********************
 * Min cost max flow * (Edmonds-Karp relabelling + fast
     heap Dijkstra)
 *********************
 * Takes a directed graph where each edge has a capacity
     ('cap') and a
 * cost per unit of flow ('cost') and returns a maximum
     flow network
 * of minimal cost ('fcost') from s to t. USE mcmf3.cpp
     FOR DENSE GRAPHS!
 *
 * PARAMETERS:
 *      - cap (global): adjacency matrix where cap[u][v]
     is the capacity
 *          of the edge u->v. cap[u][v] is 0 for non-
     existent edges.
 *      - cost (global): a matrix where cost[u][v] is the
     cost per unit
```

```
 *          of flow along the edge u->v. If cap[u][v] ==
     0, cost[u][v] is
 *          ignored. ALL COSTS MUST BE NON-NEGATIVE!
 *      - n: the number of vertices ([0, n-1] are
     considered as vertices).
 *      - s: source vertex.
 *      - t: sink.
 * RETURNS:
 *      - the flow
 *      - the total cost through 'fcost'
 *      - fnet contains the flow network. Careful: both
     fnet[u][v] and
 *          fnet[v][u] could be positive. Take the
     difference.
 * COMPLEXITY:
 *      - Worst case: O(m*log(m)*flow  <?  n*m*log(m)*
     fcost)
 * FIELD TESTING:
 *      - Valladolid 10594: Data Flow
 * REFERENCE:
 *      Edmonds, J., Karp, R.  "Theoretical Improvements
     in Algorithmic
 *          Efficieincy for Network Flow Problems".
 *      This is a slight improvement of Frank Chu's
     implementation.
 **/

#include <iostream>
using namespace std;

// the maximum number of vertices + 1
#define NN 1024

// adjacency matrix (fill this up)
int cap[NN][NN];

// cost per unit of flow matrix (fill this up)
int cost[NN][NN];

// flow network and adjacency list
int fnet[NN][NN], adj[NN][NN], deg[NN];

// Dijkstra's predecessor, depth and priority queue
int par[NN], d[NN], q[NN], inq[NN], qs;

// Labelling function
int pi[NN];

#define CLR(a, x) memset( a, x, sizeof( a ) )
#define Inf (INT_MAX/2)
#define BUBL { \
    t = q[i]; q[i] = q[j]; q[j] = t; \
    t = inq[q[i]]; inq[q[i]] = inq[q[j]]; inq[q[j]] = t; }

// Dijkstra's using non-negative edge weights (cost +
    potential)
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra( int n, int s, int t )
{
    CLR( d, 0x3F );
    CLR( par, -1 );
    CLR( inq, -1 );
    //for( int i = 0; i < n; i++ ) d[i] = Inf, par[i] =
        -1;
    d[s] = qs = 0;
    inq[q[qs++] = s] = 0;
    par[s] = n;

    while( qs )
    {
        // get the minimum from q and bubble down
        int u = q[0]; inq[u] = -1;
        q[0] = q[--qs];
        if( qs ) inq[q[0]] = 0;
        for( int i = 0, j = 2*i + 1, t; j < qs; i = j, j =
             2*i + 1 )
        {
            if( j + 1 < qs && d[q[j + 1]] < d[q[j]] ) j++;
            if( d[q[j]] >= d[q[i]] ) break;
            BUBL;
        }

        // relax edge (u,i) or (i,u) for all i;
        for( int k = 0, v = adj[u][k]; k < deg[u]; v = adj
             [u][++k] )
        {
            // try undoing edge v->u
            if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u]
                 )
                d[v] = Pot(u,v) - cost[v][par[v] = u];

            // try using edge u->v
            if( fnet[u][v] < cap[u][v] && d[v] > Pot(u,v)
                 + cost[u][v] )
                d[v] = Pot(u,v) + cost[par[v] = u][v];

            if( par[v] == u )
            {
                // bubble up or decrease key
                if( inq[v] < 0 ) { inq[q[qs] = v] = qs; qs
                     ++; }
                for( int i = inq[v], j = ( i - 1 )/2, t;
                     d[q[i]] < d[q[j]]; i = j, j = ( i - 1
                          )/2 )
                     BUBL;
            }
        }
    }

    for( int i = 0; i < n; i++ ) if( pi[i] < Inf ) pi[i]
         += d[i];

    return par[t] >= 0;
}
#undef Pot

int mcmf4( int n, int s, int t, int &fcost )
{
    // build the adjacency list
    CLR( deg, 0 );
    for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        if( cap[i][j] || cap[j][i] ) adj[i][deg[i]++] = j;

    CLR( fnet, 0 );
    CLR( pi, 0 );
    int flow = fcost = 0;

    // repeatedly, find a cheapest path from s to t
    while( dijkstra( n, s, t ) )
    {
        // get the bottleneck capacity
        int bot = INT_MAX;
        for( int v = t, u = par[v]; v != s; u = par[v = u]
             )
            bot <?= fnet[v][u] ? fnet[v][u] : ( cap[u][v]
                 - fnet[u][v] );

        // update the flow network
        for( int v = t, u = par[v]; v != s; u = par[v = u]
             )
            if( fnet[v][u] ) { fnet[v][u] -= bot; fcost -=
                 bot * cost[v][u]; }
            else { fnet[u][v] += bot; fcost += bot * cost[
                 u][v]; }

        flow += bot;
    }

    return flow;
}

//----------------- EXAMPLE USAGE -----------------
#include <iostream>
#include <stdio.h>
using namespace std;

int main()
{
  int numV;
  //  while ( cin >> numV && numV ) {
  cin >> numV;
    memset( cap, 0, sizeof( cap ) );

    int m, a, b, c, cp;
    int s, t;
    cin >> m;
    cin >> s >> t;
```

```
    // fill up cap with existing capacities.
    // if the edge u->v has capacity 6, set cap[u][v] = 6.
    // for each cap[u][v] > 0, set cost[u][v] to  the
    // cost per unit of flow along the edge i->v
    for (int i=0; i<m; i++) {
      cin >> a >> b >> cp >> c;
      cost[a][b] = c; // cost[b][a] = c;
      cap[a][b] = cp; // cap[b][a] = cp;
    }

    int fcost;
    int flow = mcmf3( numV, s, t, fcost );
    cout << "flow: " << flow << endl;
    cout << "cost: " << fcost << endl;

    return 0;
}
```

Listing 13: Suffix Arrays

```
/*
ID : omarazazy
LANG : C++
PROG : hidden
*/
#include <stdio.h>
#include <algorithm>
#include <string.h>
using namespace std;

