Team Reference Document

HaKings

Alfredo Altamirano Montealvo
Carlos Salvador Garza Garza
Diego Emilio Gutierrez Yepiz

Contents

1 vimrc

```
syntax on
inoremap jj <ESC>
colorscheme elflord
set ai si sw=4 ts=4
set nu
set backspace=start,indent,eol
set clipboard=unnamed
set ignorecase
set smartcase
set incsearch
set scrolloff=3
highlight linenr ctermbg=darkblue
set hi=100
set nowrap
```

2 Header

```
#include <bits/stdc++.h>
#define _ ios_base::sync_with_stdio(0), cin.tie(0), cin.tie(0), cout.tie(0);
#define INF 1000000000
#define FOR(i, a, b) for(int i=int(a); i<int(b); i++)
#define FORC(cont, it) for(decltype((cont).begin()) it=(cont).begin(); it!=(cont).end(); it++)
#define pb push_back
#define mp make_pair
#define eb emplace_back
#define fi first
#define se second
#define all(x) (x).begin(), (x).end()
using namespace std; typedef long long ll; typedef pair<int, int> ii; typedef vector<int> vi; typedef vector<ii> vii; typedef vector<vi> vvi;
```

3 Primes

```

#define SIZE 1000000
bitset<SIZE> sieve;
void buildSieve() {
  sieve.set();
  sieve[0] = sieve[1] = 0;
  int root = sqrt(SIZE);
  FOR(i, 2, root+1)
    if (sieve[i])
      for(int j = i*i; j < SIZE; j+=i)
        sieve[j] = 0;
}

vi primesList;
void buildPrimesList() {
  if(!sieve[2])
    buildSieve();
  primesList.reserve(SIZE/log(SIZE));
  FOR(i, 2, SIZE+1)
    if(sieve[i])
      primesList.pb(i);
}

vii primeFactorization(int N) {
  vii factors;
  int idx = 0, pf = primesList[0];
  while(pf*pf <= N) {
    while(N%pf==0) {
      N /= pf;
      if(factors.size() && factors.back().first == pf)
        factors.back().second++;
      else
        factors.pb(ii(pf, 1));
    }
    pf = primesList[++idx];
   }
  if(N!=1) factors.pb(ii(N, 1));
  return factors;
}

void getDivisors(vii pf, int d, int index, vi &div) {
  if (index == pf.size()) {
    div.pb(d);
    return;
  }
  for (int i = 0; i <= pf[index].second; i++) {
    getDivisors(pf, d, index+1, div);
    d *= pf[index].first;
  }
  return;
}

vi divisors(ll N) {
  vii pf = primeFactorization(N);
  vi div;
  getDivisors(pf, 1ll, 0, div);
  sort(div.begin(), div.end());
  return div;
}

bool isPrime(int n) {
  if(n < 2) return false;
  if(n == 2 || n == 3) return true;
  if(!(n&1 && n%3)) return false;
  long long sqrtN = sqrt(n)+1;
  for(long long i = 6LL; i <= sqrtN; i += 6)
    if(!(n%(i-1)) || !(n%(i+1))) return false;
  return true;
}
```

4 Segment Tree

```
struct SegmentTree {
  vi t; int N;
  SegmentTree(vi &values) {
    N = values.size();
    t.assign(N<<1, 0);
    for(int i = 0; i < N; i++) t[i+N] = values[i];
    for(int i = N-1; i; --i) t[i] = combine(t[i<<1], t[i<<1|1]);
  }
  int combine(int a, int b) { return a+b; }
  void set(int index, int value) {
    t[index+N] = value;
    for(int i = (index+N)>>1; i; i >>= 1) t[i] = combine(t[i<<1], t[i<<1|1]);
  }
  int query(int from, int to) {
    int ansL = 0, ansR = 0;
    for(int l = N+from, r = N+to; l<r; l >>= 1, r >>= 1) {
      if (l&1) ansL = combine(ansL, t[l++]);
      if (r&1) ansR = combine(ansR, t[--r]);
    }
    return combine(ansL, ansR);
  }
};

struct LazySegmentTree {
  vi t, d; int n, h;
  LazySegmentTree(vi &values) {
    n = values.size();
    h = sizeof(int) * 8 - __builtin_clz(n);
    t.assign(n<<1, 0), d.assign(n, 0);
    for(int i = 0; i < N; i++) t[i+N] = values[i];
    build(i+N, n<<1);
  }
  void calc(int p, int k) {
    if (d[p] == 0) t[p] = t[p<<1] + t[p<<1|1];
    else t[p] = d[p] * k;
  }
  void apply(int p, int value, int k) {
    t[p] = value * k;
    if (p < n) d[p] = value;
  }
  void push(int l, int r) {
    int s = h, k = 1 << (h-1);
    for (l += n, r += n-1; s > 0; --s, k >>= 1)
      for (int i = l >> s; i <= r >> s; ++i) if (d[i]) {
        apply(i<<1, d[i], k);
        apply(i<<1|1, d[i], k);
        d[i] = 0;
      }
  }
  void build(int l, int r) {
    int k = 2;
    for (l += n, r += n-1; l; k <<= 1) {
      l >>= 1, r >>= 1;
      for (int i = r; i >= l; --i) calc(i, k);
    }
  }
  void modify(int l, int r, int value) {
    if (value == 0) return;
    push(l, l + 1); push(r - 1, r);
    int l0 = l, r0 = r, k = 1;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1, k <<= 1) {
      if (l&1) apply(l++, value, k);
      if (r&1) apply(--r, value, k);
    }
    build(l0, l0 + 1);
    build(r0 - 1, r0);
  }
  int query(int l, int r) {
    push(l, l + 1); push(r - 1, r);
    int res = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l&1) res += t[l++];
      if (r&1) res += t[--r];
    }
    return res;
  }
};
```

5 Geometry

5.1 Point

```
const double PI = 2*asin(1);

bool eq(double a, double b) { return fabs(a-b) < EPS; }
bool les(double a, double b) { return !eq(a, b) && a < b; }
struct Point {
  double x, y, z;
  Point() : x(0), y(0), z(0) {}
  Point(double x, double y) : x(x), y(y), z(0) {}
  Point(double x, double y, double z) : x(x), y(y), z(z) {}
  bool operator <(const Point &p) const {
        return les(x, p.x) || (eq(x, p.x) && les(y, p.y)) || (eq(x, p.x) && eq(y, p.y) && les(z, p.z));
  }
    bool operator==(const Point &p) {
        return eq(x, p.x) && eq(y, p.y) && eq(z, p.z);
    }
};

double DEG_to_RAD(double deg) {
    return deg/180*2*asin(1);
}

double dist(Point p1, Point p2) {
  return sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2) + pow(p1.z-p2.z, 2)); }

Point rotate(Point p, double theta) {
  double rad = DEG_to_RAD(theta);
  return Point(p.x*cos(rad) - p.y*sin(rad),
        p.x*sin(rad) + p.y*cos(rad));
}

double ANG(double rad) { return rad*180/PI; }
double angulo(Point p) {
  double d = atan(double(p.y)/p.x);
  if(p.x < 0)
    d += PI;
  else if(p.y < 0)
    d += 2*PI;
  return ANG(d);
}
```

