VinUniversity ICPC Team Notebook (2025-26)

Contents

1 Mathematics

1.1 Combinatorics

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$
$$\binom{n}{k} = \binom{n}{n-k}$$
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$
$$\sum_{k=0}^{r} \binom{m}{k}\binom{n}{r-k} = \binom{m+n}{r}$$
$$\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1}$$
$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$
$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$
$$\sum_{k=0}^{n} \binom{k}{r} = \binom{n+1}{r+1}$$
$$(1+x)^\alpha = \sum_{k=0}^{\infty} \binom{\alpha}{k} x^k$$
$$k\binom{n}{k} = n\binom{n-1}{k-1}$$

1.2 Prime numbers

```cpp
// O(sqrt(x)) Exhaustive Primality Test
#include <bits/stdc++.h>
#define EPS 1e-7
typedef long long LL;
bool IsPrimeSlow (LL x)
{
  if(x<=1) return false;
  if(x<=3) return true;
  if (!(x%2) || !(x%3)) return false;
  LL s=(LL)(sqrt((long double)(x))+EPS);
  for(LL i=5;i<=s;i+=6)
  {
    if (!(x%i) || !(x%(i+2))) return false;
  }
  return true;
}
// Primes less than 1000:
//     2    3    5    7   11   13   17   19   23   29   31   37
//    41   43   47   53   59   61   67   71   73   79   83   89
//    97  101  103  107  109  113  127  131  137  139  149  151
//   157  163  167  173  179  181  191  193  197  199  211  223
//   227  229  233  239  241  251  257  263  269  271  277  281
//   283  293  307  311  313  317  331  337  347  349  353  359
//   367  373  379  383  389  397  401  409  419  421  431  433
//   439  443  449  457  461  463  467  479  487  491  499  503
//   509  521  523  541  547  557  563  569  571  577  587  593
//   599  601  607  613  617  619  631  641  643  647  653  659
//   661  673  677  683  691  701  709  719  727  733  739  743
//   751  757  761  769  773  787  797  809  811  821  823  827
//   829  839  853  857  859  863  877  881  883  887  907  911
//   919  929  937  941  947  953  967  971  977  983  991  997
// 4  The largest prime < 10^4 is 9973.
// 5  The largest prime < 10^5 is 99991.
// 6  The largest prime < 10^6 is 999983.
// 7  The largest prime < 10^7 is 9999991.
// 8  The largest prime < 10^8 is 99999989.
// 9  The largest prime < 10^9 is 999999937.
//10  The largest prime < 10^10 is 9999999967.
//11  The largest prime < 10^11 is 99999999977.
//12  The largest prime < 10^12 is 999999999989.
//13  The largest prime < 10^13 is 9999999999971.
//14  The largest prime < 10^14 is 99999999999973.
//15  The largest prime < 10^15 is 999999999999989.
//16  The largest prime < 10^16 is 9999999999999937.
//17  The largest prime < 10^17 is 99999999999999997.
//18  The largest prime < 10^18 is 999999999999999989.
// Primes past 1e9 are. 1000000007 1000000009 1000000021
// 1000000033 1000000087 1000000093 1000000097 1000000103
// Primes below 1e9: 999999937 999999929 999999893 999999883
// 999999797 999999761 999999757 999999751 999999739 999999733
// Primes <= (10^9)/31: 32257999 32257963 32257921 32257909
// Primes<=(10^9)/307: 3257299 3257297 3257291 3257269 3257257
//
//0000000111111111122222222223333333333444444444455555555556666
//3456789012345678901234567890123456789012345678901234567890123
```

1.3 Highly Composite Numbers

```python
# Prints all HCN (highly composite numbers) <= MAXN (=10**18)
# The value of MAXN can be changed arbitrarily. When MAXN =
# 10**100, the program takes <1s to generate the list of HCN.
from math import log
MAXN = 10**18

# TODO: Generates a list of the first primes (with product > MAXN).
primes = gen_primes() # primes = [2, 3, 5, 7, 11, ...]

# Generates a list of the hcn <= MAXN.
def gen_hcn():
    # List of (number, number of divisors, exponents of the
        factorization)
  hcn = [(1, 1, [])]
  for i in range(len(primes)):
    new_hcn = []
    for el in hcn:
      new_hcn.append(el)
      if len(el[2]) < i: continue
      e_max = el[2][i-1] if i >= 1 else int(log(MAXN, 2))
      n = el[0]
      for e in range(1, e_max+1):
        n *= primes[i]
        if n > MAXN: break
        div = el[1] * (e+1)
        exponents = el[2] + [e]
        new_hcn.append((n, div, exponents))
    new_hcn.sort()
    hcn = [(1, 1, [])]
    for el in new_hcn:
      if el[1] > hcn[-1][1]: hcn.append(el)
  return hcn

# Biggest HCN smaller than 10^9, 10^12, 10^18, and their number of
    divisors:
# 735134400          1344   2^6*3^3*5^2*7*11*13*17
# 963761198400       6720   2^6*3^4*5^2*7*11*13*17*19*23
# 897612484786617600 103680 2^8*3^4*5^2*7^2*11*13*17*19*23*29*31*37
```

1.4 Number theory (modular, linear Diophantine)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

typedef vector<int> VI;
typedef pair<int, int> PII;

// return a % b (positive value)
int mod(int a, int b) {
  return ((a%b) + b) % b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
  while (b) { int t = a%b; a = b; b = t; }
  return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
  return a / gcd(a, b)*b;
}

// (a^b) mod m via successive squaring
int powermod(int a, int b, int m)
{
  int ret = 1;
  while (b)
  {
    if (b & 1) ret = mod(ret*a, m);
    a = mod(a*a, m);
    b >>= 1;
  }
  return ret;
}
```

```
// returns g = gcd(a, b); finds x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a / b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x - q*xx; x = t;
    t = yy; yy = y - q*yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI ret;
  int g = extended_euclid(a, n, x, y);
  if (!(b%g)) {
    x = mod(x*(b / g), n);
    for (int i = 0; i < g; i++)
      ret.push_back(mod(x + i*(n / g), n));
  }
  return ret;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int g = extended_euclid(a, n, x, y);
  if (g > 1) return -1;
  return mod(x, n);
}

// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
  if (!a && !b)
  {
    if (c) return false;
    x = 0; y = 0;
    return true;
  }
  if (!a)
  {
    if (c % b) return false;
    x = 0; y = c / b;
    return true;
  }
  if (!b)
  {
    if (c % a) return false;
    x = c / a; y = 0;
    return true;
  }
  int g = gcd(a, b);
  if (c % g) return false;
  x = c / g * mod_inverse(a / g, b / g);
  y = (c - a*x) / b;
  return true;
}

int main() {
  // expected: 2
  cout << gcd(14, 30) << endl;

  // expected: 2 -2 1
  int x, y;
  int g = extended_euclid(14, 30, x, y);
  cout << g << " " << x << " " << y << endl;

  // expected: 95 451
  VI sols = modular_linear_equation_solver(14, 30, 100);
  for (int i = 0; i < sols.size(); i++) cout << sols[i] << " ";
  cout << endl;

  // expected: 8
  cout << mod_inverse(8, 9) << endl;

  // expected: 5 -15
  if (!linear_diophantine(7, 2, 5, x, y)) cout << "ERROR" << endl;
  cout << x << " " << y << endl;
  return 0;
}
```

1.5 Chinese Remainder Theorem

```
// Official version: https://cp-algorithms.com/math/chinese-
     remainder-theorem.html

#define ll long long
struct Congruence {
    ll a, m;
};

// CRT = Chinese Remainder Theorem
ll CRT(vector<Congruence> const& congruences) {
    long long M = 1;
    for (auto const& congruence : congruences) {
        M *= congruence.m;
    }

    long long solution = 0;
    for (auto const& congruence : congruences) {
        long long a_i = congruence.a;
        long long M_i = M / congruence.m;
        long long N_i = mod_inv(M_i, congruence.m);
        solution = (solution + a_i * M_i % M * N_i) % M;
    }
    return solution;
}
// CRT({{2,3},{3,5},{2,7}}) = 23 mod 105
// CRT({{3,4},{5,6}}) = 11 mod 12
```

1.6 Discrete Log

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

using namespace std;

// returns any x such that a^x = b (mod m)
// O(m^0.5) complexity
int discrete_log(int a, int b, int m) {
    assert(gcd(a, m) == 1);

    int n = (int)sqrt(m) + 1;

    int an = 1;
    for (int i = 0; i < n; ++i)
        an = ((long long)an * a) % m;

    unordered_map<int, int> vals;
    for (int i = 1, cur = an; i <= n; ++i) {
        if (!vals.count(cur))
            vals[cur] = i;
        cur = ((long long)cur * an) % m;
    }

    for (int i = 0, cur = b; i <= n; ++i) {
        if (vals.count(cur)) {
            int res = (long long)vals[cur] * n - i;
            if (res < m)
                return res;
        }
        cur = ((long long)cur * a) % m;
    }
    return -1;
}

// usage example
int main() {
    // 2^x = 3 (mod 5), x = 3
    cout << discrete_log(2, 3, 5) << endl;
}
```