FILE *out = fopen( "hidden.out" , "w" );

const int MAXN = 100010 , MAXLG = 20;
int N , suf[ MAXLG ][ MAXN ] , dpth;
char str[ MAXN ];

class suffix
{
  public:
  int a , b , i;
  inline bool operator < ( const suffix &x )const{return
       ((a < x.a) || ((a == x.a) && (b < x.b)));}
  inline bool operator == ( const suffix &x )const{return
       ((a == x.a) && (b == x.b));}
}L[ MAXN ];

inline int lcp( int a , int b )
{
  int ret = 0;
  for( int c = dpth - 1 ; c >= 0 && a < N && b < N ; c-- )
      if( suf[ c ][ a ] == suf[ c ][ b ] )
      {
        a += (1 << c);
        b += (1 << c);
        ret += (1 << c);
      }
  return ret;
}

int main()
{
  freopen( "hidden.in" , "r" , stdin );
  scanf( "%d" ,&N );
  for( int c = 0 ; c < N ; )
  {
    char ch = getchar();
    if( ch >= 'a' && ch <= 'z' )str[ c ++ ] = ch;
  }
  str[ N ] = '\0';
  for( int c = 0 ; c < N ; c++ )
    suf[ 0 ][ c ] = str[ c ] - 'a';
  for( int siz = 1 , ii = 1 ; 1 ; siz <<= 1 , ii++ )
  {
    for( int c = 0 ; c < N ; c++ )
    {
      L[ c ].a = suf[ ii - 1 ][ c ];
      L[ c ].b = (c + siz < N) ? suf[ ii - 1 ][ c + siz ]
           : -1;
      L[ c ].i = c;
    }
    sort( L , L + N );
    int cnt = 0;
    suf[ ii ][ L[ 0 ].i ] = 0;
    for( int c = 1 ; c < N ; c++ )
      suf[ ii ][ L[ c ].i ] = ( (L[ c ] == L[ c - 1 ]) ?
           cnt : ++cnt );
    if( cnt == N - 1 )
    {
      dpth = ii + 1;
      break;
    }
  }
  int ret = 0;
  for( int c = 1 ; c < N ; c++ )
  {
    if( lcp( c , ret ) >= N - c )
    {
      if( lcp( 0 , ret + (N - c) ) >= N - c )
      {
        if( lcp( 0 , N - c ) < N - c && suf[ dpth - 1 ][ N
             - c ] < suf[ dpth - 1 ][ 0 ] )ret = c;
      }
      else if( suf[ dpth - 1 ][ 0 ] < suf[ dpth - 1 ][ ret
           + N - c ] )ret = c;
    }
    else if( suf[ dpth - 1 ][ c ] < suf[ dpth - 1 ][ ret ]
         )ret = c;
  }
  fprintf( out , "%d\n" ,ret );
  return 0;
}
```

Listing 14: Miscellaneous

```
/**
 *   ////////////////////////////
 *   // Miscellaneous Algorithms //
 *   ////////////////////////////
 *
 * LAST MODIFIED:
 *      June 6, 2004
 *
 * This file is part of my library of algorithms found
     here:
 *      http://www.palmcommander.com:8081/tools/
 * LICENSE:
 *      http://www.palmcommander.com:8081/tools/LICENSE.
     html
 * Copyright (c) 2004
 * Contact author:
 *      igor at cs.ubc.ca
 **/

/***********
 * foreach *
 ***********
 * A handy macro for iterating through STL collections.
 */
#define foreach(i, c) for( __typeof( (c).begin() ) i = (c)
     .begin(); i != (c).end(); ++i )

/**********************
 * Print a Polynomial *
 **********************
 * Takes a list of coefficients, a largest power and a
     variable name
 * And prints the polynomial to stdout in the form of, e.g
 * 4*x^3+8*n-9
 * #include <math.h>
 **/
void printPoly( int coeffs[], int n, const char *var )
{
    bool empty = true;
    while( n-- )
    {
        if( coeffs[n] || empty && !n )
        {
            if( !empty || coeffs[n] < 0 ) printf( "%c", (
                 coeffs[n] > 0 ? '+' : '-' ) );
            if( abs( coeffs[n] ) > 1 || n == 0 ) printf( "
                 %d", abs( coeffs[n] ) );
            if( n > 0 )
            {
                if( abs( coeffs[n] ) > 1 ) printf( "*" );
                printf( "%s", var );
            }
            if( n > 1 ) printf( "^%d", n );
```

```
            empty = false;
        }
    }
}

/***************
 * Maximum sum *
 ***************
 * Given an array of integers on stdin, find the sub-array
 * [i, j] containing the largest sum, favouring longer
     ranges,
 * then smaller indices. This one only considers non-
     negative
 * sums. Best sum and range are returned through pointers.
 * #include <stdio.h>
 **/
void maximumSum( int *bestSum, int *first, int *last )
{
    int start = -1; sum = 0; best = -1; besti = -1; bestj
         = -1;
    scanf( "%d", &n );          // Size of input array
    for( i = 1; i < n; i++ )
    {
        scanf( "%d", &edge );   // ( Input array )[i - 1];
        if( start < 0 ) { if( edge < 0 ) continue; else
             start = i; }
        if( edge > 0 && sum < 0 ) { start = i; sum = 0; }
        sum += edge;
        if( edge > 0 && sum >= best )
        {
            if( sum > best || i - start > bestj - besti )
            {
                besti = start;
                bestj = i;
                best = sum;
            }
        }
    }

    *bestSum = best;
    if( best >= 0 ) { *first = besti - 1; *last = bestj; }
}

/**********************
 * Fast integer input *
 **********************
 * Reads an integer from standard input.
 * Almost equivalent to scanf( "%d", n ) (must be
     terminated
 * by a space or a '\n').
 * WARNING:
 * Valladolid now forbids unlocked IO. Remove the "
     _unlocked"
 * to get a version that is 1.5 - 2 times slower.
 *
 * #include <stdio.h>
 * USED BY: lis
 **/
void readn( register int *n )
{
    int sign = 1;
    register char c;
    *n = 0;
    while( ( c = getc_unlocked( stdin ) ) != '\n' )
    {
        switch( c )
        {
            case '-' : sign = -1;
            case ' ': goto hell;
            case '\n': goto hell;
            default: (*n) *= 10; (*n) += ( c - '0' );
                 break;
        }
    }
hell:
    (*n)*=sign;
}

/*********************************
 * Longest Increasing Subsequence *
 *********************************
 * Computes the longest increasing subsequence for a
     number
 * of problem instances, each given as a sequence of non-
     negative
 * integers, terminated by a -1. The whole input is
     terminated
 * by a -1.
 *
 * readn( int *n ) reads the next integer in the sequence.
 *
 * #include <vector>
 * #include <algorithm>
 * REQUIRES: readn (suggested)
 * COMPLEXITY: O(nlogn)
 **/
int lis()
{
    int n, prob = 0;
    vector< int > table;
    while( true )
    {
        table.clear();
        while( true )
        {
            readn( &n );        // "cin >> n;" might be
                 too slow
            if( n < 0 ) break;
            vector< int >::iterator i =
                upper_bound( table.begin(), table.end(), n
                     );
            if( i == table.end() )
                table.push_back( n );
            else
                *i = n;
        }
        if( table.empty() ) break;
        printf( "Test #%d:\n  maximum possible
             interceptions: %d\n\n",
            ++prob, table.size() );
    }
    return 0;
}

/*****************
 * Lex k-subsets *
 *****************
 * Given a set of n elements, prints all subsets of size k
     in
 * lexicographic order. Each one is displayed as a string
     of
 * 0's and 1's. For example, when n = 4, k = 2, the
 * routine prints
 *      0011
 *      0101
 *      0110
 *      1001
 *      1010
 *      1100
 * COMPLEXITY: O(n*2^n)
 **/
void lexKSubsets( int n, int k )
{
    for( int i = 0; i < ( 1 << n ); i++ )
    {
        int bits = 0;
        for( int j = 0; j < n; j++ ) if( i & ( 1 << j ) )
             bits++;
        if( bits != k ) continue;
        for( int j = n - 1; j >= 0; j-- )
            printf( "%c", ( ( i & ( 1 << j ) ) ? '1' : '0'
                 ) );
        printf( "\n" );
    }
}

/***************
 * fill string *
 ***************
 * Creates a string of n identical characters, c
 * #include <string>
 * USED BY: toRoman()
 **/
string fill( char c, int n )
{
    string s;
    while( n-- ) s += c;
    return s;
```