5.2 Vector

```
struct Vec {
  double x, y, z;
  Vec(double x, double y, double z) : x(x), y(y), z(z) {}
  Vec() : x(0), y(0), z(0) {}
  Vec(double x, double y) : x(x), y(y), z(0) {}
  Vec(Point a, Point b) : x(b.x-a.x), y(b.y-a.y), z(b.z-a.z) {}
};

Vec toVec(Point a, Point b){
  return Vec(a, b); }

Vec scale(Vec v, double s) {
  return Vec(v.x*s, v.y*s, v.z*s); }

Point translate(Point p, Vec v) {
  return Point(p.x+v.x, p.y+v.y, p.z+v.z); }

double dot(Vec a, Vec b) {
  return (a.x*b.x + a.y*b.y + a.z*b.z); }

double norm_sq(Vec v) {
  return v.x*v.x + v.y*v.y + v.z*v.z; }

//angle in radians
Vec rotate(Vec v, double angle) {
  Matrix rotation = CREATE(2, 2);
  rotation[0][0] = rotation[1][1] = cos(angle);
  rotation[1][0] = sin(angle);
  rotation[0][1] = -rotation[1][0];

  Matrix vec = CREATE(2, 1);
  vec[0][0] = v.x, vec[0][1] = v.y;

  Matrix res = multiply(rotation, vec);
  Vec result(res[0][0], res[0][1]);
  return result;
}

double cross (Vec a, Vec b) { return a.x*b.y - a.y*b.x; }

// returns true if r is on the left side of line pq
bool ccw(Point p, Point q, Point r){
  return cross(toVec(p, q), toVec(p, r)) > 0; }

bool collinear(Point p, Point q, Point r) {
  return abs(cross(toVec(p, q), toVec(p, r))) < EPS; }

double angle(Point a, Point o, Point b) { // returns angle aob in rad
  Vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}
```

5.3 Triangle

```
struct Triangle {
  Point A, B, C;
  Triangle() {}
  Triangle(Point A, Point B, Point C) : A(A), B(B), C(C) {}
};

double perimeter(double a, double b, double c) { return a+b+c; }

// Heron's formula
double area(double a, double b, double c){
  double s = perimeter(a, b, c)*0.5;
  return sqrt(s*(s-a)*(s-b)*(s-c));
}

double area(const Triangle &T) {
    double ab = dist(T.A, T.B);
    double bc = dist(T.B, T.C);
    double ca = dist(T.C, T.A);
    return area(ab, bc, ca);
}

double rInCircle(double ab, double bc, double ca){
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(Point a, Point b, Point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

bool inCircle(Point p1, Point p2, Point p3, Point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if(abs(r) < EPS) return false;
  Line l1, l2;
  double ratio = dist(p1, p2) / dist(p1, p3);
  Point p = translate(p2, scale(toVec(p2, p3), ratio/(1+ratio)));
  l1 = Line(p1, p);
  ratio = dist(p2, p1) / dist(p2, p3);
  l2 = Line(p2, p);
  areIntersect(l1, l2, ctr);
  return true;
}

double rCircumCircle(double ab, double bc, double ca) { return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

Point circumcenter(const Triangle &T) {
    Point A = T.A, B = T.B, C = T.C;
  double D = 2*(A.x*(B.y - C.y) + B.x*(C.y - A.y) + C.x*(A.y - B.y));
  double AA = A.x*A.x + A.y*A.y, BB = B.x*B.x + B.y*B.y, CC = C.x*C.x + C.y*C.y;
  return Point((AA*(B.y - C.y) + BB*(C.y - A.y) + CC*(A.y - B.y)) / D, (AA*(C.x - B.x) + BB*(A.x - C.x) + CC*(B.x - A.x)) / D);
}
```

5.4 Lines

```
struct Line {
  double a, b, c;
  Line() : a(0), b(0), c(0) {}
  Line(Point p1, Point p2) {
    if(abs(p1.x-p2.x) < EPS) {
      a = 1.0; b = 0.0; c = -p1.x;
    } else {
      a = -(double)(p1.y-p2.y)/(p1.x-p2.x);
      b = 1.0;
      c = -(double)(a*p1.x)-p1.y;
    }
  }
};

bool areParallel(Line l1, Line l2) {
  return (abs(l1.a-l2.a) < EPS) && (abs(l1.b-l2.b) < EPS); }

bool areSame(Line l1, Line l2) {
  return areParallel(l1, l2) && (abs(l1.c-l2.c) < EPS); }

bool areIntersect(Line l1, Line l2, Point &p) {
  if (areParallel(l1, l2)) return false;
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  if (abs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                 p.y = -(l2.a * p.x + l2.c);
  return true;
}

// Interseccion de AB con CD
// * WARNING: Does not work for collinear line segments!
bool lineSegIntersect(Point a, Point b, Point c, Point d) {
  double ucrossv1 = cross(toVec(a, b), toVec(a, c));
  double ucrossv2 = cross(toVec(a, b), toVec(a, d));
  if (ucrossv1 * ucrossv2 > 0) return false;
  double vcrossu1 = cross(toVec(c, d), toVec(c, a));
  double vcrossu2 = cross(toVec(c, d), toVec(c, b));
  return (vcrossu1 * vcrossu2 <= 0);
}

// Calcula la distancia de un punto P a una recta AB, y guarda en C la inters
double distToLine(Point p, Point a, Point b, Point &c) {
  Vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));
  return dist(p, c);
}

// Distancia a de P a segmento AB
double distToLineSegment(Point p, Point a, Point b, Point &c) {
  Vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = a; return dist(p, a); }
  if (u > 1.0) { c = b; return dist(p, b); }
  return distToLine(p, a, b, c);
}
```

5.5 Circles

```
bool circle2PtsRad(Point p1, Point p2, double r, Point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true;
} // to get the other center, reverse p1 and p2
```

5.6 Polygons

```
typedef vector<Point> Polygon;

ll cross(const Point &O, const Point &A, const Point &B) {
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

Polygon convexHull(Polygon &P) {
  int n = P.size(), k = 0;
  Polygon H(2*n);
  sort(P.begin(), P.end());
  FOR(i, 0, n) {
    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;
}

// return area when Points are in cw or ccw, p[0]  = p[n-1]
double area(const Polygon &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1*y2-x2*y1);
  }
  return abs(result) / 2.0;
}

bool isConvex(const Polygon &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;
  bool isLeft = ccw(P[0], P[1], P[2]);
  for (int i = 1; i < sz-1; i++)
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;
  return true;
}

// works for convex and concave
bool inPolygon (Point pt, const Polygon &P) {
  if((int)P.size() == 0) return false;
  double sum = 0;
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
      sum += angle(P[i], pt, P[i+1]);
    else sum -= angle(P[i], pt, P[i+1]); }
  return abs(abs(sum) - 2*PI) < EPS;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool isSimple(const Polygon &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 (lineSegIntersect(p[i], p[j], p[k], p[l]))
    return false;
  }
  }
  return true;
}

Point lineIntersectSeg(Point p, Point q, Point A, Point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x*A.y - A.x*B.y;
  double u = abs(a*p.x + b*p.y + c);
  double v = abs(a*q.x + b*q.y + c);
  return Point((p.x*v + q.x*u) / (u+v), (p.y*v + q.y*u) / (u+v));
}

// cuts polygon Q along line AB
Polygon cutPolygon(Point a, Point b, const Polygon &Q) {
  Polygon P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i+1])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.pb(Q[i]);
    if (left1 * left2 < -EPS)
      P.pb(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.pb(P.front());
  return P;
}

// only works for convex
bool pointInPolygon(Polygon &p1, Point p) {
  FOR(i, 0, p1.size() - 1)
    if (cross(p1[i], p1[i+1], p) >= 0)
      return false;
  return true;
}

// polygons must be convex
// returns polygon with size < 3 if there is no intersection
Polygon intersection(Polygon &p1, Polygon &p2) {
  set<Point> result;
  FOR(i, 0, p1.size() - 1) {
    if (pointInPolygon(p2, p1[i]))
      result.insert(p1[i]);
    FOR(j, 0, p2.size() - 1) {
      Line l1 = Line(p1[i], p1[i+1]);
      Line l2 = Line(p2[j], p2[j+1]);
      vector<Point> ps1, ps2;
      ps1.pb(p1[i]); ps1.pb(p1[i+1]);
      ps2.pb(p2[j]); ps2.pb(p2[j+1]);
      sort(ps1.begin(), ps1.end());
      sort(ps2.begin(), ps2.end());
      if (!areParallel(l1, l2)) {
        Point intersect;
        bool b = areIntersect(l1, l2, intersect);
        if (b && checkPointInSegm(intersect, ps1[0], ps1[1]) && checkPointInSegm(intersect, ps2[0], ps2[1]))
          result.insert(intersect);
      } else if (areSame(l1, l2)) {
        if (ps1[1] >= ps2[0] && ps2[1] >= ps1[0]) {
          vector<Point> ps3;
          ps3.pb(ps1[0]); ps3.pb(ps1[1]); ps3.pb(ps2[0]); ps3.pb(ps2[1]);
          sort(all(ps3));
          result.insert(ps3[1]);
          result.insert(ps3[2]);
        }
      }
    }
  }

  FOR(i, 0, p2.size() - 1) {
    if (pointInPolygon(p1, p2[i]))
      result.insert(p2[i]);
  }

  if (result.size() <= 2) {
    return Polygon(result.begin(), result.end());
  }

  Polygon p(result.begin(), result.end());
  return convexHull(p);
}
```