2 Geometry

2.1 Convex hull

```
// Compute the 2D convex hull of a set of points using the
// monotone chain algorithm. Eliminate redundant points from the
// hull if REMOVE_REDUNDANT is #defined.
//
// Running time: O(n log n)
//
//   INPUT:   a vector of input points, unordered.
//   OUTPUT:  a vector of points in the convex hull,
// counterclockwise, starting with bottommost/leftmost point
#include <bits/stdc++.h>
using namespace std;
#define ll long long
// Convex hull construction in O(n*log(n)): https://cp-algorithms.
     com/geometry/grahams-scan-convex-hull.html

struct point {
    int x, y;
};

bool isNotRightTurn(const point&a, const point&b, const point&c) {
    ll cross = (ll)(a.x-b.x)*(c.y-b.y) - (ll)(a.y-b.y)*(c.x-b.x);
    ll dot = (ll)(a.x-b.x)*(c.x-b.x) + (ll)(a.y-b.y)*(c.y-b.y);
    return cross < 0 || (cross == 0 && dot <= 0);
}

vector<point> convex_hull(vector<point> points) {
    sort(points.begin(), points.end(), [](auto a, auto b) {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    });
    int n = points.size();
    vector<point> hull;
    for (int i = 0; i < 2 * n - 1; i++) {
        int j = i < n ? i : 2 * n - 2 - i;
        while (hull.size() >= 2 &&
        isNotRightTurn(hull.end()[-2], hull.end()[-1], points[j])){
            hull.pop_back();
        }
        hull.push_back(points[j]);
    }
    hull.pop_back();
    return hull;
}

// usage example
int main() {
    vector<point>hull1=convex_hull({{0,0},{3,0},{0,3},{1,1}});
    cout << (3 == hull1.size()) << endl;

    vector<point>hull2=convex_hull({{0,0},{0,0}});
    cout << (1 == hull2.size()) << endl;
}
```

2.2 Miscellaneous geometry

```
// C++ routines for computational geometry.

#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

using namespace std;

double INF = 1e100;
double EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)     const { return PT(x*c,   y*c  ); }
  PT operator / (double c)     const { return PT(x/c,   y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  return os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
```

```cpp
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or
     collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) >
         0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+
       RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by
// William Randolph Franklin); returns 1 for strictly interior
// points, 0 for strictly exterior points, and 0 or 1 for the
// remaining points. Note that it is possible to convert this into
// an *exact* test using integer arithmetic by taking care of the
// division appropriately (making sure to deal with signs
// properly) and then by writing exact tests for checking point on
// polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y -
           p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) <
         EPS) return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R)
     {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (possibly
// nonconvex) polygon, assuming that the coordinates are listed in
// a clockwise or counterclockwise fashion.  Note that the
// centroid is often known as the "center of gravity".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is
     simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

int main() {

  // expected: (-5,2)
  cerr << RotateCCW90(PT(2,5)) << endl;

  // expected: (5,-2)
  cerr << RotateCW90(PT(2,5)) << endl;

  // expected: (-5,2)
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;

  // expected: (5,2)
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

  // expected: (5,2) (7.5,3) (2.5,1)
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) <<
           endl;

  // expected: 6.78903
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

  // expected: 1 0 1
  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) <<
           endl;

  // expected: 0 0 1
  cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) <<
           endl;

  // expected: 1 1 1 0
  cerr << SegmentsIntersect(PT(0,0),PT(2,4),PT(3,1),PT(-1,3))<<" "
       << SegmentsIntersect(PT(0,0),PT(2,4),PT(4,3),PT(0,5)) <<" "
       << SegmentsIntersect(PT(0,0),PT(2,4),PT(2,-1),PT(-2,1))<<" "
       << SegmentsIntersect(PT(0,0),PT(2,4),PT(5,5),PT(1,7))<<endl;

  // expected: (1,2)
  cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT
       (-1,3)) << endl;

  // expected: (1,1)
  cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

  vector<PT> v;
  v.push_back(PT(0,0));
  v.push_back(PT(5,0));
  v.push_back(PT(5,5));
  v.push_back(PT(0,5));

  // expected: 1 1 1 0 0
  cerr << PointInPolygon(v, PT(2,2)) << " "
       << PointInPolygon(v, PT(2,0)) << " "
       << PointInPolygon(v, PT(0,2)) << " "
       << PointInPolygon(v, PT(5,2)) << " "
       << PointInPolygon(v, PT(2,5)) << endl;

  // expected: 0 1 1 1 1
  cerr << PointOnPolygon(v, PT(2,2)) << " "
       << PointOnPolygon(v, PT(2,0)) << " "
       << PointOnPolygon(v, PT(0,2)) << " "
       << PointOnPolygon(v, PT(5,2)) << " "
       << PointOnPolygon(v, PT(2,5)) << endl;

  // expected: (1,6)
  //           (5,4) (4,5)
  //           blank line
  //           (4,5) (5,4)
  //           blank line
  //           (4,5) (5,4)
  vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1),
       5);
  for (int i=0;i<u.size();i++) cerr << u[i] << " "; cerr << endl;
  u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
  for (int i=0;i<u.size();i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
  for (int i=0;i<u.size();i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
  for (int i=0;i<u.size();i++) cerr << u[i] << " "; cerr << endl;
  u=CircleCircleIntersection(PT(1,1),PT(4.5,4.5),10,sqrt(2.0)/2.0);
  for (int i=0;i<u.size();i++) cerr << u[i] << " "; cerr << endl;
  u=CircleCircleIntersection(PT(1,1),PT(4.5,4.5),5,sqrt(2.0)/2.0);
  for (int i=0;i<u.size();i++) cerr << u[i] << " "; cerr << endl;
```

```
  // area should be 5.0
  // centroid should be (1.1666666, 1.166666)
  PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
  vector<PT> p(pa, pa+4);
  PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;

  return 0;
}
```

2.3 Slow Delaunay triangulation

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
//
// OUTPUT:   triples = a vector containing m triples of indices
//                     corresponding to triangle vertices

#include<vector>
#include<cstdio>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
  int n = x.size();
  vector<T> z(n);
  vector<triple> ret;

  for (int i = 0; i < n; i++)
      z[i] = x[i] * x[i] + y[i] * y[i];

  for (int i = 0; i < n-2; i++) {
  for (int j = i+1; j < n; j++) {
  for (int k = i+1; k < n; k++) {
    if (j == k) continue;
    double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
    double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
    double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
    bool flag = zn < 0;
    for (int m = 0; flag && m < n; m++)
    flag = flag && ((x[m]-x[i])*xn +
        (y[m]-y[i])*yn +
        (z[m]-z[i])*zn <= 0);
    if (flag) ret.push_back(triple(i, j, k));
  }
  }
  }
  return ret;
}

int main()
{
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //          0 3 2

    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}
```

2.4 Point in Polygon

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

using namespace std;

using ll = long long;

int pointInPolygon(int qx, int qy, const vector<int> &x, const
     vector<int> &y) {
    int n = x.size();
    int cnt = 0;
    for (int i = 0, j = n - 1; i < n; j = i++) {
        if (y[i] == qy && (x[i] == qx || (y[j] == qy && (x[i] <= qx
            || x[j] <= qx) && (x[i] >= qx || x[j] >= qx))))
            return 0;  // boundary
        if ((y[i] > qy) != (y[j] > qy)) {
            ll det = ((ll)x[i] - qx) * ((ll)y[j] - qy) - ((ll)x[j]
                - qx) * ((ll)y[i] - qy);
            if (det == 0)
                return 0;  // boundary
            if ((det > 0) != (y[j] > y[i]))
                ++cnt;
        }
    }
    return cnt % 2 == 0 ? -1 /* exterior */ : 1 /* interior */;
}

// usage example
int main() {
    vector<int> x{0, 0, 2, 2};
    vector<int> y{0, 2, 2, 0};
    cout << (1 == pointInPolygon(1, 1, x, y)) << endl;
    cout << (0 == pointInPolygon(0, 0, x, y)) << endl;
    cout << (-1 == pointInPolygon(0, 3, x, y)) << endl;
}
```

3 Numerical algorithms

3.1 Systems of linear equations, matrix inverse, determinant

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
  if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k
      ; }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." <<
        endl; exit(0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}

int main() {
  const int n = 4;
  const int m = 2;
  double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
  double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
  VVT a(n), b(n);
  for (int i = 0; i < n; i++) {
    a[i] = VT(A[i], A[i] + n);
    b[i] = VT(B[i], B[i] + m);
  }

  double det = GaussJordan(a, b);

  // expected: 60
  cout << "Determinant: " << det << endl;

  // expected: -0.233333 0.166667 0.133333 0.0666667
  //           0.166667 0.166667 0.333333 -0.333333
  //           0.233333 0.833333 -0.133333 -0.0666667
  //           0.05 -0.75 -0.1 0.2
  cout << "Inverse: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

  // expected: 1.63333 1.3
  //           -0.166667 0.5
  //           2.36667 1.7
  //           -1.85 -1.35
  cout << "Solution: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
      cout << b[i][j] << ' ';
    cout << endl;
  }
}
```

3.2 Reduced row echelon form, matrix rank

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting.  This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxm matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

#include <iostream>
#include <vector>
```

```
#include <cmath>

using namespace std;

const double EPSILON = 1e-10;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

int rref(VVT &a) {
  int n = a.size();
  int m = a[0].size();
  int r = 0;
  for (int c = 0; c < m && r < n; c++) {
    int j = r;
    for (int i = r + 1; i < n; i++)
      if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
    if (fabs(a[j][c]) < EPSILON) continue;
    swap(a[j], a[r]);

    T s = 1.0 / a[r][c];
    for (int j = 0; j < m; j++) a[r][j] *= s;
    for (int i = 0; i < n; i++) if (i != r) {
      T t = a[i][c];
      for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
    }
    r++;
  }
  return r;
}

int main() {
  const int n = 5, m = 4;
  double A[n][m] = {
    {16,  2,  3, 13},
    { 5, 11, 10,  8},
    { 9,  7,  6, 12},
    { 4, 14, 15,  1},
    {13, 21, 21, 13}};
  VVT a(n);
  for (int i = 0; i < n; i++)
    a[i] = VT(A[i], A[i] + m);

  int rank = rref(a);

  // expected: 3
  cout << "Rank: " << rank << endl;

  // expected: 1 0 0 1
  //           0 1 0 3
  //           0 0 1 -3
  //           0 0 0 3.10862e-15
  //           0 0 0 2.22045e-15
  cout << "rref: " << endl;
  for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 4; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }
}
```