```
}

/******************
 * Roman numerals *
 ******************
 * Converts an integer in the range [1, 4000) to a lower
      case Roman numeral
 * #include <string>
 * REQUIRES: fill()
 **/
string toRoman( int n )
{
    if( n < 4 ) return fill( 'i', n );
    if( n < 6 ) return fill( 'i', 5 - n ) + "v";
    if( n < 9 ) return string( "v" ) + fill( 'i', n - 5 );
    if( n < 11 ) return fill( 'i', 10 - n ) + "x";
    if( n < 40 ) return fill( 'x', n / 10 ) + toRoman( n %
         10 );
    if( n < 60 ) return fill( 'x', 5 - n / 10 ) + 'l' +
        toRoman( n % 10 );
    if( n < 90 ) return string( "l" ) + fill( 'x', n / 10
        - 5 ) + toRoman( n % 10 );
    if( n < 110 ) return fill( 'x', 10 - n / 10 ) + "c" +
        toRoman( n % 10 );
    if( n < 400 ) return fill( 'c', n / 100 ) + toRoman( n
         % 100 );
    if( n < 600 ) return fill( 'c', 5 - n / 100 ) + 'd' +
        toRoman( n % 100 );
    if( n < 900 ) return string( "d" ) + fill( 'c', n /
        100 - 5 ) + toRoman( n % 100 );
    if( n < 1100 ) return fill( 'c', 10 - n / 100 ) + "m"
        + toRoman( n % 100 );
    if( n < 4000 ) return fill( 'm', n / 1000 ) + toRoman(
         n % 1000 );
    return "?";
}

/***************************
 * Milk Judge (Valladolid) *
 ***************************
 * Stop the program in one of 8 possible ways, essentially
 * giving back the 3 lowest bits of 'n', whatever n is.
 **/
void milkJudge( int n )
{
    switch( n & 7 )
    {
        case 0: exit( 0 );                                //
              Wrong Answer
        case 1: assert( false );                          //
              SIGABRT
        case 2: *( int* )( n - n ) = 0;                   //
              SIGSEGV
        case 3: n /= ( n - 3 );                           //
              SIGFPE
        case 4: while( 1 );                               //
              Time Limit Exceeded
        case 5: while( 1 ) malloc( 1024 * 1024 );         //
              Memory Limit Exceeded
        case 6: malloc( 32 * 1024 * 1024 ); while(1);     //
              32MB + TLE
        case 7: while( 1 ) printf( "." );                 //
              Output Limit Exceeded
    }
}
```

Listing 15: KMP

```
void kmpSearch() {
  int i = 0, j = 0;
  while (i < n) {
    while (j >= 0 && t[i] != p[j]) {
      j = b[j];
    }
    ++i;
    ++j;
    if (j == m) {
      report(i - j);
      j = b[j];
    }
  }
}

void kmpPreprocess() {
  int i = 0, j = -1;
  b[i] = j;
  while (i < m) {
    while (j >= 0 && p[i] != p[j]) {
      j = b[j];
    }
    ++i;
    ++j;
    b[i] = j;
  }
}
```

Listing 16: Knight Moves

```
int knightMoves(int x1, int y1, int x2, int y2) {
  int dx = abs(x2 - x1);
  int dy = abs(y2 - y1);
  int lb = (dx + 1) / 2;
  lb = max(lb, (dy + 1) / 2);
  lb = max(lb, (dx + dy + 2) / 3);
  while ((lb % 2) != ((dx + dy) % 2)) lb++;
  if (abs(dx) == 1 && dy == 0)
    return 3;
  if (abs(dy) == 1 && dx == 0)
    return 3;
  if (abs(dx) == 2 && abs(dy) == 2)
    return 4;
  return lb;
}
```

Combinatorics

Sums

$\sum_{k=0}^{n} k = n(n+1)/2$
$\sum_{k=0}^{n} k^2 = n(n+1)(2n+1)/6$
$\sum_{k=0}^{n} k^4 = (6n^5+15n^4+10n^3-n)/30$
$\sum_{k=0}^{n} x^k = (x^{n+1}-1)/(x-1)$
$1+x+x^2+\cdots = 1/(1-x)$

$\sum_{k=a}^{b} k = (a+b)(b-a+1)/2$
$\sum_{k=0}^{n} k^3 = n^2(n+1)^2/4$
$\sum_{k=0}^{n} k^5 = (2n^6+6n^5+5n^4-n^2)/12$
$\sum_{k=0}^{n} kx^k = (x-(n+1)x^{n+1}+nx^{n+2})/(x-1)^2$

Binomial coefficients

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1												
1	1	1											
2	1	2	1										
3	1	3	3	1									
4	1	4	6	4	1								
5	1	5	10	10	5	1							
6	1	6	15	20	15	6	1						
7	1	7	21	35	35	21	7	1					
8	1	8	28	56	70	56	28	8	1				
9	1	9	36	84	126	126	84	36	9	1			
10	1	10	45	120	210	252	210	120	45	10	1		
11	1	11	55	165	330	462	462	330	165	55	11	1	
12	1	12	66	220	495	792	924	792	495	220	66	12	1
	0	1	2	3	4	5	6	7	8	9	10	11	12

$\binom{n}{k} = \frac{n!}{(n-k)!k!}$

$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

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

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

$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$

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

$12! \approx 2^{28.8}$
$20! \approx 2^{61.1}$

Number of ways to pick a multiset of size k from n elements: $\binom{n+k-1}{k}$
Number of n-tuples of non-negative integers with sum s: $\binom{s+n-1}{n-1}$, at most s: $\binom{s+n}{n}$
Number of n-tuples of positive integers with sum s: $\binom{s-1}{n-1}$
Number of lattice paths from $(0,0)$ to (a,b), restricted to east and north steps: $\binom{a+b}{a}$

Multinomial theorem. $(a_1+\cdots+a_k)^n = \sum \binom{n}{n_1,\ldots,n_k} a_1^{n_1}\ldots a_k^{n_k}$, where $n_i \geq 0$ and $\sum n_i = n$.
$\binom{n}{n_1,\ldots,n_k} = M(n_1,\ldots,n_k) = \frac{n!}{n_1!\ldots n_k!}$. $M(a,\ldots,b,c,\ldots) = M(a+\cdots+b,c,\ldots)M(a,\ldots,b)$

Catalan numbers. $C_n = \frac{1}{n+1}\binom{2n}{n}$. $C_0 = 1$, $C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$. $C_{n+1} = C_n \frac{4n+2}{n+2}$.
$C_0, C_1, \ldots = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, \ldots$
C_n is the number of: properly nested sequences of n pairs of parentheses; rooted ordered binary trees with $n+1$ leaves; triangulations of a convex $(n+2)$-gon.

Derangements. Number of permutations of $n = 0, 1, 2, \ldots$ elements without fixed points is $1, 0, 1, 2, 9, 44, 265, 1854, 14833, \ldots$
Recurrence: $D_n = (n-1)(D_{n-1}+D_{n-2}) = nD_{n-1}+(-1)^n$. Corollary: number of permutations with exactly k fixed points is $\binom{n}{k}D_{n-k}$.

Stirling numbers of 1^{st} kind. $s_{n,k}$ is $(-1)^{n-k}$ times the number of permutations of n elements with exactly k permutation cycles. $|s_{n,k}| = |s_{n-1,k-1}| + (n-1)|s_{n-1,k}|$. $\sum_{k=0}^{n} s_{n,k}\, x^k = x^{\underline{n}}$

Stirling numbers of 2^{nd} kind. $S_{n,k}$ is the number of ways to partition a set of n elements into exactly k non-empty subsets. $S_{n,k} = S_{n-1,k-1} + kS_{n-1,k}$. $S_{n,1} = S_{n,n} = 1$. $x^n = \sum_{k=0}^{n} S_{n,k}\, x^{\underline{k}}$

Bell numbers. B_n is the number of partitions of n elements. $B_0, \ldots = 1, 1, 2, 5, 15, 52, 203, \ldots$
$B_{n+1} = \sum_{k=0}^{n}\binom{n}{k}B_k = \sum_{k=1}^{n} S_{n,k}$. Bell triangle: $B_r = a_{r,1} = a_{r-1,r-1}$, $a_{r,c} = a_{r-1,c-1} + a_{r,c-1}$.

Bernoulli numbers. $\sum_{k=0}^{m-1} k^n = \frac{1}{n+1}\sum_{k=0}^{n}\binom{n+1}{k}B_k m^{n+1-k}$.
$\sum_{j=0}^{m}\binom{m+1}{j}B_j = 0$. $B_0 = 1$, $B_1 = -\frac{1}{2}$. $B_n = 0$, for all odd $n \neq 1$.

Eulerian numbers. $E(n,k)$ is the number of permutations with exactly k descents ($i : \pi_i < \pi_{i+1}$) / ascents ($\pi_i > \pi_{i+1}$) / excedances ($\pi_i > i$) / $k+1$ weak excedances ($\pi_i \geq i$).

Formula: $E(n,k) = (k+1)E(n-1,k) + (n-k)E(n-1,k-1)$. $x^n = \sum_{k=0}^{n-1} E(n,k)\binom{x+k}{n}$.

Burnside's lemma. The number of orbits under group G's action on set X:
$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X_g|$, where $X_g = \{x \in X : g(x) = x\}$. ("Average number of fixed points.")
Let $w(x)$ be weight of x's orbit. Sum of all orbits' weights: $\sum_{o\in X/G} w(o) = \frac{1}{|G|}\sum_{g\in G}\sum_{x\in X_g} w(x)$.

Number Theory

Co-primes Product mod N

$$\prod_{k=1,\gcd(k,m)=1}^{m} k = \begin{cases} -1 & m = 4, p^\alpha, 2\times p^\alpha \\ 1 & otherwise \end{cases} \quad (1)$$

Linear diophantine equation. $ax+by=c$. Let $d=\gcd(a,b)$. A solution exists iff $d|c$. If (x_0,y_0) is any solution, then all solutions are given by $(x,y) = (x_0 + \frac{b}{d}t, y_0 - \frac{a}{d}t)$, $t\in\mathbb{Z}$. To find some solution (x_0,y_0), use extended GCD to solve $ax_0+by_0 = d = \gcd(a,b)$, and multiply its solutions by $\frac{c}{d}$.

Linear diophantine equation in n variables: $a_1x_1 + \cdots + a_nx_n = c$ has solutions iff $\gcd(a_1,\ldots,a_n)|c$. To find some solution, let $b = \gcd(a_2,\ldots,a_n)$, solve $a_1x_1 + by = c$, and iterate with $a_2x_2 + \cdots = y$.

Extended GCD