6 Suffix Array

```
struct SuffixArray {
  vi sa, lcp;
  int N, Q = 1<<7;
  vector<int> csort(vector<ii> &val) {
    #define get(t, num) ((num) ? ((t).fi) : ((t).se))
    vi currentOrder(N, 0), nextOrder(N, 0);
    FOR(i, 0, N) currentOrder[i] = i;
    auto cur = &currentOrder, nex = &nextOrder;
    FOR(j, 0, 2) {
      vi freq(N, 0), rank(N, 0), count(N, 0);
      FOR(i, 0, N) freq[get(val[i], j)]++;
      for(int i = 0, k = 0; i < N; i++) rank[i] = k, k += freq[i];
      FOR(i, 0, N) (*nex)[rank[get(val[(*cur)[i]], j)]++] = (*cur)[i];
      swap(cur, nex);
    }
    return *cur;
  }
  SuffixArray(char *S, int N) : N(N) {
    sa.assign(N, 0);
    FOR(i, 0, N) sa[i] = i;
    vi freq(Q, 0);
    int index[Q], rank[N], k = 0;
    FOR(i, 0, N) freq[S[i]]++;
    FOR(i, 0, Q) index[i] = k, k += freq[i];
    FOR(i, 0, N) rank[i] = index[S[i]];
    for(int len = 2; len <= N*2; len <<= 1) {
      vii val;
      FOR(i, 0, N) val.eb(rank[i], (i + len/2 >= N) ? 0 : rank[i + len/2]);
      vi order = csort(val);
      FOR(i, 0, N) rank[order[i]] = i && val[order[i]] == val[order[i-1]] ? rank[order[i-1]] : i;
    }
    FOR(i, 0, N) sa[rank[i]] = i;
    buildLCP(S);
  }
  void buildLCP(char *S) {
    vi phi(N), plcp(N);
    int L = 0;
    phi[sa[0]] = -1;
    FOR(i, 1, N) phi[sa[i]] = sa[i-1];
    FOR(i, 0, N) {
      if(phi[i] == -1) { plcp[i] = 0; continue; }
      while(S[i+L] == S[phi[i]+L]) L++;
      plcp[i] = L;
      L = max(L-1, int(0));
    }
    FOR(i, 0, N) lcp.pb(plcp[sa[i]]);
  }
};
```

7 Linear Suffix Array

```

/* Linear Suffix Array
  int N = 6, SA[6];
  char S[6] = "abcab";
  SA_IS((unsigned char*)S, SA, N, 256);
  FOR(i, 0, N) cout << S+SA[i] << endl;
*/
#include <unistd.h>

unsigned char mask[] = { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };
#define tget(i) ( (t[(i)/8]&mask[(i)%8]) ? 1 : 0 )
```

```
#define tset(i, b) t[(i)/8]=(b) ? (mask[(i)%8]|t[(i)/8]) : ((~mask[(i)%8])&t[(i)/8])
#define chr(i) (cs==sizeof(int)?((int*)s)[i]:((unsigned char *)s)[i])
#define isLMS(i) (i>0 && tget(i) && !tget(i-1))

void getBuckets(unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
  int i, sum = 0;
  for (i = 0; i <= K; i++) bkt[i] = 0;
  for (i = 0; i < n; i++) bkt[chr(i)]++;
  for (i = 0; i <= K; i++) {
    sum += bkt[i];
    bkt[i] = end ? sum : sum - bkt[i];
  }
}
void induceSAl(unsigned char *t, int *SA, unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
  int i, j;
  getBuckets(s, bkt, n, K, cs, end);
  for (i = 0; i < n; i++) {
    j = SA[i] - 1;
    if (j >= 0 && !tget(j))
      SA[bkt[chr(j)]++] = j;
  }
}
void induceSAs(unsigned char *t, int *SA, unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
  int i, j;
  getBuckets(s, bkt, n, K, cs, end);
  for (i = n - 1; i >= 0; i--) {
    j = SA[i] - 1;
    if (j >= 0 && tget(j)) SA[--bkt[chr(j)]] = j;
  }
}

void SA_IS(unsigned char *s, int *SA, int n, int K, int cs = 1) {
  int i, j;
  unsigned char *t = (unsigned char *) malloc(n / 8 + 1); // LS-type array in bits
  tset(n-2, 0);
  tset(n-1, 1);
  for (i = n - 3; i >= 0; i--)
    tset(i, (chr(i)<chr(i+1) || (chr(i)==chr(i+1) && tget(i+1)==1))?1:0);
  int *bkt = (int *) malloc(sizeof(int) * (K + 1));
  getBuckets(s, bkt, n, K, cs, 1);
  for (i = 0; i < n; i++) SA[i] = -1;
  for (i = 1; i < n; i++) if (isLMS(i))
    SA[--bkt[chr(i)]] = i;
  induceSAl(t, SA, s, bkt, n, K, cs, 0);
  induceSAs(t, SA, s, bkt, n, K, cs, 1);
  free(bkt);
  int n1 = 0;
  for (i = 0; i < n; i++)
    if (isLMS(SA[i]))
      SA[n1++] = SA[i];
  for (i = n1; i < n; i++)
    SA[i] = -1;
  int name = 0, prev = -1;
  for (i = 0; i < n1; i++) {
    int pos = SA[i];
    bool diff = 0;
    for (int d = 0; d < n; d++)
      if (prev == -1 || chr(pos+d) != chr(prev+d) || tget(pos+d) != tget(prev+d)) {
        diff = 1;
        break;
      } else if (d > 0 && (isLMS(pos+d) || isLMS(prev+d))) break;
    if (diff) {
      name++;
      prev = pos;
    }
    pos = (pos % 2 == 0) ? pos / 2 : (pos - 1) / 2;
    SA[n1 + pos] = name - 1;
  }
  for (i = n - 1, j = n - 1; i >= n1; i--) if (SA[i] >= 0)
    SA[j--] = SA[i];
  int *SA1 = SA, *s1 = SA + n - n1;
  if (name < n1) SA_IS((unsigned char*) s1, SA1, n1, name - 1, sizeof(int));
  else for (i = 0; i < n1; i++)
    SA1[s1[i]] = i;
  bkt = (int *) malloc(sizeof(int) * (K + 1));
  getBuckets(s, bkt, n, K, cs, true);
  for (i = 1, j = 0; i < n; i++) if (isLMS(i))
    s1[j++] = i;
  for (i = 0; i < n1; i++) SA1[i] = s1[SA1[i]];
  for (i = n1; i < n; i++) SA[i] = -1;
  for (i = n1 - 1; i >= 0; i--) {
    j = SA[i];
    SA[i] = -1;
    SA[--bkt[chr(j)]] = j;
  }
  induceSAl(t, SA, s, bkt, n, K, cs, 0);
  induceSAs(t, SA, s, bkt, n, K, cs, 1);
  free(bkt);
  free(t);
}
```