3.3 Simplex algorithm

```
// Two-phase simplex algorithm for solving linear opti. of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

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

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j]
        = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][
        n + 1] = b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m + 1][n] = 1;
  }

  void Pivot(int r, int s) {
    double inv = 1.0 / D[r][s];
    for (int i = 0; i < m + 2; i++) if (i != r)
      for (int j = 0; j < n + 2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] * inv;
    for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
    for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N
            [j] < N[s]) s = j;
      }
      if (D[x][s] > -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS) continue;
        if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s
            ] ||
          (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i
            ] < B[r]) r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r =
        i;
    if (D[r][n + 1] < -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -
          numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] &&
              N[j] < N[s]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n +
        1];
    return D[m][n + 1];
  }
};

int main() {

  const int m = 4;
  const int n = 3;
  DOUBLE _A[m][n] = {
    { 6, -1, 0 },
    { -1, -5, 0 },
    { 1, 5, 1 },
    { -1, -5, -1 }
  };
  DOUBLE _b[m] = { 10, -4, 5, -5 };
  DOUBLE _c[n] = { 1, -1, 0 };

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

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

  cerr << "VALUE: " << value << endl; // VALUE: 1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

4 Graph algorithms

4.1 Fast Dijkstra's algorithm

```
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)

#include <queue>
#include <cstdio>

using namespace std;
const int INF = 2000000000;
typedef pair<int, int> PII;

int main() {

  int N, s, t;
  scanf("%d%d%d", &N, &s, &t);
  vector<vector<PII> > edges(N);
  for (int i = 0; i < N; i++) {
    int M;
    scanf("%d", &M);
    for (int j = 0; j < M; j++) {
      int vertex, dist;
      scanf("%d%d", &vertex, &dist);
      edges[i].push_back(make_pair(dist, vertex)); // note order of
          arguments here
    }
  }

  // priority queue but top element has the "smallest" priority
  priority_queue<PII, vector<PII>, greater<PII> > Q;
  vector<int> dist(N, INF), dad(N, -1);
  Q.push(make_pair(0, s));
  dist[s] = 0;
  while (!Q.empty()) {
    PII p = Q.top();
    Q.pop();
    int here = p.second;
    if (here == t) break;
    if (dist[here] != p.first) continue;

    for (vector<PII>::iterator it = edges[here].begin(); it !=
        edges[here].end(); it++) {
      if (dist[here] + it->first < dist[it->second]) {
        dist[it->second] = dist[here] + it->first;
        dad[it->second] = here;
        Q.push(make_pair(dist[it->second], it->second));
      }
    }
  }

  printf("%d\n", dist[t]);
  if (dist[t] < INF)
    for (int i = t; i != -1; i = dad[i])
      printf("%d%c", i, (i == s ? '\n' : ' '));
  return 0;
```

```
}

/*
Sample input:
5 0 4
2 1 2 3 1
2 2 4 4 5
3 1 4 3 3 4 1
2 0 1 2 3
2 1 5 2 1

Expected:
5
4 2 3 0
*/
```

4.2 Bellman Ford's shortest path for negative cycle detection

```
// This function runs the Bellman-Ford algorithm for single source
// shortest paths with negative edge weights.  The function returns
// false if a negative weight cycle is detected.  Otherwise, the
// function returns true and dist[i] is the length of the shortest
// path from start to i.
//
// Running time: O(|V|^3)
//
//   INPUT:   start, w[i][j] = cost of edge from i to j
//   OUTPUT:  dist[i] = min weight path from start to i
//            prev[i] = previous node on the best path from the
//                      start node

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

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

bool BellmanFord (const VVT &w, VT &dist, VI &prev, int start){
  int n = w.size();
  prev = VI(n, -1);
  dist = VT(n, 1000000000);
  dist[start] = 0;

  for (int k = 0; k < n; k++){
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        if (dist[j] > dist[i] + w[i][j]){
          if (k == n-1) return false;
          dist[j] = dist[i] + w[i][j];
          prev[j] = i;
        }
      }
    }
  }

  return true;
}
```

4.3 Strongly connected components

```
#include <vector>
#include <algorithm>
using namespace std;
vector<bool> visited; // keeps track of visited vertices

// runs depth first search starting at vertex v.
// each visited vertex is appended to the output vector when dfs
    leaves it.
void dfs(int v, vector<vector<int> > const& adj, vector<int> &
     output) {
    visited[v] = true;
    for (auto u : adj[v])
        if (!visited[u])
            dfs(u, adj, output);
    output.push_back(v); // used to record the t_out of each vertex
}

// input: adj -- adjacency list of G
// output: components -- the strongy connected components in G
// output: adj_cond -- adjacency list of G^SCC (by root vertices)
void strongly_connected_components(vector<vector<int> > const& adj,
                                  vector<vector<int> > &components,
                                  vector<vector<int> > &adj_cond) {
    int n = adj.size();
    components.clear(), adj_cond.clear();

    vector<int> order; // sorted list of G's vertices by exit time

    visited.assign(n, false);

    // first series of depth first searches
    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs(i, adj, order);

    // create adjacency list of G^T
    vector<vector<int> > adj_rev(n);
    for (int v = 0; v < n; v++)
        for (int u : adj[v])
            adj_rev[u].push_back(v);

    visited.assign(n, false);
    reverse(order.begin(), order.end());

    vector<int> roots(n, 0);
    // gives the root vertex of a vertex's SCC

    // second series of depth first searches
    for (auto v : order)
        if (!visited[v]) {
            std::vector<int> component;
            dfs(v, adj_rev, component);
            components.push_back(component);
            int root = *min_element(begin(component), end(component
                ));
            // can choose any element in the component!!!.
            for (auto u : component)
                roots[u] = root;
        }

    // add edges to condensation graph
    adj_cond.assign(n, {});
    for (int v = 0; v < n; v++)
        for (auto u : adj[v])
            if (roots[v] != roots[u])
                adj_cond[roots[v]].push_back(roots[u]);
}
```

4.4 Bridges And Articulation Points

```
// Official version
#include <bits/stdc++.h>
using namespace std;
const int maxN = 10010;

int n, m;
bool joint[maxN];
int timeDfs = 0, bridge = 0;
int low[maxN], num[maxN];
vector <int> g[maxN];

void dfs(int u, int pre) {
    int child = 0;
    num[u] = low[u] = ++timeDfs;
    for (int v : g[u]) {
        if (v == pre) continue;
        if (!num[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] == num[v]) bridge++;
            child++;
            if (pre != -1 && low[v] >= num[u]) joint[u] = true;
        }
        else low[u] = min(low[u], num[v]);
    }
    if (pre == -1) {
        if (child > 1) joint[u] = true;
    }
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    for (int i = 1; i <= n; i++)
        if (!num[i]) dfs(i, -1);

    int cntJoint = 0;
    for (int i = 1; i <= n; i++) cntJoint += joint[i];

    cout << cntJoint << ' ' << bridge;
}
```

4.5 Eulerian path

```
struct Edge;
typedef list<Edge>::iterator iter;

struct Edge
{
  int next_vertex;
  iter reverse_edge;

  Edge(int next_vertex)
    :next_vertex(next_vertex)
    { }
};

const int max_vertices = ;
int num_vertices;
list<Edge> adj[max_vertices]; // adjacency list

vector<int> path;

void find_path(int v)
{
  while(adj[v].size() > 0)
  {
    int vn = adj[v].front().next_vertex;
    adj[vn].erase(adj[v].front().reverse_edge);
    adj[v].pop_front();
    find_path(vn);
  }
  path.push_back(v);
}

void add_edge(int a, int b)
{
  adj[a].push_front(Edge(b));
  iter ita = adj[a].begin();
  adj[b].push_front(Edge(a));
  iter itb = adj[b].begin();
  ita->reverse_edge = itb;
  itb->reverse_edge = ita;
}
```

4.6 Finding centroid

```
#include <iostream>
#include <vector>
using namespace std;
const int maxn = 200010;
int n;
vector<int> adj[maxn];
int subtree_size[maxn];

int get_subtree_size(int node, int parent = -1) {
  int &res = subtree_size[node];
  res = 1;
  for (int i : adj[node]) {
    if (i == parent) { continue; }
    res += get_subtree_size(i, node);
  }
```