```
// Finds g = gcd(a,b) and x, y such that ax+by=g. Bounds: |x|<=b+1, |y|<=a+1.
void gcdext(int &g, int &x, int &y, int a, int b)
{ if (b == 0) { g = a; x = 1; y = 0; }
  else        { gcdext(g, y, x, b, a % b); y = y - (a / b) * x; } }
```

Multiplicative inverse of a modulo m: x in $ax+my=1$, or $a^{\phi(m)-1} \pmod m$.

Chinese Remainder Theorem. System $x \equiv a_i \pmod{m_i}$ for $i = 1,\ldots,n$, with pairwise relatively-prime m_i has a unique solution modulo $M = m_1m_2\ldots m_n$: $x = a_1b_1\frac{M}{m_1} + \cdots + a_nb_n\frac{M}{m_n} \pmod M$, where b_i is modular inverse of $\frac{M}{m_i}$ modulo m_i.

System $x \equiv a \pmod m$, $x \equiv b \pmod n$ has solutions iff $a \equiv b \pmod g$, where $g = \gcd(m,n)$. The solution is unique modulo $L = \frac{mn}{g}$, and equals: $x \equiv a + T(b-a)m/g \equiv b + S(a-b)n/g \pmod L$, where S and T are integer solutions of $mT + nS = \gcd(m,n)$.

Prime-counting function. $\pi(n) = |\{p \le n : p \text{ is prime}\}|$. $n/\ln(n) < \pi(n) < 1.3n/\ln(n)$. $\pi(1000) = 168$, $\pi(10^6) = 78498$, $\pi(10^9) = 50\ 847\ 534$. n-th prime $\approx n\ln n$.

Miller-Rabin's primality test. Given $n = 2^rs+1$ with odd s, and a random integer $1 < a < n$.
If $a^s \equiv 1 \pmod n$ or $a^{2^js} \equiv -1 \pmod n$ for some $0 \le j \le r-1$, then n is a probable prime. With bases 2, 7 and 61, the test indentifies all composites below 2^{32}. Probability of failure for a random a is at most 1/4.

Pollard-ρ. Choose random x_1, and let $x_{i+1} = x_i^2 - 1 \pmod n$. Test $\gcd(n, x_{2^k+i} - x_{2^k})$ as possible n's factors for $k = 0,1,\ldots$ Expected time to find a factor: $O(\sqrt{m})$, where m is smallest prime power in n's factorization. That's $O(n^{1/4})$ if you check $n = p^k$ as a special case before factorization.

Fermat primes. A Fermat prime is a prime of form $2^{2^n}+1$. The only known Fermat primes are 3, 5, 17, 257, 65537. A number of form 2^n+1 is prime only if it is a Fermat prime.

Perfect numbers. $n > 1$ is called perfect if it equals sum of its proper divisors and 1. Even n is perfect iff $n = 2^{p-1}(2^p-1)$ and 2^p-1 is prime (Mersenne's). No odd perfect numbers are yet found.

Carmichael numbers. A positive composite n is a Carmichael number ($a^{n-1} \equiv 1 \pmod n$ for all a: $\gcd(a,n) = 1$), iff n is square-free, and for all prime divisors p of n, $p-1$ divides $n-1$.

Number/sum of divisors. $\tau(p_1^{a_1}\ldots p_k^{a_k}) = \prod_{j=1}^k (a_j+1)$. $\sigma(p_1^{a_1}\ldots p_k^{a_k}) = \prod_{j=1}^k \frac{p_j^{a_j+1}-1}{p_j-1}$.

Euler's phi function. $\phi(n) = |\{m \in \mathbb{N}, m \leq n, \gcd(m,n) = 1\}|$.
$\phi(mn) = \frac{\phi(m)\phi(n)\gcd(m,n)}{\phi(\gcd(m,n))}$. $\phi(p^a) = p^{a-1}(p-1)$. $\sum_{d|n}\phi(d) = \sum_{d|n}\phi(\frac{n}{d}) = n$.

Euler's theorem. $a^{\phi(n)} \equiv 1 \pmod{n}$, if $\gcd(a,n) = 1$.
Wilson's theorem. p is prime iff $(p-1)! \equiv -1 \pmod{p}$.

Mobius function. $\mu(1) = 1$. $\mu(n) = 0$, if n is not squarefree. $\mu(n) = (-1)^s$, if n is the product of s distinct primes. Let f, F be functions on positive integers. If for all $n \in N$, $F(n) = \sum_{d|n} f(d)$, then $f(n) = \sum_{d|n}\mu(d)F(\frac{n}{d})$, and vice versa. $\phi(n) = \sum_{d|n}\mu(d)\frac{n}{d}$. $\sum_{d|n}\mu(d) = 1$.
If f is multiplicative, then $\sum_{d|n}\mu(d)f(d) = \prod_{p|n}(1 - f(p))$, $\sum_{d|n}\mu(d)^2 f(d) = \prod_{p|n}(1 + f(p))$.

Legendre symbol. If p is an odd prime, $a \in \mathbb{Z}$, then $\left(\frac{a}{p}\right)$ equals 0, if $p|a$; 1 if a is a quadratic residue modulo p; and -1 otherwise. Euler's criterion: $\left(\frac{a}{p}\right) = a^{\left(\frac{p-1}{2}\right)} \pmod{p}$.

Jacobi symbol. If $n = p_1^{a_1} \cdots p_k^{a_k}$ is odd, then $\left(\frac{a}{n}\right) = \prod_{i=1}^{k}\left(\frac{a}{p_i}\right)^{k_i}$.

Primitive roots. If the order of g modulo m ($\min n > 0$: $g^n \equiv 1 \pmod{m}$) is $\phi(m)$, then g is called a primitive root. If Z_m has a primitive root, then it has $\phi(\phi(m))$ distinct primitive roots. Z_m has a primitive root iff m is one of 2, 4, p^k, $2p^k$, where p is an odd prime. If Z_m has a primitive root g, then for all a coprime to m, there exists unique integer $i = \text{ind}_g(a)$ modulo $\phi(m)$, such that $g^i \equiv a \pmod{m}$. $\text{ind}_g(a)$ has logarithm-like properties: $\text{ind}(1) = 0$, $\text{ind}(ab) = \text{ind}(a) + \text{ind}(b)$.

If p is prime and a is not divisible by p, then congruence $x^n \equiv a \pmod{p}$ has $\gcd(n, p-1)$ solutions if $a^{(p-1)/\gcd(n,p-1)} \equiv 1 \pmod{p}$, and no solutions otherwise. (Proof sketch: let g be a primitive root, and $g^i \equiv a \pmod{p}$, $g^u \equiv x \pmod{p}$. $x^n \equiv a \pmod{p}$ iff $g^{nu} \equiv g^i \pmod{p}$ iff $nu \equiv i \pmod{p}$.)

Discrete logarithm problem. Find x from $a^x \equiv b \pmod{m}$. Can be solved in $O(\sqrt{m})$ time and space with a meet-in-the-middle trick. Let $n = \lceil\sqrt{m}\rceil$, and $x = ny - z$. Equation becomes $a^{ny} \equiv ba^z \pmod{m}$. Precompute all values that the RHS can take for $z = 0, 1, \ldots, n-1$, and brute force y on the LHS, each time checking whether there's a corresponding value for RHS.

Pythagorean triples. Integer solutions of $x^2 + y^2 = z^2$ All relatively prime triples are given by: $x = 2mn, y = m^2 - n^2, z = m^2 + n^2$ where $m > n, \gcd(m,n) = 1$ and $m \not\equiv n \pmod{2}$. All other triples are multiples of these. Equation $x^2 + y^2 = 2z^2$ is equivalent to $(\frac{x+y}{2})^2 + (\frac{x-y}{2})^2 = z^2$.

Postage stamps/McNuggets problem. Let a, b be relatively-prime integers. There are exactly $\frac{1}{2}(a-1)(b-1)$ numbers *not* of form $ax + by$ ($x, y \geq 0$), and the largest is $(a-1)(b-1) - 1 = ab - a - b$.

Fermat's two-squares theorem. Odd prime p can be represented as a sum of two squares iff $p \equiv 1 \pmod{4}$. A product of two sums of two squares is a sum of two squares. Thus, n is a sum of two squares iff every prime of form $p = 4k + 3$ occurs an even number of times in n's factorization.

RSA. Let p and q be random distinct large primes, $n = pq$. Choose a small odd integer e, relatively prime to $\phi(n) = (p-1)(q-1)$, and let $d = e^{-1} \pmod{\phi(n)}$. Pairs (e,n) and (d,n) are the public and secret keys, respectively. Encryption is done by raising a message $M \in Z_n$ to the power e or d, modulo n.

Geometry

Pick's theorem. $I = A - B/2 + 1$, where A is the area of a lattice polygon, I is number of lattice points inside it, and B is number of lattice points on the boundary. Number of lattice points minus one on a line segment from $(0,0)$ and (x,y) is $\gcd(x,y)$.

$a \cdot b = a_x b_x + a_y b_y = |a| \cdot |b| \cdot \cos(\theta)$
$a \times b = a_x b_y - a_y b_x = |a| \cdot |b| \cdot \sin(\theta)$
3D: $a \times b = (a_y b_z - a_z b_y,\ a_z b_x - a_x b_z,\ a_x b_y - a_y b_x)$

Line $ax + by = c$ through $A(x_1, y_1)$ and $B(x_2, y_2)$: $a = y_1 - y_2$, $b = x_2 - x_1$, $c = ax_1 + by_1$.

Half-plane to the left of the directed segment AB: $ax + by \geq c$.
Normal vector: (a, b). Direction vector: $(b, -a)$. Perpendicular line: $-bx + ay = d$.
Point of intersection of $a_1x + b_1y = c_1$ and $a_2x + b_2y = c_2$ is $\frac{1}{a_1b_2 - a_2b_1}(c_1b_2 - c_2b_1, a_1c_2 - a_2c_1)$.
Distance from line $ax + by + c = 0$ to point (x_0, y_0) is $|ax_0 + by_0 + c|/\sqrt{a^2 + b^2}$.
Distance from line AB to P (for any dimension): $\frac{|(A-P)\times(B-P)|}{|A-B|}$.
Point-line segment distance:

```
if (dot(B-A, P-A) < 0) return dist(A,P);
if (dot(A-B, P-B) < 0) return dist(B,P);
return fabs(cross(P,A,B) / dist(A,B));
```

Projection of point C onto line AB is $\frac{AB \cdot AC}{AB \cdot AB}AB$.
Projection of (x_0, y_0) onto line $ax + by = c$ is $(x_0, y_0) + \frac{1}{a^2+b^2}(ad, bd)$, where $d = c - ax_0 - by_0$.
Projection of the origin is $\frac{1}{a^2+b^2}(ac, bc)$.

Segment-segment intersection. Two line segments intersect if one of them contains an endpoint of the other segment, or each segment straddles the line, containing the other segment (AB straddles line l if A and B are on the opposite sides of l.)

Circle-circle and circle-line intersection.

```
a = x2 - x1;   b = y2 - y1;   c = [(r1^2 - x1^2 - y1^2) - (r2^2 - x2^2 - y2^2)] / 2;
d = sqrt(a^2 + b^2);
if not |r1 - r2| <= d <= |r1 + r2|, return "no solution"
if d == 0, circles are concentric, a special case
// Now intersecting circle (x1,y1,r1) with line ax+by=c
Normalize line: a /= d; b /= d; c /= d;     // d=sqrt(a^2+b^2)
e = c - a*x1 - b*y1;
h = sqrt(r1^2 - e^2);                       // check if r1<e for circle-line test
return (x1, y1) + (a*e, b*e) +/- h*(-b, a);
```

Circle from 3 points (circumcircle). Intersect two perpendicular bisectors. Line perpendicular to $ax + by = c$ has the form $-bx + ay = d$. Find d by substituting midpoint's coordinates.

Angular bisector of angle ABC is line BD, where $D = \frac{BA}{|BA|} + \frac{BC}{|BC|}$.
Center of incircle of triangle ABC is at the intersection of angular bisectors, and is $\frac{a}{a+b+c}A + \frac{b}{a+b+c}B + \frac{c}{a+b+c}C$, where a, b, c are lengths of sides, opposite to vertices A, B, C. Radius $= \frac{2S}{a+b+c}$.

Counter-clockwise rotation around the origin. $(x, y) \mapsto (x\cos\phi - y\sin\phi, x\sin\phi + y\cos\phi)$.
90-degrees counter-clockwise rotation: $(x, y) \mapsto (-y, x)$. Clockwise: $(x, y) \mapsto (y, -x)$.

3D rotation by ccw angle ϕ around axis $\mathbf{n}$: $\mathbf{r}' = \mathbf{r}\cos\phi + \mathbf{n}(\mathbf{n} \cdot \mathbf{r})(1 - \cos\phi) + (\mathbf{n} \times \mathbf{r})\sin\phi$

Plane equation from 3 points. $N \cdot (x, y, z) = N \cdot A$, where N is normal: $N = (B - A) \times (C - A)$.

3D figures

Sphere	Volume $V = \frac{4}{3}\pi r^3$, surface area $S = 4\pi r^2$
	$x = \rho\sin\theta\cos\phi$, $y = \rho\sin\theta\sin\phi$, $z = \rho\cos\theta$, $\phi \in [-\pi, \pi]$, $\theta \in [0, \pi]$
Spherical section	Volume $V = \pi h^2(r - h/3)$, surface area $S = 2\pi rh$
Pyramid	Volume $V = \frac{1}{3}hS_{base}$
Cone	Volume $V = \frac{1}{3}\pi r^2 h$, lateral surface area $S = \pi r\sqrt{r^2 + h^2}$

Area of a simple polygon. $\frac{1}{2}\sum_{i=0}^{n-1}(x_iy_{i+1} - x_{i+1}y_i)$, where $x_n = x_0, y_n = y_0$.
Area is negative if the boundary is oriented clockwise.

Winding number. Shoot a ray from given point in an arbitrary direction. For each intersection of ray with polygon's side, add $+1$ if the side crosses it counterclockwise, and -1 if clockwise.

Convex Hull

```
bool operator <(Point a, Point b) { return a.x < b.x || (a.x == b.x && a.y < b.y); }

// Returns convex hull in counter-clockwise order.
```

```
// Note: the last point in the returned list is the same as the first one.
vector<Point> ConvexHull(vector<Point> P) {
    int n = P.size(), k = 0;  vector<Point> H(2*n);
    sort(P.begin(), P.end());
    for (int i = 0; i < n; i++)
      { while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;  H[k++] = P[i]; }
    for (int i = n-2, t = k+1; i >= 0; i--)
      { while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;  H[k++] = P[i]; }
    H.resize(k);
    return H;
}
```

Trigonometric identities

$\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$	$\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$
$\sin(\alpha-\beta) = \sin\alpha\cos\beta - \cos\alpha\sin\beta$	$\cos(\alpha-\beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$
$\tan(\alpha+\beta) = \frac{\tan\alpha+\tan\beta}{1-\tan\alpha\tan\beta}$	$\sin 2\alpha = 2\sin\alpha\cos\alpha$, $\cos 2\alpha = \cos^2\alpha - \sin^2\alpha$
$\cos^2\alpha = \frac{1}{2}(1+\cos 2\alpha)$	$\sin^2\alpha = \frac{1}{2}(1-\cos 2\alpha)$
$\sin\alpha + \sin\beta = 2\sin\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2}$	$\cos\alpha + \cos\beta = 2\cos\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2}$
$\sin\alpha - \sin\beta = 2\sin\frac{\alpha-\beta}{2}\cos\frac{\alpha+\beta}{2}$	$\cos\alpha - \cos\beta = -2\sin\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}$
$\tan\alpha + \tan\beta = \frac{\sin(\alpha+\beta)}{\cos\alpha\cos\beta}$	$\cot\alpha + \cot\beta = \frac{\sin(\alpha+\beta)}{\sin\alpha\sin\beta}$
$\sin\alpha\sin\beta = \frac{1}{2}[\cos(\alpha-\beta) - \cos(\alpha+\beta)]$	$\cos\alpha\cos\beta = \frac{1}{2}[\cos(\alpha-\beta) + \cos(\alpha+\beta)]$
$\sin\alpha\cos\beta = \frac{1}{2}[\sin(\alpha+\beta) + \sin(\alpha-beta)]$	$\sin' x = \cos x$, $\cos' x = -\sin x$