8 Trie

```

/* Trie
Constructs a tree for storing strings
*/
#define ALPHABET_SIZE 52
int getIndex(char c) {
  if(c >= 'A' && c <= 'Z')
    return c-'A';
  return c-'a'+26;
}

struct Trie {
  int words, prefixes;
  Trie *edges[ALPHABET_SIZE];
  Trie() : words(0), prefixes(0) { FOR(i, 0, ALPHABET_SIZE) edges[i] = 0; }
  ~Trie(){ FOR(i, 0, ALPHABET_SIZE) if(edges[i]) delete edges[i]; }
  void insert(char *word, int pos = 0) {
    if(word[pos] == 0) {
      words++;
      return;
    }
    prefixes++;
    int index = getIndex(word[pos]);
    if(edges[index] == 0)
      edges[index] = new Trie;
    edges[index]->insert(word, pos+1);
  }
  int countWords(char *word, int pos = 0) {
    if(word[pos] == 0)
      return words;
    int index = getIndex(word[pos]);
    if(edges[index]==0)
      return 0;
    return edges[index]->countWords(word, pos+1);
  }
  int countPrefix(char *word, int pos = 0) {
    if(word[pos] == 0)
      return prefixes;
    int index = getIndex(word[pos]);
    if(edges[index] == 0)
      return 0;
    return edges[index]->countPrefix(word, pos+1);
  }
};
```

9 DSU

```

struct UnionFindDS {
  vi tree;
  UnionFindDS(int n) { FOR(i, 0, n) tree.pb(i); }
  int root(int i) { return tree[i] == i ? i : tree[i] = root(tree[i]); }
  bool connected(int i, int j) {return root(i) == root(j);}
  void connect(int i, int j) { tree[root(i)] = tree[root(j)]; }
};

struct UnionFindDS2 {
  vi tree, sizes;
  int N;
  UnionFindDS2(int n) : N(n) {
    tree.reserve(n);
    FOR(i, 0, n) tree[i] = i;
    sizes.assign(n, 1);
  }
  int root(int i) { return (tree[i] == i) ? i : (tree[i] = root(tree[i]));}
  int countSets() { return N;}
  int getSize(int i) { return sizes[root(i)];}
  bool connected(int i, int j) { return root(i) == root(j);}
  void connect(int i, int j) {
    int ri = root(i), rj = root(j);
    if(ri != rj) {
      N--;
      sizes[rj] += sizes[ri];
      tree[ri] = rj;
    }
  }
};
```

10 LCA

```
struct LCA {
  vi order, height, index, st;
  int minIndex(int i, int j) {
    return height[i] < height[j] ? i : j;
  }
  LCA(Graph &g, ll root) {
    index.assign(g.V, -1);
    dfs(g, root, 0);
    st.assign(height.size()*2, 0);
    FOR(i, 0, height.size())
      st[height.size() + i] = i;
    for(int i = height.size()-1; i; i--)
      st[i] = minIndex(st[i<<1], st[i<<1|1]);
  }
  void dfs(Graph &g, ll cv, ll h) {
    index[cv] = order.size();
    order.pb(cv), height.pb(h);
    FORC(g.edges[cv], edge)
      if(index[edge->to] == -1) {
        dfs(g, edge->to, height.back() + 1);
        order.pb(cv), height.pb(h);
      }
  }
  ll query(ll i, ll j) {
    int from = index[i], to = index[j];
    if (from > to) swap(from, to);
    int idx = from;
    for(int l = from + height.size(), r = to + height.size() + 1; l < r; l >>= 1, r >>= 1) {
      if (l&1) idx = minIndex(idx, st[l++]);
      if (r&1) idx = minIndex(idx, st[--r]);
    }
    return order[idx];
  }
};
```

11 HLD

```

struct HeavyLightDecomposition {
  vector<vi> lists;
  vi values, listIndex, posIndex, parent, treeSizes;
  vector<SparseTable> sts;
  LCA *lca;
  HeavyLightDecomposition(Graph &g, vi values) : values(values) {
    lca = new LCA(g, 0);
    listIndex = posIndex = parent = treeSizes = vi(g.V, -1);
    getTreeSizes(g, 0);
    makeLists(g, 0, -1);
    FORC(lists, list) {
      vi v;
      FORC(*list, it) v.pb(values[*it]);
      sts.pb(SparseTable(v));
    }
  }
  ~HeavyLightDecomposition() { delete lca; }
  int getTreeSizes(Graph &g, int cv) {
    treeSizes[cv] = 1;
    FORC(g.edges[cv], edge) if(edge->to != parent[cv])
        parent[edge->to] = cv, treeSizes[cv] += getTreeSizes(g, edge->to);
    return treeSizes[cv];
  }
  void makeLists(Graph &g, int cv, int listNum) {
    if(listNum == -1)
      listNum = lists.size(), lists.pb(vi());
    listIndex[cv] = listNum;
    posIndex[cv] = lists[listNum].size();
    lists[listNum].pb(cv);
    int MAX = -1;
    FORC(g.edges[cv], edge) if(edge->to != parent[cv])
      if(MAX == -1 || treeSizes[edge->to] > treeSizes[MAX]) MAX = edge->to;
    FORC(g.edges[cv], edge) if(edge->to != parent[cv])
      makeLists(g, edge->to, edge->to == MAX ? listNum : -1);
  }
  int query(int from, int to) {
    int anc = lca->query(from, to), posLeft, posRight;
    int result = min(queryToAncestor(from, anc, posLeft), queryToAncestor(to, anc, posRight));
    if(posLeft < posRight) swap(posLeft, posRight);
    result = min(result, values[lists[listIndex[anc]][sts[listIndex[anc]].query(posIndex[anc], posRight)]]);
    if(posRight != posLeft)
      result = min(result, values[lists[listIndex[anc]][sts[listIndex[anc]].query(posRight+1, posLeft)]]);
    return result;
  }
  int queryToAncestor(int from, int anc, int &posInAncestorList) {
    int result = INF, left = from;
    while(listIndex[left] != listIndex[anc]) {
      result = min(result, values[lists[listIndex[left]][sts[listIndex[left]].query(0, posIndex[left])]]);
      left = parent[lists[listIndex[left]][0]];
    }
    posInAncestorList = posIndex[left];
    return result;
  }
};
```

12 Edmonds Karp

```
/* Edmonds-Karp
O(VE^2)
Finds a the maxflow from source to sink of a directed graph.
The weight of an edge denotes the capacity of the edge.
The negative weight edges are the edges with flow.
*/
int augment(MatrixGraph &g, int flow, vi &parent, int source, int cv, int minEdge) {
  if(cv == source)
    return minEdge;
  if(parent[cv] != -1) {
    flow = augment(g, flow, parent, source, parent[cv], min(minEdge, g.edges[parent[cv]][cv].weight));
    g.edges[parent[cv]][cv].weight -= flow;
    g.edges[cv][parent[cv]].weight += flow;
  }
  return flow;
}

int maxFlow(MatrixGraph &g, int source, int sink) {
  int mf = 0, flow = -1;
  while(flow) {
    vi distanceTo(g.V, INF);
    distanceTo[source] = 0;
    queue<int> q; q.push(source);
    vi parent(g.V, -1);
    while(!q.empty()) {
      int cv = q.front(); q.pop();
      if(cv == sink) break;
      FOR(i, 0, g.V)
        if(g.edges[cv][i].weight > 0 && distanceTo[i] == INF)
          distanceTo[i] = distanceTo[cv] + 1, q.push(i), parent[i] = cv;
    }
    mf += flow = augment(g, 0, parent, source, sink, INF);
  }
  return mf;
}
```

13 Max Bipartite Matching

```
struct MaxBipartiteMatching {
  int L, R;
  vvi edgesL;
  vi visitedL, matchR, matchL, inCoverL, inCoverR;
  MaxBipartiteMatching(int L, int R) : L(L), R(R) { edgesL.assign(L, vi()); }
  void addEdge(int l, int r) { edgesL[l].pb(r); }
  bool augment(int l) {
    if (visitedL[l]) return 0;
    visitedL[l] = 1;
    for (auto r: edgesL[l])
      if (matchR[r] == -1 || augment(matchR[r])) { matchR[r] = l; return 1; }
    return 0;
  }
  int maxMatching() {
    int ans = 0;
    matchR.assign(R, -1), matchL.assign(L, -1);
    for(int i = 0; i < L; i++)
      visitedL.assign(L, 0), ans += augment(i);
    for(int i = 0; i < R; i++) if (matchR[i] != -1) matchL[matchR[i]] = i;
    return ans;
  }
  void augment2(int l) {
    if (l == -1 || !inCoverL[l]) return;
    inCoverL[l] = 0;
    for (auto r: edgesL[l])
      if (!inCoverR[r]) inCoverR[r] = 1, augment2(matchR[r]);
  }
  void minCover() { // assuming matching found
    inCoverL.assign(L, 1), inCoverR.assign(R, 0);
    for(int i = 0; i < L; i++)
      if (matchL[i] == -1) augment2(i);
  }
};
```