```
  return res;
}

int get_centroid(int node, int parent = -1) {
  for (int i : adj[node]) {
    if (i == parent) { continue; }

    if (subtree_size[i] * 2 > n) { return get_centroid(i, node); }
  }
  return node;
}

int main() {
  cin >> n;
  for (int i = 0; i < n - 1; i++) {
    int a, b;
    cin >> a >> b;
    a--;
    b--;
    adj[a].push_back(b);
    adj[b].push_back(a);
  }

  get_subtree_size(0);
  cout << get_centroid(0) + 1 << endl;
}
```

4.7 Centroid Decomposition

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

// a number that is large enough while not causing overflow
const int INF = 1e9;

vector<vector<int>> adj;
vector<int> subtree_size;
// min_dist[v] := the minimal distance between v and a red node
vector<int> min_dist;
vector<bool> is_removed;
vector<vector<pair<int, int>>> ancestors;

int get_subtree_size(int node, int parent = -1) {
  subtree_size[node] = 1;
  for (int child : adj[node]) {
    if (child == parent || is_removed[child]) { continue; }
    subtree_size[node] += get_subtree_size(child, node);
  }
  return subtree_size[node];
}

int get_centroid(int node, int tree_size, int parent = -1) {
  for (int child : adj[node]) {
    if (child == parent || is_removed[child]) { continue; }
    if (subtree_size[child] * 2 > tree_size) {
      return get_centroid(child, tree_size, node);
    }
  }
  return node;
}

/**
 * Calculate the distance between current 'node' and the
 * centroid' it belongs to. The distances between a node and
 * all its centroid ancestors are stored in the vector
 * 'ancestors'.
 * Arguments:
 *  cur_dist: the distance between 'node' and 'centroid'
 */
void get_dists(int node, int centroid, int parent = -1, int
      cur_dist = 1)
{
  for (int child : adj[node]) {
    if (child == parent || is_removed[child]) { continue; }
    cur_dist++;
    get_dists(child, centroid, node, cur_dist);
    cur_dist--;
  }
  ancestors[node].push_back({centroid, cur_dist});
}

void build_centroid_decomp(int node = 0) {
  int centroid = get_centroid(node, get_subtree_size(node));
  /*
   * For all nodes in the subtree rooted at 'centroid',
   * calculate their distances to the centroid
   */
  for (int child : adj[centroid]) {
    if (is_removed[child]) { continue; }
    get_dists(child, centroid, centroid);
  }

  is_removed[centroid] = true;
  for (int child : adj[centroid]) {
    if (is_removed[child]) { continue; }
    // build the centroid decomposition for all child components
    build_centroid_decomp(child);
  }
}

/**
 * Paint 'node' red by updating all of its ancestors' minimal
 * distances to a red node.
 */
void paint(int node) {
  for (auto &[ancestor, dist] : ancestors[node]) {
    min_dist[ancestor] = min(min_dist[ancestor], dist);
  }
  min_dist[node] = 0;
}

/** Print the minimal distance between 'node' to a red node */
void query(int node) {
  int ans = min_dist[node];
  for (auto &[ancestor, dist] : ancestors[node]) {
    if (!dist) { continue; }
    /*
     * The distance between 'node' and a red painted node
     * is the sum of the distance from 'node' to one of
     * its ancestors ('dist') and the distance from this
     * ancestor to the nearest red node
     * ('min_dist[ancestor]').
     */
    ans = min(ans, dist + min_dist[ancestor]);
  }

  cout << ans << "\n";
}

int main() {
  int N, M;
  cin >> N >> M;

  adj.assign(N, vector<int>());
  for (int i = 0; i < N - 1; i++) {
    int a, b;
    cin >> a >> b;
    a--, b--;
    adj[a].push_back(b);
    adj[b].push_back(a);
  }

  subtree_size.assign(N, 0);
  ancestors.assign(N, vector<pair<int, int>>());
  is_removed.assign(N, false);
  build_centroid_decomp();

  min_dist.assign(N, INF);
  paint(0);
  for (int i = 0; i < M; i++) {
    int t, v;
    cin >> t >> v;
    v--;
    if (t == 1) {
      paint(v);
    } else {
      query(v);
    }
  }
}
```

5 Combinatorial optimization

5.1 Dinic max-flow for sparse graph

```
#include <bits/stdc++.h>
using namespace std;
const int N = 5001;
#define fi first
#define se second
struct TEdge
{
    int v,rit; //rit: reverse edge
    long long cap,flow;
};

map<pair<int, int> , long long>  ww;
int n,m;
void enter()
{
    cin >> n >> m;
    for (int i=0,a,b,c; i<m; i++)
    {
        cin >> a >> b >> c;
        ww[{a,b}] += c;
        ww[{b,a}] += c;
    }
}

vector< TEdge> g[N];
void init()
{
    int ru, rv;
    for (pair<pair<int, int>, int> p: ww)
    {
        if (p.fi.fi < p.fi.se)
        {
            ru = g[p.fi.fi].size();
            rv = g[p.fi.se].size();
            g[p.fi.fi].push_back({p.fi.se, rv, p.se, 0});
            g[p.fi.se].push_back({p.fi.fi, ru, p.se, 0});
        }
    }
}

int MF = 1;
int tt;
int d[N], ni[N];

bool bfs()
{
    for (int i=1; i<=n; i++)
        d[i] = 0;
    d[1] = 1;
    queue<int> qu;
    int u;
    qu.push(1);
    while (!qu.empty())
    {
        u = qu.front();
        qu.pop();
        for (auto v : g[u])
        {
            if (!d[v.v])
            {
                if (v.flow + MF <= v.cap)
                {
                    d[v.v] = d[u] + 1;
                    qu.push(v.v);
                }
            }
        }
    }
    return d[n];
}


long long dfs (int u, long long ff)
{
    if (u ==n)
        return ff;
    for (;ni[u] < g[u].size(); ++ni[u])
    {
        if (d[g[u][ni[u]].v] == d[u] + 1)
        {
            int fff = dfs(g[u][ni[u]].v, min(g[u][ni[u]].cap - g[u
                ][ni[u]].flow, ff));
            if (fff >= MF)
            {
                g[u][ni[u]].flow += fff;
                g[g[u][ni[u]].v][g[u][ni[u]].rit].flow -= fff;
                return fff;
            }
        }
    }
    return 0;
}
```

```
long long max_flow()
{
    long long res = 0,d;
    MF = 1 << 30;
    while (MF)
    {
        while (bfs())
        {
            for (int i=1; i<=n; i++)
                ni[i] = 0;
            do
            {
                d = dfs(1, 1<<30);
                res += d;
            } while (d);
        }
        MF >>= 1;
    }
    return res;
}

int main()
{
    ios_base::sync_with_stdio(0);
    enter();
    init();
    cout << max_flow();
    return 0;
}
```

5.2 Min-cost max-flow

```
// Implementation of min cost max flow algorithm using adjacency
// matrix (Edmonds and Karp 1972).  This implementation keeps track
     of
// forward and reverse edges separately (so you can set cap[i][j]
     !=
// cap[j][i]).  For a regular max flow, set all edge costs to 0.
//
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - (maximum flow value, minimum cost value)
//     - To obtain the actual flow, look at positive values only.

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
  int N;
  VVL cap, flow, cost;
  VI found;
  VL dist, pi, width;
  VPII dad;

  MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

  void AddEdge(int from, int to, L cap, L cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
  }

  void Relax(int s, int k, L cap, L cost, int dir) {
    L val = dist[s] + pi[s] - pi[k] + cost;
    if (cap && val < dist[k]) {
      dist[k] = val;
      dad[k] = make_pair(s, dir);
      width[k] = min(cap, width[s]);
    }
  }

  L Dijkstra(int s, int t) {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);
    fill(width.begin(), width.end(), 0);
    dist[s] = 0;
    width[s] = INF;

    while (s != -1) {
      int best = -1;
      found[s] = true;
      for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
        Relax(s, k, flow[k][s], -cost[k][s], -1);
        if (best == -1 || dist[k] < dist[best]) best = k;
      }
      s = best;
    }

    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }

  pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};

// BEGIN CUT
// The following code solves UVA problem #10594: Data Flow

int main() {
  int N, M;

  while (scanf("%d%d", &N, &M) == 2) {
    VVL v(M, VL(3));
    for (int i = 0; i < M; i++)
      scanf("%Ld%Ld%Ld", &v[i][0], &v[i][1], &v[i][2]);
    L D, K;
    scanf("%Ld%Ld", &D, &K);

    MinCostMaxFlow mcmf(N+1);
    for (int i = 0; i < M; i++) {
      mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K, v[i][2]);
      mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K, v[i][2]);
    }
    mcmf.AddEdge(0, 1, D, 0);

    pair<L, L> res = mcmf.GetMaxFlow(0, N);

    if (res.first == D) {
      printf("%Ld\n", res.second);
    } else {
      printf("Impossible.\n");
    }
  }

  return 0;
}

// END CUT
```

5.3 Min Cost Matching

```
//
     ///////////////////////////////////////////////////////////////////

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


#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

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

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

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

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

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

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

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

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

    int j = 0;
    while (true) {

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

```
      }
      seen[j] = 1;

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

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

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

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

    mated++;
  }

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

  return value;
}
```

5.4 Push-relabel max-flow

```
// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic.  This implementation is
// significantly faster than straight Ford-Fulkerson.  It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).