Law of sines: $\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C} = 2R_{out}$.	Inscribed/outscribed circles: $R_{out} = \frac{abc}{4S}$, $R_{in} = \frac{2S}{a+b+c}$
Law of cosines: $c^2 = a^2 + b^2 - 2ab\cos C$.	Heron: $\sqrt{s(s-a)(s-b)(s-c)}$, $s = \frac{a+b+c}{2}$.
Law of tangents: $\frac{a+b}{a-b} = \frac{\tan[\frac{1}{2}(A+B)]}{\tan[\frac{1}{2}(A-B)]}$	Δ's area, given side and adjacent angles: $\frac{c^2}{2(\cot\alpha+\cot\beta)}$

Math

Stirling's approximation $z! = \Gamma(z+1) = \sqrt{2\pi}\ z^{z+1/2}\ e^{-z}(1 + \frac{1}{12z} + \frac{1}{288z^2} - \frac{139}{51840z^3} + \dots)$

Simpson's rule. $\int_a^{b=a+2h} f(x)dx = \frac{b-a}{6}(f(a) + 4f(a+h) + f(b)) + O(h^5 f^{(4)}(\xi))$.
Boole's. $\int_a^{b=a+4h} f(x)dx = \frac{b-a}{90}(7f(a) + 32f(a+h) + 12f(a+2h) + 32f(a+3h) + 7f(b)) + O(h^7 f^{(6)}(\xi))$.

Newton's method. $x_{n+1} = x_n - J^{-1}(x_n)\cdot F(x_n)$, where $J(x)$ is Jacobian matrix $J_{ij} = \frac{\partial f_i}{\partial x_j}$.

Runge-Kutta, 4-th order. Solves initial-value problem $y'(x) = f(x,y)$, $y(x_0) = y_0$.

$k_1 = f(x_n, y_n)$	$k_4 = f(x_n + h, y_n + k_3 h)$
$k_2 = f(x_n + h/2, y_n + k_1 h/2)$	$y_{n+1} = y_n + (k_1 + 2k_2 + 2k_3 + k_4)h/6 + O(h^5)$
$k_3 = f(x_n + h/2, y_n + k_2 h/2)$	$x_{n+1} = x_n + h$

Taylor series. $f(x) = f(a) + \frac{x-a}{1!}f'(a) + \frac{(x-a)^2}{2!}f^{(2)}(a) + \dots + \frac{(x-a)^n}{n!}f^{(n)}(a) + \dots$.
$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$
$\ln x = 2(a + \frac{a^3}{3} + \frac{a^5}{5} + \dots)$, where $a = \frac{x-1}{x+1}$. $\ln x^2 = 2\ln x$.
$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots$, $\arctan x = \arctan c + \arctan\frac{x-c}{1+xc}$ (e.g c=.2)
$\pi = 4\arctan 1$, $\pi = 6\arcsin\frac{1}{2}$

Cauchy's formula. $f(z_0) = \frac{1}{2\pi i}\oint_\gamma \frac{f(z)}{z-z_0}\,dz$

Circle cut area. $\int 2\sqrt{1-x^2}\,dx = x\sqrt{1-x^2} + \arcsin x$

Trigonometric substitution. $t = \tan\frac{x}{2}$, $x = 2\arctan t$, $dx = \frac{2}{1+t^2}\,dt$, $\sin x = \frac{2t}{1+t^2}$, $\cos x = \frac{1-t^2}{1+t^2}$.
Parametric curve length. $\int_a^b \sqrt{[x'(t)]^2 + [y'(t)]^2 + [z'(t)]^2}dt$
Directional derivative. $\frac{\partial f}{\partial \vec{a}} = \frac{\vec{a}}{|\vec{a}|}\cdot\nabla f$
Surface normal. $\vec{n}(u,v) = \frac{\partial \vec{r}}{\partial u} \times \frac{\partial \vec{r}}{\partial v}$

Surface area. $\iint_D |\vec{n}(u,v)|\ du\ dv$. For surfaces $z = z(u,v)$, $|\vec{n}(u,v)|^2 = 1 + (z'_x)^2 + (z'_y)^2$
Green's theorem. $\oint_C P\ dx + Q\ dy = \iint_D (\frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y})\ dx\ dy$
Stokes' theorem. $\int_D \nabla \times \vec{a} \cdot d\vec{S} = \oint_{\partial D} \vec{a} \cdot d\vec{r}$
Normal distribution. $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp -\frac{(x-\mu)^2}{2\sigma^2}$.
Linear DE. $y' + P(x)y = Q(x)$. Solution: $y = e^{-A}(\int Pe^A dx + C)$, where $A = \int Q(x)dx$.
$a_n y^{(n)} + \cdots + a_1 y' + a_0 = 0$. Guess: $y = e^{rx}$. (for multiplicity m roots: $y = x^k e^{rx}$, $k = 0, \ldots, m-1$)
Largange multipliers. Extrema points of $f(x) = f(x_1, \ldots, x_n)$ on domain specified by system $\phi_1(x) = 0, \ldots, \phi_m(x) = 0$ are found by solving system: $\Phi = f(x) + \lambda_1\phi_1(x) + \cdots + \lambda_m\phi_m(x)$, $\frac{\partial\Phi}{\partial x_i} = \frac{\partial\Phi}{\partial\lambda_j} = 0$ (i=1..n, j=1..m)

Bayes' theorem. $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$

Graph Theory

Euler's theorem. For any planar graph, $V - E + F = 1 + C$, where V is the number of graph's vertices, E is the number of edges, F is the number of faces in graph's planar drawing, and C is the number of connected components. Corollary: $V - E + F = 2$ for a 3D polyhedron.

Vertex covers and independent sets. Let M, C, I be a max matching, a min vertex cover, and a max independent set. Then $|M| \le |C| = N - |I|$, with equality for bipartite graphs. Complement of an MVC is always a MIS, and vice versa. Given a bipartite graph with partitions (A, B), build a network: connect source to A, and B to sink with edges of capacities, equal to the corresponding nodes' weights, or 1 in the unweighted case. Set capacities of the original graph's edges to the infinity. Let (S, T) be a minimum s-t cut. Then a maximum(-weighted) independent set is $I = (A \cap S) \cup (B \cap T)$, and a minimum(-weighted) vertex cover is $C = (A \cap T) \cup (B \cap S)$.

Matrix-tree theorem. Let matrix $T = [t_{ij}]$, where t_{ij} is the number of multiedges between i and j, for $i \ne j$, and $t_{ii} = -\deg_i$. Number of spanning trees of a graph is equal to the determinant of a matrix obtained by deleting any k-th row and k-th column from T.

Euler tours. Euler tour in an undirected graph exists iff the graph is connected and each vertex has an even degree. Euler tour in a directed graph exists iff in-degree of each vertex equals its out-degree, and underlying undirected graph is connected. Construction:

```
doit(u):
  for each edge e = (u, v) in E, do: erase e, doit(v)
  prepend u to the list of vertices in the tour
```

Stable marriages problem. While there is a free man m: let w be the most-preferred woman to whom he has not yet proposed, and propose m to w. If w is free, or is engaged to someone whom she prefers less than m, match m with w, else deny proposal.

Stoer-Wagner's min-cut algorithm. Start from a set A containing an arbitrary vertex. While $A \ne V$, add to A the most tightly connected vertex ($z \notin A$ such that $\sum_{x \in A} w(x, z)$ is maximized.) Store cut-of-the-phase (the cut between the last added vertex and rest of the graph), and merge the two vertices added last. Repeat until the graph is contracted to a single vertex. Minimum cut is one of the cuts-of-the-phase.

Tarjan's offline LCA algorithm. (Based on DFS and union-find structure.)

```
DFS(x):
  ancestor[Find(x)] = x
  for all children y of x:
     DFS(y); Union(x, y); ancestor[Find(x)] = x
  seen[x] = true
  for all queries {x, y}:
    if seen[y] then output "LCA(x, y) is ancestor[Find(y)]"
```

Strongly-connected components. Kosaraju's algorithm.
1. Let G^T be a transpose G (graph with reversed edges.)

1. Call DFS(G^T) to compute finishing times $f[u]$ for each vertex u.
3. For each vertex u, in the order of decreasing $f[u]$, perform DFS(G, u).
4. Each tree in the 3rd step's DFS forest is a separate SCC.

2-SAT. Build an implication graph with 2 vertices for each variable – for the variable and its inverse; for each clause $x \vee y$ add edges $(\overline{x}, y)$ and $(\overline{y}, x)$. The formula is satisfiable iff x and $\overline{x}$ are in distinct SCCs, for all x. To find a satisfiable assignment, consider the graph's SCCs in topological order from sinks to sources (i.e. Kosaraju's last step), assigning 'true' to all variables of the current SCC (if it hasn't been previously assigned 'false'), and 'false' to all inverses.

Randomized algorithm for non-bipartite matching. Let G be a simple undirected graph with even $|V(G)|$. Build a matrix A, which for each edge $(u, v) \in E(G)$ has $A_{i,j} = x_{i,j}$, $A_{j,i} = -x_{i,j}$, and is zero elsewhere. Tutte's theorem: G has a perfect matching iff $\det G$ (a multivariate polynomial) is identically zero. Testing the latter can be done by computing the determinant for a few random values of $x_{i,j}$'s over some field. (e.g. Z_p for a sufficiently large prime p)

Prufer code of a tree. Label vertices with integers 1 to n. Repeatedly remove the leaf with the smallest label, and output its only neighbor's label, until only one edge remains. The sequence has length $n-2$. Two isomorphic trees have the same sequence, and every sequence of integers from 1 and n corresponds to a tree. Corollary: the number of labelled trees with n vertices is n^{n-2}.

Erdos-Gallai theorem. A sequence of integers $\{d_1, d_2, \ldots, d_n\}$, with $n-1 \geq d_1 \geq d_2 \geq \cdots \geq d_n \geq 0$ is a degree sequence of some undirected simple graph iff $\sum d_i$ is even and $d_1 + \cdots + d_k \leq k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i)$ for all $k = 1, 2, \ldots, n-1$.

String Algorithms

```
char *kmp(char *text, char *pat) {
  int i,k,n=strlen(pat), *phi=...
  phi[0] = phi[1] = k = 0;
  for (i = 1; i < n; i++) {
    while (k > 0 && pat[k] != pat[i])
      k = phi[k];
    phi[i+1] = pat[k]==pat[i] ? ++k : 0;
  }
  for (i=k=0; text[i] && k < n; i++) {
    while (k > 0 && pat[k] != text[i])
      k = phi[k];
    if (pat[k] == text[i]) ++k;
  }
  return k == n ? text+i-n : NULL;
}
```

```
vector<int> zfunction(char *s) {
  int N = strlen(s), a=0, b=0;
  vector<int> z(N, N);
  for (int i = 1; i < N; i++) {
    int k = i<b ? min(b-i, z[i-a]) : 0;
    while (i+k < N && s[i+k]==s[k]) ++k;
    z[i] = k;
    if (i+k > b) { a=i; b=i+k; }
  }
  return z;
}

Definition:
z[i] = max {k: s[i..i+k-1]=s[0..k-1]}
```

Suffix array. $O(n \log^2 n)$ time, 16 bytes/char overhead.