14 Matrices

```
typedef vector<vector<double> > Matrix;
#define EPS 1E-7
#define CREATE(R, C) Matrix(R, vector<double>(C));

Matrix identity(int n) {
  Matrix m = CREATE(n, n);
  FOR(i, 0, n)
    m[i][i] = 1;
  return m;
}

Matrix multiply(Matrix m, double k) {
  FOR(i, 0, m.size())
    FOR(j, 0, m[0].size())
      m[i][j] *= k;
  return m;
}

Matrix multiply(Matrix m1, Matrix m2) {
  Matrix result = CREATE(m1.size(), m2[0].size());
  if(m1[0].size() != m2.size())
    return result;
  FOR(i, 0, result.size())
    FOR(j, 0, result[0].size())
      FOR(k, 0, m1[0].size())
        result[i][j] += m1[i][k]*m2[k][j];
  return result;
}

Matrix pow(Matrix m, int exp) {
  if(!exp) return identity(m.size());
  if(exp == 1) return m;
  Matrix result = identity(m.size());
  while(exp) {
    if(exp & 1) result = multiply(result, m);
    m = multiply(m, m);
    exp >>= 1;
  }
  return result;
}

//solves AX=B, output: A^-1 in A, X in B, returns det(A)
double gaussJordan(Matrix &a, Matrix &b) {
  int n = a.size(), m = b[0].size();
  vi irow(n), icol(n), ipiv(n);
  double det = 1;
  FOR(i, 0, n) {
    int pj = -1, pk = -1;
    FOR(j, 0, n) if (!ipiv[j])
      FOR(k, 0, n) if (!ipiv[k])
        if (pj == -1 || abs(a[j][k]) > abs(a[pj][pk])) { pj = j; pk = k; }
    if (abs(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;

    double c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    FOR(p, 0, n) a[pk][p] *= c;
    FOR(p, 0, m) b[pk][p] *= c;
    FOR(p, 0, n) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      FOR(q, 0, n) a[p][q] -= a[pk][q] * c;
      FOR(q, 0, m) b[p][q] -= b[pk][q] * c;
    }
  }
  for(int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    FOR(k, 0, n) swap(a[k][irow[p]], a[k][icol[p]]);
  }
  return det;
}

//returns the rank of a
int rref(Matrix &a) {
  int n = a.size(), m = a[0].size();
  int r = 0;
  FOR(c, 0, m) {
    int j = r;
    FOR(i, r+1, n)
      if (abs(a[i][c]) > abs(a[j][c])) j = i;
    if (abs(a[j][c]) < EPS) continue;
    swap(a[j], a[r]);
    double s = 1.0 / a[r][c];
    FOR(j, 0, m) a[r][j] *= s;
    FOR(i, 0, n) if (i != r) {
      double t = a[i][c];
      FOR(j, 0, m) a[i][j] -= t * a[r][j];
    }
    r++;
  }
  return r;
}
```

15 Dates

```
int toJulian(int day, int month, int year) {
  return 1461 * (year + 4800 + (month - 14) / 12) / 4 + 367 * (month - 2 -
    (month - 14) / 12 * 12) / 12 - 3 * ((year + 4900 + (month - 14) / 12)
    / 100) / 4 + day - 32075;
}

void toGregorian(int julian, int &day, int &month, int &year) {
  int x, n, i, j;
  x = julian + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  day = x - 2447 * j / 80;
  x = j / 11;
  month = j + 2 - 12 * x;
  year = 100 * (n - 49) + i + x;
}

bool isLeap(int year) { return (year%4 == 0 && year%100 != 0) || year%400 == 0; }
```

16 Articulation Points/Bridges

```
/* Articulation Points
O(V+E)
Finds all articulation points and bridges in a graph.
An articulation point is a vertex whose removal would disconnect the graph.
An bridge is a vertex whose removal disconnects the graph.
*/
vi low2, num2, parent, strongPoints;
int counter2, root, rootChildren;
void dfs1(Graph &g, int v) {
  low2[v] = num2[v] = counter2++;
  FORC(g.edges[v], edge) {
    if(num2[edge->to] == -1) {
      parent[edge->to] = v;
      if(v == root) rootChildren++;
      dfs1(g, edge->to);
      if(low2[edge->to] >= num2[v]) strongPoints[v] = true;
      if(low2[edge->to] > num2[v]) edge->strong = g.edges[edge->to][edge->backEdge].strong = true;
      low2[v] = min(low2[v], low2[edge->to]);
    } else if(edge->to != parent[v])
      low2[v] = min(low2[v], num2[edge->to]);
  }
}

vi articulationPointsAndBridges(Graph &g) {
  counter2 = 0;
  num2 = vi(g.V, -1), low2 = vi(g.V, 0), parent = vi(g.V, -1), strongPoints = vi(g.V, 0);
  FOR(i, 0, g.V)
    if(num2[i] == -1) {
      root = i, rootChildren = 0;
      dfs1(g, i);
      strongPoints[root] = rootChildren > 1;
    }
  return strongPoints;
}
```

17 SSC

```

/* Strongly Connected Components
O(V+E)
Partitions the vertices of a directed graph into strongly connected components.
A strongly connected component is a subset of a graph where every vertex is reachable from every other vertex.
Returns V where V_i is the index of the component of node i.
*/
vi low1, num1, components;
int counter1, SCCindex;
vector<bool> visited;
stack<int> S;

void dfs(Graph &g, int cv) {
  low1[cv] = num1[cv] = counter1++;
  S.push(cv);
  visited[cv] = true;
  FORC(g.edges[cv], edge) {
    if(num1[edge->to] == -1)
      dfs(g, edge->to);
    if(visited[edge->to])
      low1[cv] = min(low1[cv], low1[edge->to]);
  }
  if(low1[cv] == num1[cv]) {
    int index = SCCindex++;
    while(true) {
      int v = S.top(); S.pop(); visited[v] = 0;
      components[v] = index;
      if (cv == v)
        break;
    }
  }
}

vi stronglyConnectedComponents(Graph &g) {
  counter1 = 0, SCCindex = 0;
  visited = vector<bool>(g.V, 0);
  num1 = vi(g.V, -1), low1 = vi(g.V, 0), components = vi(g.V, 0);
  S = stack<int>();
  FOR(i, 0, g.V)
    if(num1[i] == -1)
      dfs(g, i);
  return components;
}
```

18 Catalan Numbers

```
int fact(int n) {
    return n ? n*fact(n-1) : 1;
}

int nthCatalan(int n) {
  return fact(2*n)/(pow(fact(n), 2)*(n+1));
}

int nextCatalan(int n, int previous) {
  return previous*2*(2*n+1)/(n+2);
}
```

19 Euclid

```

/* GCD
*/
int gcd(int a, int b) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}

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

/* Extended Euclid
Finds x,y such that d = ax + by.
Returns d = gcd(a,b).
*/
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;
}

/* Modular Linaer Equation Solver
Finds all solutions to ax = b (mod n)
*/
vi modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  vi solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    FOR(i, 0, d)
      solutions.pb(mod(x + i*(n/d), n));
  }
  return solutions;
}

/* Modular Inverse
```