#include <cmath>
#include <vector>
#include <iostream>
#include <queue>

using namespace std;

typedef long long LL;

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

struct PushRelabel {
  int N;
  vector<vector<Edge> > G;
  vector<LL> excess;
  vector<int> dist, active, count;
  queue<int> Q;

  PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N),
      count(2*N) {}

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

  void Enqueue(int v) {
    if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v);
        }
  }

  void Push(Edge &e) {
    int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
    if (dist[e.from] <= dist[e.to] || amt == 0) return;
    e.flow += amt;
    G[e.to][e.index].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    Enqueue(e.to);
  }

  void Gap(int k) {
    for (int v = 0; v < N; v++) {
      if (dist[v] < k) continue;
      count[dist[v]]--;
      dist[v] = max(dist[v], N+1);
      count[dist[v]]++;
      Enqueue(v);
    }
  }

  void Relabel(int v) {
    count[dist[v]]--;
    dist[v] = 2*N;
    for (int i = 0; i < G[v].size(); i++)
      if (G[v][i].cap - G[v][i].flow > 0)
dist[v] = min(dist[v], dist[G[v][i].to] + 1);
    count[dist[v]]++;
    Enqueue(v);
  }

  void Discharge(int v) {
    for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v]
        [i]);
    if (excess[v] > 0) {
      if (count[dist[v]] == 1)
Gap(dist[v]);
      else
Relabel(v);
    }
  }

  LL GetMaxFlow(int s, int t) {
    count[0] = N-1;
    count[N] = 1;
    dist[s] = N;
    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
      excess[s] += G[s][i].cap;
      Push(G[s][i]);
    }

    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
      active[v] = false;
      Discharge(v);
    }

    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
  }
};

// The following code solves SPOJ problem 4110: Fast Maximum Flow (
    FASTFLOW)
/* Input
The first line contains the two integers N and M. The next M lines
    each contain
three integers A, B, and C, denoting that there is an edge of
    capacity C (1 <= C <= 109) between nodes A and B (1 <= A, B
    <= N). Note that it is possible for there to be duplicate
    edges, as well as an edge from a node to itself.

Output
Print a single integer (which may not fit into a 32-bit integer)
    denoting the
maximum flow / minimum cut between 1 and N.

Example
Input:
4 6
1 2 3
2 3 4
3 1 2
2 2 5
3 4 3
4 3 3
Output:
5
*/
int main() {
  int n, m;
  scanf("%d%d", &n, &m);

  PushRelabel pr(n);
  for (int i = 0; i < m; i++) {
   int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    if (a == b) continue;
    pr.AddEdge(a-1, b-1, c);
    pr.AddEdge(b-1, a-1, c);
  }
  printf("%Ld\n", pr.GetMaxFlow(0, n-1));
  return 0;
}
```

5.5 Unweighted Bipartite Matching

```
// Max matching for unweighted bipartie graph
// Kuhn's algorithm O(n^2)
/*
Given a **bipartite graph** $G = (X \cup Y, E)$. The vertices of
    $X$ are denoted $x_1, x_2, \ldots, x_m$, and the vertices of
    $Y$ are denoted $y_1, y_2, \ldots, y_n$.

A **matching** on $G$ is a set of edges $E' \subseteq E$ such that
    no two edges in $E'$ share a common vertex.

**Requirement:** Find a **maximum matching** (having the most edges
    ) on $G$.

## Input
* **Line 1:** Contains two integers, $m$ and $n$ ($1 \le m, n \le
    100$).
* **Subsequent lines:** Each line contains two positive integers,
    $i$ and $j$, representing an edge $(x_i, y_j) \in E$.

## Output
* **Line 1:** The number of edges in the maximum matching found (
    $K$).
* **$K$ subsequent lines:** Each line contains two numbers, $u$ and
     $v$, representing the edge $(x_u, y_v)$ chosen for the
    maximum matching.
*/
#include <bits/stdc++.h>
using namespace std;

const int N = 102;

int n, m, Assigned[N];

int Visited[N], t = 0;

vector<int> a[N];

bool visit(int u) {
    if (Visited[u] != t)
        Visited[u] = t;
    else
        return false;
    for (int i = 0; i < a[u].size(); i++) {
        int v = a[u][i];
        if (!Assigned[v] || visit(Assigned[v])) {
            Assigned[v] = u;
```

```
            return true;
        }
    }
    return false;
}

int main() {

    scanf("%d%d", &m, &n);
    int x, y;
    while (scanf("%d%d", &x, &y) > 0)
        a[x].push_back(y);
    int Count = 0;
    for (int i = 1; i <= m; i++) {
        t++;
        Count += visit(i);
    }

    printf("%d\n", Count);
    for (int i = 1; i <= n; i++)
        if (int j = Assigned[i])
            printf("%d %d\n", j, i);
}
```

5.6 Global min-cut

```
// Adjacency matrix implementation of Stoer-Wagner min cut
     algorithm.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//
// OUTPUT:
//     - (min cut value, nodes in half of min cut)

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

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

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;

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

// BEGIN CUT
// The following code solves UVA problem #10989: Bomb, Divide and
     Conquer
int main() {
  int N;
  cin >> N;
  for (int i = 0; i < N; i++) {
    int n, m;
    cin >> n >> m;
    VVI weights(n, VI(n));
    for (int j = 0; j < m; j++) {
      int a, b, c;
      cin >> a >> b >> c;
      weights[a-1][b-1] = weights[b-1][a-1] = c;
    }
    pair<int, VI> res = GetMinCut(weights);
    cout << "Case #" << i+1 << ": " << res.first << endl;
  }
}
// END CUT
```

5.7 Graph cut inference

```
// Special-purpose {0,1} combinatorial optimization solver for
// problems of the following by a reduction to graph cuts:
//
//        minimize         sum_i  psi_i(x[i])
//  x[1]...x[n] in {0,1}      + sum_{i < j}  phi_{ij}(x[i], x[j])
//
// where
//      psi_i : {0, 1} --> R
//   phi_{ij} : {0, 1} x {0, 1} --> R
//
// such that
//   phi_{ij}(0,0) + phi_{ij}(1,1) <= phi_{ij}(0,1) + phi_{ij}(1,0)
     (*)
//
// This can also be used to solve maximization problems where the
// direction of the inequality in (*) is reversed.
//
// INPUT: phi -- a matrix such that phi[i][j][u][v] = phi_{ij}(u, v
     )
//        psi -- a matrix such that psi[i][u] = psi_i(u)
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution
//
// To use this code, create a GraphCutInference object, and call
     the
// DoInference() method.  To perform maximization instead of
     minimization,
// ensure that #define MAXIMIZATION is enabled.

#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef vector<VVI> VVVI;
typedef vector<VVVI> VVVVI;

const int INF = 1000000000;

// comment out following line for minimization
#define MAXIMIZATION

struct GraphCutInference {
  int N;
  VVI cap, flow;
  VI reached;

  int Augment(int s, int t, int a) {
    reached[s] = 1;
    if (s == t) return a;
    for (int k = 0; k < N; k++) {
      if (reached[k]) continue;
      if (int aa = min(a, cap[s][k] - flow[s][k])) {
  if (int b = Augment(k, t, aa)) {
    flow[s][k] += b;
    flow[k][s] -= b;
    return b;
  }
      }
    }
    return 0;
  }

  int GetMaxFlow(int s, int t) {
    N = cap.size();
    flow = VVI(N, VI(N));
    reached = VI(N);

    int totflow = 0;
    while (int amt = Augment(s, t, INF)) {
      totflow += amt;
      fill(reached.begin(), reached.end(), 0);
    }
    return totflow;
  }

  int DoInference(const VVVVI &phi, const VVI &psi, VI &x) {
    int M = phi.size();
    cap = VVI(M+2, VI(M+2));
    VI b(M);
    int c = 0;

    for (int i = 0; i < M; i++) {
      b[i] += psi[i][1] - psi[i][0];
      c += psi[i][0];
      for (int j = 0; j < i; j++)
  b[i] += phi[i][j][1][1] - phi[i][j][0][1];
      for (int j = i+1; j < M; j++) {
  cap[i][j] = phi[i][j][0][1] + phi[i][j][1][0] - phi[i][j][0][0] -
     phi[i][j][1][1];
  b[i] += phi[i][j][1][0] - phi[i][j][0][0];
  c += phi[i][j][0][0];
      }
    }

#ifdef MAXIMIZATION
    for (int i = 0; i < M; i++) {
      for (int j = i+1; j < M; j++)
  cap[i][j] *= -1;
      b[i] *= -1;
    }
    c *= -1;
#endif

    for (int i = 0; i < M; i++) {
      if (b[i] >= 0) {
  cap[M][i] = b[i];
      } else {
  cap[i][M+1] = -b[i];
  c += b[i];
      }
    }

    int score = GetMaxFlow(M, M+1);
    fill(reached.begin(), reached.end(), 0);
    Augment(M, M+1, INF);
    x = VI(M);
    for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;
    score += c;
#ifdef MAXIMIZATION
    score *= -1;
#endif

    return score;
  }

};

int main() {

  // solver for "Cat vs. Dog" from NWERC 2008

  int numcases;
  cin >> numcases;
  for (int caseno = 0; caseno < numcases; caseno++) {
    int c, d, v;
    cin >> c >> d >> v;

    VVVVI phi(c+d, VVVI(c+d, VVI(2, VI(2))));
    VVI psi(c+d, VI(2));
    for (int i = 0; i < v; i++) {
      char p, q;
      int u, v;
      cin >> p >> u >> q >> v;
      u--; v--;
      if (p == 'C') {
  phi[u][c+v][0][0]++;
  phi[c+v][u][0][0]++;
      } else {
  phi[v][c+u][1][1]++;
  phi[c+u][v][1][1]++;
      }
```

```
    }

    GraphCutInference graph;
    VI x;
    cout << graph.DoInference(phi, psi, x) << endl;
  }

  return 0;
}
```

6 Data structures

6.1 Binary Indexed Tree

```
#include <iostream>
using namespace std;

#define LOGSZ 17

int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);