```
// Input: text, N
// Output: sa[] is a sorted list of offsets to all non-empty suffixes,
// lcp[i] = length of longest common prefix of text+sa[i] and text+sa[i+1]
char text[MAX]; long long key[MAX]; int N, sa[MAX], rank[MAX], *lcp=(int*)key;
struct Cmp { bool operator()(int i, int j) const { return key[i]<key[j]; } };

void build() {
    for (int i = 0; i < N; i++) { sa[i] = i; key[i] = text[i]; }
    sort(sa, sa+N, Cmp());
    for (int K = 1; ; K *= 2) {
        for (int i = 0; i < N; i++)
            rank[sa[i]] = i>0 && key[sa[i-1]]==key[sa[i]] ? rank[sa[i-1]] : i;
        if (K >= N) break;
        for (int i = 0; i < N; i++)
            key[i] = rank[i] * (N+1LL) + (i+K < N ? rank[i+K]+1 : 0);
```

```
        sort(sa, sa+N, Cmp());
    }
    for (int i = 0, k = 0; i < N; i++) {
        if (k > 0) k--;
        if (rank[i] == N-1) { lcp[N-1] = -1; k = 0; continue; }
        int j = sa[rank[i]+1];
        while (text[i+k] == text[j+k]) k++;
        lcp[rank[i]] = k;
    }
}
```

Burrows-Wheeler inverse transform. Let $B[1..n]$ be the input (last column of sorted matrix of string's rotations.) Get the first column, $A[1..n]$, by sorting B. For each k-th occurence of a character c at index i in A, let $next[i]$ be the index of corresponding k-th occurence of c in B. The r-th fow of the matrix is $A[r]$, $A[next[r]]$, $A[next[next[r]]]$, ...

Huffman's algorithm. Start with a forest, consisting of isolated vertices. Repeatedly merge two trees with the lowest weights.

Miscellanious

Bit tricks
Clearing the lowest 1 bit: `x & (x - 1)`, all trailing 1's: `x & (x + 1)`
Setting the lowest 0 bit: `x | (x + 1)`
Enumerating subsets of a bitmask m: `x=0; do { ...; x=(x+1+~m)&m; } while (x!=0);`
`__builtin_ctz`/`__builtin_clz` returns the number of trailing/leading zero bits.
`__builtin_popcount(unsigned x)` counts 1-bits (slower than table lookups).
For 64-bit unsigned integer type, use the suffix '`ll`', i.e. `__builtin_popcountll`.

Warnsdorff's heuristic for knight's tour. At each step choose a square which has the least number of valid moves that the knight can make from there.

Optimal BST. $root[i, j-1] \le root[i, j] \le root[i+1, j]$.

Flow-shop scheduling (Johnson's problem). Schedule N jobs on 2 machines to minimize completition time. i-th job takes a_i and b_i time to execute on 1st and 2nd machine, respectively. Each job must be first executed on the first machine, then on second. Both machines execute all jobs in the same order. Solution: sort jobs by key $a_i < b_i \ ? \ a_i : (\infty - b_i)$, i.e. first execute all jobs with $a_i < b_i$ in order of increasing a_i, then all other jobs in order of decreasing b_i.

Days of week

January 1, 1600: Saturday	January 1, 1900: Monday	June 13, 2042: Friday
January 1, 2008: Tuesday	April 1, 2008: Tuesday	April 9, 2008: Wednesday
December 31, 1999: Friday	January 1, 3000: Wednesday	

Data Structures

Fenwick Tree

```
int a[MAXN];

// value[n] += x
void add(int n, int x) { for (; n < MAXN; n |= n + 1) a[n] += x; }

// Returns value[0] + value[1] + ... + value[n]
int sum(int n) { int s=0; while (n>=0) { s+=a[n]; n=(n&(n+1))-1; } return s; }
```

AVL Tree

```
struct Node {
    Node *l, *r;  int h, size, key;
    Node(int k) : l(0), r(0), h(1), size(1), key(k) {}
    void u() { h=1+max(l?l->h:0, r?r->h:0); size=(l?l->size:0)+1+(r?r->size:0); }
};

Node *rotl(Node *x) { Node *y=x->r; x->r=y->l; y->l=x; x->u(); y->u(); return y; }
Node *rotr(Node *x) { Node *y=x->l; x->l=y->r; y->r=x; x->u(); y->u(); return y; }

Node *rebalance(Node *x) {
    x->u();
    if (x->l->h > 1 + x->r->h) {
        if (x->l->l->h < x->l->r->h) x->l = rotl(x->l);
        x = rotr(x);
    } else if (x->r->h > 1 + x->l->h) {
        if (x->r->r->h < x->r->l->h) x->r = rotr(x->r);
        x = rotl(x);
    }
    return x;
}

Node *insert(Node *x, int key) {
    if (x == NULL) return new Node(key);
    if (key < x->key) x->l = insert(x->l, key); else x->r = insert(x->r, key);
    return rebalance(x);
}
```

Treap

```
struct Node {
    int key, aux, size;  Node *l, *r;    // BST w.r.t. key;  min-heap w.r.t. aux
    Node(int k) : key(k), aux(rand()), size(1), l(0), r(0) {}
};

Node *upd(Node *p) { if(p) p->size=1+(p->l?p->l->size:0)+(p->r?p->r->size:0); return p; }

void split(Node *p, Node *by, Node **L, Node **R) {
    if (p == NULL) { *L = *R = NULL; }
    else if (p->key < by->key) { split(p->r, by, &p->r, R); *L = upd(p); }
    else { split(p->l, by, L, &p->l); *R = upd(p); }
}

Node *merge(Node *L, Node *R) {
    Node *p;
    if (L == NULL || R == NULL) p = (L != NULL ? L : R);
    else if (L->aux < R->aux) { L->r = merge(L->r, R); p = L; }
    else { R->l = merge(L, R->l); p = R; }
    return upd(p);
}

Node *insert(Node *p, Node *n) {
    if (p == NULL) return upd(n);
    if (n->aux <= p->aux) { split(p, n, &n->l, &n->r); return upd(n); }
    if (n->key < p->key) p->l = insert(p->l, n); else p->r = insert(p->r, n);
    return upd(p);
}

Node *erase(Node *p, int key) {
    if (p == NULL) return NULL;
```

```
    if (key == p->key) { Node *q = merge(p->l, p->r); delete p; return upd(q); }
    if (key < p->key) p->l = erase(p->l, key); else p->r = erase(p->r, key);
    return upd(p);
}
```