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

/* Chinese Remainder Theorem
Returns \[x = a_i (mod n_i)\]
n's must be pairwise coprimes
*/
int chinese_remainder(int *n, int *a, int len) {
  int p, i, prod = 1, sum = 0;
  for (i = 0; i < len; i++) prod *= n[i];
  for (i = 0; i < len; i++) {
    p = prod / n[i];
    sum += a[i] * mod_inverse(p, n[i]) * p;
  }
  return sum % prod;
}
```

20 Miller Rabin

```
/* Miller-Rabin Primality Test
O(log(N)^3)
*/
ll mulmod(ll a, ll b, ll c) {
  ll x = 0, y = a % c;
  while (b) {
    if (b & 1) x = (x + y) % c;
    y = (y << 1) % c;
    b >>= 1;
  }
  return x % c;
}

ll fastPow(ll x, ll n, ll MOD) {
  ll ret = 1;
  while (n) {
    if (n & 1) ret = mulmod(ret, x, MOD);
    x = mulmod(x, x, MOD);
    n >>= 1;
  }
  return ret;
}

bool isPrime(ll n) {
  ll d = n - 1;
  int s = 0;
  while (d % 2 == 0) {
    s++;
    d >>= 1;
  }
  // It's garanteed that these values will work for any number smaller than 3*10**18 (3 and 18 zeros)
  int a[9] = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
  FOR(i, 0, 9) {
    bool comp = fastPow(a[i], d, n) != 1;
    if(comp) FOR(j, 0, s) {
      ll fp = fastPow(a[i], (1LL << (ll)j)*d, n);
      if (fp == n - 1) {
        comp = false;
        break;
      }
    }
    if(comp) return false;
  }
  return true;
}
```

21 Eulerian Path

```
/* Eulerian Path
O(V+E)
Partitions the vertices of a directed graph into strongly connected components.
A strongly connected component is a subset of a graph where every vertex is reachable from every other vertex.
Returns V where V_i is the index of the component of node i.
*/
vi low1, num1, components;
int counter1, SCCindex;
vector<bool> visited;
stack<int> S;

void dfs(Graph &g, int cv) {
  low1[cv] = num1[cv] = counter1++;
  S.push(cv);
  visited[cv] = true;
  FORC(g.edges[cv], edge) {
    if(num1[edge->to] == -1)
      dfs(g, edge->to);
    if(visited[edge->to])
      low1[cv] = min(low1[cv], low1[edge->to]);
  }
  if(low1[cv] == num1[cv]) {
    int index = SCCindex++;
    while(true) {
      int v = S.top(); S.pop(); visited[v] = 0;
      components[v] = index;
      if (cv == v)
        break;
    }
  }
}

vi stronglyConnectedComponents(Graph &g) {
  counter1 = 0, SCCindex = 0;
  visited = vector<bool>(g.V, 0);
  num1 = vi(g.V, -1), low1 = vi(g.V, 0), components = vi(g.V, 0);
  S = stack<int>();
  FOR(i, 0, g.V)
    if(num1[i] == -1)
      dfs(g, i);
  return components;
}
```

22 Nth Permutation

seq must be sorted

```
string nthPermutation(string seq, int permNum) {
  if(!seq.length()) return "";
  int f = fact(seq.length() - 1);
  int q = permNum/f, r = permNum%f;
  return seq[q] + nthPermutation(seq.substr(0, q) + seq.substr(q+1), r);
}
```

23 Shunting Yard

For parsing mathematical expressions specified in infix notation

```
void output(ostream &out, string x) {
    out << x << "␣";
}
string readToken(istream &in) {
    string t; int c;
    while((c = in.peek()) != EOF) {
        if(isalpha(c) || isdigit(c)) t.pb((char)c), in.get();
        else if(t != "") return t;
        else {in.get(); if(!isspace(c)) {t.pb((char)c); return t;}}
    } return t;
}

#define LEFT 0
#define RIGHT 1
#define isOp(x) (prec.find(x) != prec.end())
void shunting(istream &in, ostream &out) {
    string token;
    stack<string> ops;
    map<string, int> prec;
    prec["^"] = 6;
    prec["*"] = prec["/"] = prec["%"] = 5;
    prec["+"] = prec["-"] = 4;
    map<string, int> assoc; // default 0
    assoc["^"] = RIGHT;
    while((token = readToken(in)) != "") {
        if(isOp(token)) {
            while(!ops.empty() && isOp(ops.top())
            && ((assoc[token] == LEFT && prec[token] <= prec[ops.top()])
              || (assoc[token] == RIGHT && prec[token] < prec[ops.top()])))
                output(out, ops.top()), ops.pop();
            ops.push(token);
        } else if(token == "(") {
            ops.push(token);
        } else if(token == ")") {
            while(!ops.empty() && ops.top() != "(")
                output(out, ops.top()), ops.pop();
            // ops.empty() || ops.top() != "(" ====> MISMATCH
            ops.pop();
        } else // numbers vars
            output(out, token);
    }
    while(!ops.empty()) { // if ops.top() == ")"  || ops.top() == "(" =======> MISMATCH
        output(out, ops.top()), ops.pop();
    }
}
```

24 Sieve of Atkin

```
/* Sieve of Atkin
Obtains primes in the range [1, n]
*/
typedef vector<ll> vll;
vll primes;
void sieve_atkins(ll n) {
   vector<bool> isPrime(n + 1);
   isPrime[2] = isPrime[3] = true;
   for (ll i = 5; i <= n; i++)
      isPrime[i] = false;

   ll lim = ceil(sqrt(n));
   for (ll x = 1; x <= lim; x++) {
      for (ll y = 1; y <= lim; y++) {
         ll num = (4 * x * x + y * y);
         if (num <= n && (num % 12 == 1 || num % 12 == 5))
            isPrime[num] = true;
         num = (3 * x * x + y * y);
         if (num <= n && (num % 12 == 7))
            isPrime[num] = true;
         if (x > y) {
            num = (3 * x * x - y * y);
            if (num <= n && (num % 12 == 11))
               isPrime[num] = true;
         }
      }
   }

   for (ll i = 5; i <= lim; i++)
      if (isPrime[i])
         for (ll j = i * i; j <= n; j += i)
            isPrime[j] = false;

   for (ll i = 2; i <= n; i++)
      if (isPrime[i])
         primes.pb(i);
}
```

25 KMP

```
/* KMP
O(N+M)
Searches for a pattern in a string
*/
vi buildTable(string& pattern) {
  vi table(pattern.length()+1);
  int i = 0, j = -1, m = pattern.length();
  table[0] = -1;
  while(i < m) {
    while(j >= 0 && pattern[i] != pattern[j]) j = table[j];
    i++, j++;
    table[i] = j;
  }
  return table;
}

vi find(string& text, string& pattern) {
  vi matches;
  int i = 0, j = 0, n = text.length(), m = pattern.length();
  vi table = buildTable(pattern);
  while(i < n) {
    while(j >= 0 && text[i] != pattern[j]) j = table[j];
    i++, j++;
    if(j == m) {
      matches.pb(i-j);
      j = table[j];
    }
  }
  return matches;
}
```

26 Sparse Table

```
/* Sparse Table
O(N*log(N)) construction
O(1) queries
Answers RMQ
*/
struct SparseTable {
  vi A; vvi M;
  int log2(int n) { int i=0; while(n >>= 1) i++; return i; }
  SparseTable(vi arr) { //O(NlogN)
    int N = arr.size();
    A.assign(N, 0);
    M.assign(N, vi(log2(N)+1));
    int i, j;
    for(i=0; i<N; i++)
      M[i][0] = i, A[i] = arr[i];

    for(j=1; 1<<j <= N; j++)
      for(i=0; i + (1<<j) - 1 < N; i++)
        if(A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
          M[i][j] = M[i][j - 1];
        else
          M[i][j] = M[i + (1 << (j - 1))][j - 1];
  }
  //returns the index of the minimum value
  int query(int i, int j) {
    if(i > j) swap(i, j);
    int k = log2(j-i+1);
    if(A[M[i][k]] < A[M[j-(1 << k)+1][k]])
      return M[i][k];
    return M[j-(1 << k)+1][k];
  }
};
```

27 Fibonacci