// add v to value at x
void set(int x, int v) {
  while(x <= N) {
    tree[x] += v;
    x += (x & -x);
  }
}

// get cumulative sum up to and including x
int get(int x) {
  int res = 0;
  while(x) {
    res += tree[x];
    x -= (x & -x);
  }
  return res;
}

// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
  int idx = 0, mask = N;
  while(mask && idx < N) {
    int t = idx + mask;
    if(x >= tree[t]) {
      idx = t;
      x -= tree[t];
    }
    mask >>= 1;
  }
  return idx;
}
```

6.2 DSU rollback

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

// BeginCodeSnip{DSU}
class DSU {
  private:
  vector<ll> p, sz, sum;
  // stores all history info related to merges
  vector<pair<ll &, ll>> history;

  public:
  DSU(int n) : p(n), sz(n, 1), sum(n) { iota(p.begin(), p.end(), 0)
      ; }

  void init_sum(const vector<ll> a) {
    for (int i = 0; i < (int)a.size(); i++) { sum[i] = a[i]; }
  }

  int get(int x) { return (p[x] == x) ? x : get(p[x]); }

  ll get_sum(int x) { return sum[get(x)]; }

  void unite(int a, int b) {
    a = get(a);
    b = get(b);
    if (a == b) { return; }
    if (sz[a] < sz[b]) { swap(a, b); }

    // add to history
    history.push_back({p[b], p[b]});
    history.push_back({sz[a], sz[a]});
    history.push_back({sum[a], sum[a]});

    p[b] = a;
    sz[a] += sz[b];
    sum[a] += sum[b];
  }

  void add(int x, ll v) {
    x = get(x);
    history.push_back({sum[x], sum[x]});
    sum[x] += v;
  }

  int snapshot() { return history.size(); }

  void rollback(int until) {
    while (snapshot() > until) {
      history.back().first = history.back().second;
      history.pop_back();
    }
  }
};
// EndCodeSnip

const int MAXN = 3e5;

DSU dsu(MAXN);

struct Query {
  int t, u, v, x;
};

vector<Query> tree[MAXN * 4];

void update(Query &q, int v, int query_l, int query_r, int tree_l,
     int tree_r) {
  if (query_l > tree_r || query_r < tree_l) { return; }
  if (query_l <= tree_l && query_r >= tree_r) {
    tree[v].push_back(q);
    return;
  }
  int m = (tree_l + tree_r) / 2;
  update(q, v * 2, query_l, query_r, tree_l, m);
  update(q, v * 2 + 1, query_l, query_r, m + 1, tree_r);
}

void dfs(int v, int l, int r, vector<ll> &ans) {
  int snapshot = dsu.snapshot();
  // perform all available operations upon entering
  for (Query &q : tree[v]) {
    if (q.t == 1) { dsu.unite(q.u, q.v); }
    if (q.t == 2) { dsu.add(q.v, q.x); }
  }
  if (l == r) {
    // answer type 3 query if we have one
    for (Query &q : tree[v]) {
      if (q.t == 3) { ans[l] = dsu.get_sum(q.v); }
    }
  } else {
    // go deeper into the tree
    int m = (l + r) / 2;
    dfs(2 * v, l, m, ans);
    dfs(2 * v + 1, m + 1, r, ans);
  }
  // undo operations upon exiting
  dsu.rollback(snapshot);
}

int main() {
  int n, q;
  cin >> n >> q;
  vector<ll> a(n);
  for (int i = 0; i < n; i++) { cin >> a[i]; }
  dsu.init_sum(a);

  map<pair<int, int>, int> index_added;
  for (int i = 0; i < q; i++) {
    int t;
    cin >> t;
    if (t == 0) {
      int u, v;
      cin >> u >> v;
      if (u > v) swap(u, v);
      // store index this edge is added, marks beginning of
           interval
      index_added[{u, v}] = i;
    } else if (t == 1) {
      int u, v;
      cin >> u >> v;
      if (u > v) swap(u, v);
      Query cur_q = {1, u, v};
      // add all edges that are deleted to interval [index added, i
         - 1]
      update(cur_q, 1, index_added[{u, v}], i - 1, 0, q - 1);
      index_added[{u, v}] = -1;
    } else if (t == 2) {
      int v, x;
      cin >> v >> x;
      Query cur_q = {2, -1, v, x};
      // add all sum queries to interval [i, q - 1]
      update(cur_q, 1, i, q - 1, 0, q - 1);
    } else if (t == 3) {
      int v;
      cin >> v;
      Query cur_q = {3, -1, v};
      // add all output queries to interval [i, i]
      update(cur_q, 1, i, i, 0, q - 1);
    }
  }

  // add all edges that are not deleted to interval [index added, q
       - 1]
  for (auto [edge, index] : index_added) {
    if (index != -1) {
      Query cur_q = {1, edge.first, edge.second};
      update(cur_q, 1, index, q - 1, 0, q - 1);
    }
  }

  vector<ll> ans(q, -1);
  dfs(1, 0, q - 1, ans);
  for (int i = 0; i < q; i++) {
    if (ans[i] != -1) { cout << ans[i] << "\n"; }
  }
}
```

6.3 Segment Tree Beats

```
/* Given a size N integer sequence a0, a1, ..., aN-1. Process the
     following Q queries in order:

* 0 l r b: For each i=l, ..., r-1, a_i <- min(ai, b)
* 1 l r b: For each i=l, ..., r-1, a_i <- max(ai, b)
* 2 l r b: For each i=l, ..., r-1, a_i <- a_i + b
* 3 l r: Print Sum from i=l to r-1 of a_i
*/

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

const int MAXN = 200001;  // 1-based

int N;
ll A[MAXN];
#define llINF ((ll)1e18)

struct Node {
  ll sum;   // Sum tag
  ll max1;  // Max value
  ll max2;  // Second Max value
  ll maxc;  // Max value count
  ll min1;  // Min value
  ll min2;  // Second Min value
  ll minc;  // Min value count
  ll lazy;  // Lazy tag
} T[MAXN * 4];

void merge(int t) {
  // sum
  T[t].sum = T[t << 1].sum + T[t << 1 | 1].sum;

  // max
  if (T[t << 1].max1 == T[t << 1 | 1].max1) {
```

```
    T[t].max1 = T[t << 1].max1;
    T[t].max2 = max(T[t << 1].max2, T[t << 1 | 1].max2);
    T[t].maxc = T[t << 1].maxc + T[t << 1 | 1].maxc;
  } else {
    if (T[t << 1].max1 > T[t << 1 | 1].max1) {
      T[t].max1 = T[t << 1].max1;
      T[t].max2 = max(T[t << 1].max2, T[t << 1 | 1].max1);
      T[t].maxc = T[t << 1].maxc;
    } else {
      T[t].max1 = T[t << 1 | 1].max1;
      T[t].max2 = max(T[t << 1].max1, T[t << 1 | 1].max2);
      T[t].maxc = T[t << 1 | 1].maxc;
    }
  }

  // min
  if (T[t << 1].min1 == T[t << 1 | 1].min1) {
    T[t].min1 = T[t << 1].min1;
    T[t].min2 = min(T[t << 1].min2, T[t << 1 | 1].min2);
    T[t].minc = T[t << 1].minc + T[t << 1 | 1].minc;
  } else {
    if (T[t << 1].min1 < T[t << 1 | 1].min1) {
      T[t].min1 = T[t << 1].min1;
      T[t].min2 = min(T[t << 1].min2, T[t << 1 | 1].min1);
      T[t].minc = T[t << 1].minc;
    } else {
      T[t].min1 = T[t << 1 | 1].min1;
      T[t].min2 = min(T[t << 1].min1, T[t << 1 | 1].min2);
      T[t].minc = T[t << 1 | 1].minc;
    }
  }
}

void push_add(int t, int tl, int tr, ll v) {
  if (v == 0) { return; }
  T[t].sum += (tr - tl + 1) * v;
  T[t].max1 += v;
  if (T[t].max2 != -llINF) { T[t].max2 += v; }
  T[t].min1 += v;
  if (T[t].min2 != llINF) { T[t].min2 += v; }
  T[t].lazy += v;
}

// corresponds to a chmin update
void push_max(int t, ll v, bool l) {
  if (v >= T[t].max1) { return; }
  T[t].sum -= T[t].max1 * T[t].maxc;
  T[t].max1 = v;
  T[t].sum += T[t].max1 * T[t].maxc;
  if (l) {
    T[t].min1 = T[t].max1;
  } else {
    if (v <= T[t].min1) {
      T[t].min1 = v;
    } else if (v < T[t].min2) {
      T[t].min2 = v;
    }
  }
}

// corresponds to a chmax update
void push_min(int t, ll v, bool l) {
  if (v <= T[t].min1) { return; }
  T[t].sum -= T[t].min1 * T[t].minc;
  T[t].min1 = v;
  T[t].sum += T[t].min1 * T[t].minc;
  if (l) {
    T[t].max1 = T[t].min1;
  } else {
    if (v >= T[t].max1) {
      T[t].max1 = v;
    } else if (v > T[t].max2) {
      T[t].max2 = v;
    }
  }
}

void pushdown(int t, int tl, int tr) {
  if (tl == tr) return;
  // sum
  int tm = (tl + tr) >> 1;
  push_add(t << 1, tl, tm, T[t].lazy);
  push_add(t << 1 | 1, tm + 1, tr, T[t].lazy);
  T[t].lazy = 0;

  // max
  push_max(t << 1, T[t].max1, tl == tm);
  push_max(t << 1 | 1, T[t].max1, tm + 1 == tr);

  // min
  push_min(t << 1, T[t].min1, tl == tm);
  push_min(t << 1 | 1, T[t].min1, tm + 1 == tr);
}

void build(int t = 1, int tl = 0, int tr = N - 1) {
  T[t].lazy = 0;
  if (tl == tr) {
    T[t].sum = T[t].max1 = T[t].min1 = A[tl];
    T[t].maxc = T[t].minc = 1;
    T[t].max2 = -llINF;
    T[t].min2 = llINF;
    return;
  }

  int tm = (tl + tr) >> 1;
  build(t << 1, tl, tm);
  build(t << 1 | 1, tm + 1, tr);
  merge(t);
}

void update_add(int l, int r, ll v, int t = 1, int tl = 0, int tr =
      N - 1) {
  if (r < tl || tr < l) { return; }
  if (l <= tl && tr <= r) {
    push_add(t, tl, tr, v);
    return;
  }
  pushdown(t, tl, tr);

  int tm = (tl + tr) >> 1;
  update_add(l, r, v, t << 1, tl, tm);
  update_add(l, r, v, t << 1 | 1, tm + 1, tr);
  merge(t);
}

void update_chmin(int l, int r, ll v, int t = 1, int tl = 0, int tr
      = N - 1) {
  if (r < tl || tr < l || v >= T[t].max1) { return; }
  if (l <= tl && tr <= r && v > T[t].max2) {
    push_max(t, v, tl == tr);
    return;
  }
  pushdown(t, tl, tr);

  int tm = (tl + tr) >> 1;
  update_chmin(l, r, v, t << 1, tl, tm);
  update_chmin(l, r, v, t << 1 | 1, tm + 1, tr);
  merge(t);
}

void update_chmax(int l, int r, ll v, int t = 1, int tl = 0, int tr
      = N - 1) {
  if (r < tl || tr < l || v <= T[t].min1) { return; }
  if (l <= tl && tr <= r && v < T[t].min2) {
    push_min(t, v, tl == tr);
    return;
  }
  pushdown(t, tl, tr);

  int tm = (tl + tr) >> 1;
  update_chmax(l, r, v, t << 1, tl, tm);
  update_chmax(l, r, v, t << 1 | 1, tm + 1, tr);
  merge(t);
}

ll query_sum(int l, int r, int t = 1, int tl = 0, int tr = N - 1) {
  if (r < tl || tr < l) { return 0; }
  if (l <= tl && tr <= r) { return T[t].sum; }
  pushdown(t, tl, tr);

  int tm = (tl + tr) >> 1;
  return query_sum(l, r, t << 1, tl, tm) + query_sum(l, r, t << 1 |
      1, tm + 1, tr);
}

int main() {
  int Q;

  cin >> N >> Q;
  for (int i = 0; i < N; i++) { cin >> A[i]; }
  build();
  for (int q = 0; q < Q; q++) {
    int t;
    cin >> t;
    if (t == 0) {
      int l, r;
      ll x;
      cin >> l >> r >> x;
      update_chmin(l, r - 1, x);
    } else if (t == 1) {
      int l, r;
      ll x;
      cin >> l >> r >> x;
      update_chmax(l, r - 1, x);
    } else if (t == 2) {
      int l, r;
      ll x;
      cin >> l >> r >> x;
      update_add(l, r - 1, x);
    } else if (t == 3) {
      int l, r;
      cin >> l >> r;
      cout << query_sum(l, r - 1) << '\n';
    }
  }
}
```