```
/* Fibbonacci
*/
int fibn(int n) { //max 91
  double goldenRatio = (1+sqrt(5))/2;
  return round((pow(goldenRatio, n+1) - pow(1-goldenRatio, n+1))/sqrt(5));
}

int fibonacci(int n) {
  Matrix m = CREATE(2, 2);
  m[0][0] = 1, m[0][1] = 1, m[1][0] = 1, m[1][1] = 0;
  Matrix fib0 = CREATE(2, 1);
  fib0[0][0] = 1, fib0[1][0] = 1; //fib0 y fib1
  Matrix r = multiply(pow(m, n), fib0);
  return r[1][0];
}
```

28 Treap

```
#define LC(a) ((a) == ((a)->parent->left))
template<typename K>
struct Treap {
  struct Node {
    K key;
    int priority;
    Node *left, *right, *parent;};
  Node *root = 0;
  Treap(Node *root = 0) : root(root) { srand(time(0));
        }
  void fixDown(Node *n) {
    bool a, b;
    while((a = (n->left && n->priority < n->left->
          priority)) || (b = (n->right && n->priority
          < n->right->priority))) rotate(n, a && b ?
          rand()%2 : b); }
  Node *find(K key, bool leaf, Node *start, bool
        onlyLeft = 0) {
    if (!start) return 0;
    Node *n = start, *next;
    while((next = (key < n->key || onlyLeft ? n->left :
           n->right)) && (leaf || key != n->key)) n =
          next;
    return n; }
  void insert(K key, int priority = -1) {
    Node *p = find(key, 1, root), *n = new Node { key,
          priority == -1 ? rand()%100000 : priority ,
          0, 0, p };
    if (!root) { root = n; return; }
    (key < p->key ? p->left : p->right) = n;
    while(p && n->priority > p->priority) rotate(p, !LC
          (n)), p = n->parent; }
  void erase(K key) {
    if (!root) return;
    Node *n = find(key, 0, root), *del = n;
    if (!n || n->key != key) return;
    if (n->left && n->right) {
      del = find(key, 1, n->right, 1);
      n->key = del->key, n->priority = del->priority;
      fixDown(n); }
    if (del->left || del->right) (del->left ? del->left
           : del->right)->parent = del->parent;
    if (del->parent) (LC(del) ? del->parent->left : del
          ->parent->right) = del->left ? del->left :
          del->right;
    else root = del->left ? del->left : del->right;
    delete del; }
  void rotate(Node *n, bool left) {
    Node *u = (left ? n->right : n->left), *p = n->
          parent;
    if (p) (LC(n) ? p->left : p->right) = u;
    else root = u;
    Node *c = left ? u->left : u->right;
    (left ? n->right : n->left) = c;
    if (c) (left ? u->left : u->right)->parent = n;
    (left ? u->left : u->right) = n;
    u->parent = p, n->parent = u; }
};
```

29 LIS

```
/* Longest Increasing Subsequence
*/
vi longestIncreasingSubsequence(vi v) {
  vii best;
  vi parent(v.size(), -1);
  FOR(i, 0, v.size()) {
    ii item = ii(v[i], i);
    vii::iterator it = upper_bound(best.begin(), best.
          end(), item);
    if (it == best.end()) {
      parent[i] = (best.size() == 0 ? -1 : best.back().
            second);
      best.pb(item);
    } else {
      parent[i] = parent[it->second];
      *it = item;
    }
  }
  vi lis;
  for(int i=best.back().second; i >= 0; i=parent[i])
    lis.pb(v[i]);
  reverse(lis.begin(), lis.end());
  return lis;
}
```

30 Kadane

```
/* Maximum Subarray
*/
int maximumSubarray(int numbers[], int N) {
  int maxSoFar = numbers[0], maxEndingHere = numbers
        [0];
  FOR(i, 1, N) {
    if(maxEndingHere < 0) maxEndingHere = numbers[i];
    else maxEndingHere += numbers[i];
    maxSoFar = max(maxEndingHere, maxSoFar);
  }
  return maxSoFar;
}
int maxCircularSum (int a[], int n) {
   int max_kadane = maximumSubarray(a, n);
   int max_wrap  =  0;
   FOR(i, 0, n) {
        max_wrap += a[i];
        a[i] = -a[i];
   }
   max_wrap = max_wrap + maximumSubarray(a, n);
   return (max_wrap > max_kadane)? max_wrap :
          max_kadane;
}
```

31 Notes

```
printf("%ld\n", strtol("222", 0, x)); //base x to long
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
regmatch_t matches[1];
regcomp(&reg, pattern.c_str(), REG_EXTENDED|REG_ICASE);
if(regexec(&reg, str.c_str(), 1, matches, 0) == 0)
cout << "match" << endl;
regfree(&reg);
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
template <typename T>
string toString(T n) { ostringstream ss; ss << n;
       return ss.str(); }
template <typename T>
T toNum(const string &Text) { istringstream ss(Text); T
        result; return ss >> result ? result : 0; }
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
lower_bound: finds first that does not compare less
       than val.
upper_bound: finds first that compares greater than val
       .
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
do {} while(next_permutation(arr, arr+N));
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
scanf:
%d -> base10 int | %d+
%o -> base8 int  | %d+
%x -> base16 int | %d+
%a -> base10 or base16 double | ex. 123, 34.24,
       5464.324e+3, 53423E+2, 0x242.435, base16 if
       preceded by 0x
%c -> char or array of chars | ex. scanf("%c", &mychar)
        -> 'a', scanf("%4c", mycharptr) -> "asdf" (\0
       not included)
%s -> string
matching: scanf("abc%d", &myint) with input: "ab34␣
       ascz24␣abc345" would store 345 in myint, use %%
       to match %
%*d means match an int but dont store it in a parameter
%3d means match an integer but read only the 3 first
       characters
%lld stores in a long long int %d matches int, more
       specifiers are:
%le long double
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
Bellman-Ford for solving system of inequalities of the
       type x_i - x_j <= c
create a node for every x
create a source node
create a zero weight edge from s to every other node
for every inequality x_i - x_j <= c, make an edge from
       i to j of weight c
run bellman ford starting at s
the value for x_i is d_i
if there was a negative weight cycle, the system is
       inconsistent
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
In a bipartite graph, the size of the maximum
       independent set (or dominating set) = V-MCBM
In a bipartite graph, the size of the min vertex cover
       = MCBM
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
char str[] = "abc.␣sdfksm␣sgfda␣afdex..␣NJK-␣,,␣␣␣.␣␣
       hb564567....";
char * token = strtok(str, ".␣");
while (token != NULL) {
    printf ("%s\n",token);
    token = strtok (NULL, ".␣");
}
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
The number of pieces in which a circle is divided if n
       points on its circumference are joined by chords
        with no three internally concurrent:
g(n) = nCat4 + nCat2 + 1
A = i+b/2-1 where A is the area of a polygon, i is the
       number of integer points on the polygon and b is
        the number of integer points on the boundary
the number of spanning trees in complete a bipartite
       graph K(n, m) is m^(n-1) * n^(m-1)
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
//splitting by spaces
istringstream iss(line);
vector<string> tokens;
copy(istream_iterator<string>(iss), istream_iterator<
       string>(), back_inserter<vector<string> >(tokens
       ));
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
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
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
       tree_order_statistics_node_update> ordered_set;
*X.find_by_order(1);
X.order_of_key(-5);
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
Newton Rhapson - Find a root of a polynomial
start with random x
x_next = x - f(x)/f'(x)
repeat␣until␣f(x)␣is␣zero
~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~␣~
#define␣turnOffLastBit(S)␣((S)␣&␣(S␣-␣1))
#define␣turnOnLastZero(S)␣((S)␣|␣(S␣+␣1))
#define␣turnOffLastConsecutiveBits(S)␣((S)␣&␣(S␣+␣1))
#define␣turnOnLastConsecutiveZeroes(S)␣((S)␣|␣(S␣-␣1))
```

32 Formulas

32.1 Catalan Numbers

$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^{n} \frac{n+k}{k}, n \geq 0$

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

32.2 Law of Cosines

$c^2 = a^2 + b^2 - 2ab * cos(C)$

32.3 Law of Sines

$\frac{a}{sin(A)} = \frac{b}{sin(B)}$

32.4 Newton Raphson

$x_{n+1} = x_n - \frac{f(x_0)}{f'(x_0)}$

32.5 Arithmetic Series

$\sum_{k=1}^{n}(a_1 + (k-1)d) = na_1 + \frac{1}{2}nd(n-1)$

32.6 Geometric Series

$\sum_{k=1}^{n} r^k = \frac{r(1-r^n)}{1-r}$

$\sum_{k=1}^{\infty} r^k = \frac{r}{1-r}, |r| < 1$

32.7 Simpson's Rule

$\int_a^b f(x)dx \approx \frac{b-a}{6}(f(a) + 4f(\frac{a+b}{2}) + f(b))$

32.8 Stirling's Approximation

$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$

$ln(n!) = n * ln(n) - n + \frac{ln(2n)}{2}$

32.9 Sum of Powers

$\sum_{k=1}^{n} k^2 = \frac{1}{6}n(n+1)(2n+1)$

$\sum_{k=1}^{n} k^3 = (\sum_{k=1}^{n} k)^2 = (\frac{1}{2}n(n+1))^2$

32.10 Fermat's little Theorem

$a^p \equiv a \pmod{p}$ where p is prime

$a^{p-1} \equiv 1 \pmod{p}$ where p is prime and a is not divisible by p

32.11 Euler's Totient Function

$\phi(n) = n\Pi_{p|n}(1 - \frac{1}{p})$ where p is prime

32.12 Euler's Theorem

$a^{\phi(n)} \equiv 1 \pmod{n}$ where gcd(a, n) = 1

32.13 Convex Polygon Centroid

Given the polygon $P = A_1, A_2, ..., A_n$
let $a = (A_{k+1} - A_1),\ k = 1, 2, ..., n-1$ (the edge vectors)
let $C = A_1 + \frac{1}{3}(a_k + a_{k+1}),\ k = 1, 2, ..., n-2$ (the centroids of the triangles)
let $w = \frac{1}{2}(a_k \times a_{k+1}),\ k = 1, 2, ..., n-2$ (the areas of the triangles)

$$centroid = \frac{\sum_{k=1}^{n-2} w_k C_k}{\sum_{k=1}^{n-2} w_k} = A_1 + \frac{1}{3}\frac{\sum_{k=1}^{n-2}(a_k+a_{k+1})(a_k \times a_{k+1})}{\sum_{k=1}^{n-2}(a_k \times a_{k+1})}$$

32.14 Regular Polyhedron Volume

$volume = L^3$

32.15 Kirchoff Theorem

Let D be the degree matrix of G
Let A be the adjacency matrix of G
Let $Q = D - A$
Let Q' be the matrix resulting from deleting any row and any column from Q
The number of spanning trees in a graph is equal to the determinant of Q'

There are n^{n-2} spanning trees in a complete graph

There are $m^{n-1} * n^{m-1}$ spanning trees in complete a bipartite graph

32.16 Derangements

A derangement is a permutation of a set where all elements are in a different position than their original position

$der(n) = (n-1)*(der(n-1)+der(n-2)), der(0) = 1, der(1) = 0$

32.17 Planar Graph Faces

$F = E - V - 2$

33 Others

33.1 Dinic