7 String algorithms

7.1 Suffix array

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

struct SA {
    string s;
    vector<int> p;
    int n;

    SA (string s) : s(s) {
        s = s + "$";
        n = s.size();
        p.resize(n);

        for (int i=0; i<n; ++i)
            p[i] = i;

        sort (p.begin(), p.end(), [&] (int a, int b) {
            return s[a] < s[b];
        });
        vector<int> rank(n, 0);
        for (int i=0; i<n; ++i) {
            rank[i] = lower_bound(p.begin(), p.end(), i, [&] (int a
                , int b) {
                return s[a] < s[b];
            }) - p.begin();
        }

        vector<int> rank_new(n), p_new(n), cnt(n);
        for (int k=1; k<n; k*=2) {
            for (int i = 0; i < n; i++) {
                p_new[i] = p[i] - k;
                if (p_new[i] < 0) p_new[i] += n;
            }
            cnt.assign(n, 0); rank_new.assign(n, 0);
            for (int i = 0; i < n; i++)
                cnt[rank[p_new[i]]]++;
            for (int i = 1; i < n; i++)
                cnt[i] += cnt[i-1];
            for (int i = n-1; i >= 0; i--)
                p[--cnt[rank[p_new[i]]]] = p_new[i];
            rank_new[p[0]] = 0;
            int classes = 0;
            for (int i = 1; i < n; i++) {
                pair<int, int> cur = {rank[p[i]], rank[(p[i] + k) %
                     n]};
                pair<int, int> prev = {rank[p[i-1]], rank[(p[i-1] +
                     k) % n]};
                if (cur != prev)
                    ++classes;
                rank_new[p[i]] = classes;
            }
            rank.swap(rank_new);
        }
    }
};

// Input "pppppplppp" -> Output "9 5 8 4 7 3 6 2 1 0"
// "ababba" -> "6 5 0 2 4 1 3"
int main() {
    ios_base::sync_with_stdio(0);
```

```
  cin.tie(0);
  string s; cin >> s;
  SA sa(s);
  for (auto x : sa.p) cout << x << " ";
}
```

7.2 Knuth-Morris-Pratt

```
/*
Finds all occurrences of the pattern string p within the
text string t. Running time is O(n + m), where n and m
are the lengths of p and t, respectively.
*/

#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef vector<int> VI;

void buildPi(string& p, VI& pi)
{
  pi = VI(p.length());
  int k = -2;
  for(int i = 0; i < p.length(); i++) {
    while(k >= -1 && p[k+1] != p[i])
      k = (k == -1) ? -2 : pi[k];
    pi[i] = ++k;
  }
}

int KMP(string& t, string& p)
{
  VI pi;
  buildPi(p, pi);
  int k = -1;
  for(int i = 0; i < t.length(); i++) {
    while(k >= -1 && p[k+1] != t[i])
      k = (k == -1) ? -2 : pi[k];
    k++;
    if(k == p.length() - 1) {
      // p matches t[i-m+1, ..., i]
      cout << "matched at index " << i-k << ": ";
      cout << t.substr(i-k, p.length()) << endl;
      k = (k == -1) ? -2 : pi[k];
    }
  }
  return 0;
}

int main()
{
  string a = "AABAACAADAABAABA", b = "AABA";
  KMP(a, b); // expected matches at: 0, 9, 12
  return 0;
}
```

7.3 Z function

```
vector<int> z_function(string s) {
    int n = s.size();
    vector<int> z(n);
    int l = 0, r = 0;
    for(int i = 1; i < n; i++) {
        if(i < r) {
            z[i] = min(r - i, z[i - l]);
        }
        while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
        if(i + z[i] > r) {
            l = i;
            r = i + z[i];
        }
    }
    return z;
}
```

8 Miscellaneous

8.1 Longest increasing subsequence

```
// Given a list of numbers of length n, this routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
//   INPUT: a vector of integers
//   OUTPUT: a vector containing the longest increasing subsequence

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG

VI LongestIncreasingSubsequence(VI v) {
  VPII best;
  VI dad(v.size(), -1);

  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
    VPII::iterator it = lower_bound(best.begin(), best.end(), item)
        ;
    item.second = i;
#else
    PII item = make_pair(v[i], i);
    VPII::iterator it = upper_bound(best.begin(), best.end(), item)
        ;
#endif
    if (it == best.end()) {
      dad[i] = (best.size() == 0 ? -1 : best.back().second);
      best.push_back(item);
    } else {
      dad[i] = it == best.begin() ? -1 : prev(it)->second;
      *it = item;
    }
  }

  VI ret;
  for (int i = best.back().second; i >= 0; i = dad[i])
    ret.push_back(v[i]);
  reverse(ret.begin(), ret.end());
  return ret;
}
```

8.2 Dates

```
// Routines for performing computations on dates.  In these
//     routines,
// months are expressed as integers from 1 to 12, days are
//     expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.

#include <iostream>
#include <string>

using namespace std;

string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "
    Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt (int m, int d, int y){
  return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/
//     day/year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;

  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay (int jd){
  return dayOfWeek[jd % 7];
}

int main (int argc, char **argv){
  int jd = dateToInt (3, 24, 2004);
  int m, d, y;
  intToDate (jd, m, d, y);
  string day = intToDay (jd);

  // expected output:
  //    2453089
  //    3/24/2004
  //    Wed
  cout << jd << endl
    << m << "/" << d << "/" << y << endl
    << day << endl;
}
```

8.3 C++ input/output

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    // Ouput a specific number of digits past the decimal point,
    // in this case 5
    cout.setf(ios::fixed); cout << setprecision(5);
    cout << 100.0/7.0 << endl;
    cout.unsetf(ios::fixed);

    // Output the decimal point and trailing zeros
    cout.setf(ios::showpoint);
    cout << 100.0 << endl;
    cout.unsetf(ios::showpoint);

    // Output a '+' before positive values
    cout.setf(ios::showpos);
    cout << 100 << " " << -100 << endl;
    cout.unsetf(ios::showpos);

    // Output numerical values in hexadecimal
    cout << hex << 100 << " " << 1000 << " " << 10000 << dec <<
        endl;
}
```

8.4 Latitude/longitude

```
/*
Converts from rectangular coordinates to latitude/longitude and
    vice
versa. Uses degrees (not radians).
*/

#include <iostream>
#include <cmath>

using namespace std;

struct ll
{
  double r, lat, lon;
```

```
};

struct rect
{
  double x, y, z;
};

ll convert(rect& P)
{
  ll Q;
  Q.r = sqrt(P.x*P.x+P.y*P.y+P.z*P.z);
  Q.lat = 180/M_PI*asin(P.z/Q.r);
  Q.lon = 180/M_PI*acos(P.x/sqrt(P.x*P.x+P.y*P.y));

  return Q;
}

rect convert(ll& Q)
{
  rect P;
  P.x = Q.r*cos(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.y = Q.r*sin(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.z = Q.r*sin(Q.lat*M_PI/180);

  return P;
}

int main()
{
  rect A;
  ll B;

  A.x = -1.0; A.y = 2.0; A.z = -3.0;

  B = convert(A);
  cout << B.r << " " << B.lat << " " << B.lon << endl;

  A = convert(B);
  cout << A.x << " " << A.y << " " << A.z << endl;
}
```

8.5 Random STL stuff

```
// Example for using stringstreams and next_permutation

#include <algorithm>
#include <iostream>
#include <sstream>
#include <vector>

using namespace std;

int main(void){
  vector<int> v;

  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);

  // Expected output: 1 2 3 4
  //                  1 2 4 3
  //                  ...
  //                  4 3 2 1
  do {
    ostringstream oss;
    oss << v[0] << " " << v[1] << " " << v[2] << " " << v[3];

    // for input from a string s,
    //   istringstream iss(s);
    //   iss >> variable;

    cout << oss.str() << endl;
  } while (next_permutation (v.begin(), v.end()));

  v.clear();

  v.push_back(1);
  v.push_back(2);
  v.push_back(1);
  v.push_back(3);

  // To use unique, first sort numbers.  Then call
  // unique to place all the unique elements at the beginning
  // of the vector, and then use erase to remove the duplicate
  // elements.
  sort(v.begin(), v.end());
  v.erase(unique(v.begin(), v.end()), v.end());

  // Expected output: 1 2 3
  for (size_t i = 0; i < v.size(); i++)
    cout << v[i] << " ";
  cout << endl;
}
```

8.6 Longest common subsequence

```
#include<bits/stdc++.h>
using namespace std;
int main() {
    int n; cin >> n; int m; cin >> m;
    vector<int> a(n); for (int &x:a) cin >> x;
    vector<int> b(m); for (int &x:b) cin >> x;

    vector< vector< int > > f(n+1, vector<int> (m+1, 0));
    for (int i=0; i<n; ++i) {
        for (int j=0; j<m; ++j) {
            if (a[i] == b[j]) {
                f[i+1][j+1] = f[i][j] + 1;
            } else {
                f[i+1][j+1] = max(f[i][j+1], f[i+1][j]);
            }
        }
    }

    cout << f[n][m] << endl;
    int x=n,y=m; vector<int> trace;
    while (x>0&&y>0) {
        if (a[x-1] == b[y-1]) {
            trace.push_back(a[x-1]);
            x--;y--;
        } else if (f[x][y] == f[x-1][y]) x--;
        else y--;
    }
    for (int i=trace.size()-1; i>=0; --i) cout << trace[i] << " ";
}
```

8.7 Miller-Rabin Primality Test (C)

```
// Randomized Primality Test (Miller-Rabin):
//   Error rate: 2^(-TRIAL)
//   Almost constant time. srand is needed

#include <stdlib.h>
#define EPS 1e-7

typedef long long LL;

LL ModularMultiplication(LL a, LL b, LL m)
{
  LL ret=0, c=a;
  while(b)
  {
    if(b&1) ret=(ret+c)%m;
    b>>=1; c=(c+c)%m;
  }
  return ret;
}
LL ModularExponentiation(LL a, LL n, LL m)
{
  LL ret=1, c=a;
  while(n)
  {
    if(n&1) ret=ModularMultiplication(ret, c, m);
    n>>=1; c=ModularMultiplication(c, c, m);
  }
  return ret;
}
bool Witness(LL a, LL n)
{
  LL u=n-1;
  int t=0;
  while(!(u&1)){u>>=1; t++;}
  LL x0=ModularExponentiation(a, u, n), x1;
  for(int i=1;i<=t;i++)
  {
    x1=ModularMultiplication(x0, x0, n);
    if(x1==1 && x0!=1 && x0!=n-1) return true;
    x0=x1;
  }
  if(x0!=1) return true;
  return false;
}
LL Random(LL n)
{
  LL ret=rand(); ret*=32768;
  ret+=rand(); ret*=32768;
  ret+=rand(); ret*=32768;
  ret+=rand();
  return ret%n;
}
bool IsPrimeFast(LL n, int TRIAL)
{
  while(TRIAL--)
  {
    LL a=Random(n-2)+1;
    if(Witness(a, n)) return false;
  }
  return true;
}
```

8.8 Super Duper Fast IO

```
#include <memory.h>

#include <cstdio>

const int BUF_SIZE = 65536;
char input[BUF_SIZE];

struct scanner {
    char* curPos;

    scanner() {
        fread(input, 1, sizeof(input), stdin);
        curPos = input;
    }

    void ensureCapacity() {
        int size = input + BUF_SIZE - curPos;
        if (size < 100) {
            memcpy(input, curPos, size);
            fread(input + size, 1, BUF_SIZE - size, stdin);
            curPos = input;
        }
    }

    int nextInt() {
        ensureCapacity();
        while (*curPos <= ' ')
            ++curPos;
        bool sign = false;
        if (*curPos == '-') {
            sign = true;
            ++curPos;
        }
        int res = 0;
        while (*curPos > ' ')
            res = res * 10 + (*(curPos++) & 15);
        return sign ? -res : res;
    }

    char nextChar() {
        ensureCapacity();
        while (*curPos <= ' ')
            ++curPos;
        return *(curPos++);
    }
};

int main() {
    scanner sc;
    int a = sc.nextInt();
    char b = sc.nextChar();

    printf("%d %c\n", a, b);
}
```

8.9 FFT

```
// Convolution using the fast Fourier transform (FFT).
//
// INPUT:
//     a[1...n]
//     b[1...m]
//
// OUTPUT:
//     c[1...n+m-1] such that c[k] = sum_{i=0}^k a[i] b[k-i]
//
// Alternatively, you can use the DFT() routine directly, which
     will
// zero-pad your input to the next largest power of 2 and compute
     the
// DFT or inverse DFT.

#include <iostream>
#include <vector>
#include <complex>

using namespace std;

typedef long double DOUBLE;
typedef complex<DOUBLE> COMPLEX;
typedef vector<DOUBLE> VD;
typedef vector<COMPLEX> VC;

struct FFT {
  VC A;
  int n, L;

  int ReverseBits(int k) {
    int ret = 0;
    for (int i = 0; i < L; i++) {
      ret = (ret << 1) | (k & 1);
      k >>= 1;
    }
    return ret;
  }

  void BitReverseCopy(VC a) {
    for (n = 1, L = 0; n < a.size(); n <<= 1, L++) ;
    A.resize(n);
    for (int k = 0; k < n; k++)
      A[ReverseBits(k)] = a[k];
  }

  VC DFT(VC a, bool inverse) {
    BitReverseCopy(a);
    for (int s = 1; s <= L; s++) {
      int m = 1 << s;
      COMPLEX wm = exp(COMPLEX(0, 2.0 * M_PI / m));
      if (inverse) wm = COMPLEX(1, 0) / wm;
      for (int k = 0; k < n; k += m) {
  COMPLEX w = 1;
  for (int j = 0; j < m/2; j++) {
    COMPLEX t = w * A[k + j + m/2];
    COMPLEX u = A[k + j];
    A[k + j] = u + t;
    A[k + j + m/2] = u - t;
    w = w * wm;
  }
      }
    }
    if (inverse) for (int i = 0; i < n; i++) A[i] /= n;
    return A;
  }

  // c[k] = sum_{i=0}^k a[i] b[k-i]
  VD Convolution(VD a, VD b) {
    int L = 1;
    while ((1 << L) < a.size()) L++;
    while ((1 << L) < b.size()) L++;
    int n = 1 << (L+1);

    VC aa, bb;
    for (size_t i = 0; i < n; i++) aa.push_back(i < a.size() ?
         COMPLEX(a[i], 0) : 0);
    for (size_t i = 0; i < n; i++) bb.push_back(i < b.size() ?
         COMPLEX(b[i], 0) : 0);

    VC AA = DFT(aa, false);
    VC BB = DFT(bb, false);
    VC CC;
    for (size_t i = 0; i < AA.size(); i++) CC.push_back(AA[i] * BB[
         i]);
    VC cc = DFT(CC, true);

    VD c;
    for (int i = 0; i < a.size() + b.size() - 1; i++) c.push_back(
         cc[i].real());
    return c;
  }

};

int main() {
  double a[] = {1, 3, 4, 5, 7};
  double b[] = {2, 4, 6};

  FFT fft;
  VD c = fft.Convolution(VD(a, a + 5), VD(b, b + 3));

  // expected output: 2 10 26 44 58 58 42
  for (int i = 0; i < c.size(); i++) cerr << c[i] << " ";
  cerr << endl;

  return 0;
}
```