```
// Running time: O(|V|^2 |E|)
// OUTPUT: maximum flow value;
// To obtain the actual flow values, look at all edges with
// capacity > 0 (zero capacity edges are residual edges).
struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}};
struct Dinic {
    int N;
    vector<vector<Edge> > G;
    vector<Edge *> dad;
    vector<int> Q;
    Dinic(int N) : N(N), G(N), dad(N), Q(N) {}
    void AddEdge(int from, int to, int cap) {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));}
    long long BlockingFlow(int s, int t) {
        fill(dad.begin(), dad.end(), (Edge *) NULL);
        dad[s] = &G[0][0] - 1;
        int head = 0, tail = 0;
        Q[tail++] = s;
        while (head < tail) {
            int x = Q[head++];
            for (int i = 0; i < G[x].size(); i++) {
                Edge &e = G[x][i];
                if (!dad[e.to] && e.cap - e.flow > 0) {
                    dad[e.to] = &G[x][i];
                    Q[tail++] = e.to;}}}
        if (!dad[t]) return 0;
        long long totflow = 0;
        for (int i = 0; i < G[t].size(); i++) {
            Edge *start = &G[G[t][i].to][G[t][i].index];
            int amt = INF;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
                if (!e) { amt = 0; break; }
                amt = min(amt, e->cap - e->flow);}
            if (amt == 0) continue;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
                e->flow += amt;
                G[e->to][e->index].flow -= amt;}
            totflow += amt;}
        return totflow;}
    long long GetMaxFlow(int s, int t) {
        long long totflow = 0;
        while (long long flow = BlockingFlow(s, t))
            totflow += flow;
        return totflow;}};
```

33.2 MinCostMaxFlow

```
// 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.
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);}};
```

33.3 PushRelabel

```
// 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).
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;}};
```

33.4 MinCostMatching

```
// 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.
// COST MUST BE SQUARE
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;}
```

33.5 MinCut

```
// 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)
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);}
```

33.6 GraphCutInference

```
// 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.
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;}
```

33.7 Geometry

```
#include <cassert>
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) {
    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)); }
// 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
```

```cpp
// (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" or "center of mass".
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;}
```

33.8 JavaGeometry

```java
// In this example, we read an input file containing three lines, each
// containing an even number of doubles, separated by commas.  The first two
// lines represent the coordinates of two polygons, given in counterclockwise
// (or clockwise) order, which we will call "A" and "B".  The last line
// contains a list of points, p[1], p[2], ...
//
// Our goal is to determine:
//   (1) whether B - A is a single closed shape (as opposed to multiple shapes)
//   (2) the area of B - A
//   (3) whether each p[i] is in the interior of B - A
//
// INPUT:
//   0 0 10 0 0 10
//   0 0 10 10 10 0
//   8 6
//   5 1
//
// OUTPUT:
//   The area is singular.
//   The area is 25.0
//   Point belongs to the area.
//   Point does not belong to the area.
import java.util.*;
import java.awt.geom.*;
import java.io.*;
public class JavaGeometry {
  // make an array of doubles from a string
  static double[] readPoints(String s) {
    String[] arr = s.trim().split("\\s++");
    double[] ret = new double[arr.length];
    for (int i = 0; i < arr.length; i++) ret[i] = Double.parseDouble(arr[i]);
    return ret;}
  // make an Area object from the coordinates of a polygon
  static Area makeArea(double[] pts) {
    Path2D.Double p = new Path2D.Double();
    p.moveTo(pts[0], pts[1]);
    for (int i = 2; i < pts.length; i += 2) p.lineTo(pts[i], pts[i+1]);
    p.closePath();
    return new Area(p);}
  // compute area of polygon
  static double computePolygonArea(ArrayList<Point2D.Double> points) {
    Point2D.Double[] pts = points.toArray(new Point2D.Double[points.size()]);
    double area = 0;
    for (int i = 0; i < pts.length; i++) {
      int j = (i+1) % pts.length;
      area += pts[i].x * pts[j].y - pts[j].x * pts[i].y;
    }
    return Math.abs(area)/2;}
  // compute the area of an Area object containing several disjoint polygons
  static double computeArea(Area area) {
    double totArea = 0;
    PathIterator iter = area.getPathIterator(null);
    ArrayList<Point2D.Double> points = new ArrayList<Point2D.Double>();
    while (!iter.isDone()) {
      double[] buffer = new double[6];
      switch (iter.currentSegment(buffer)) {
      case PathIterator.SEG_MOVETO:
      case PathIterator.SEG_LINETO:
        points.add(new Point2D.Double(buffer[0], buffer[1]));
        break;
      case PathIterator.SEG_CLOSE:
        totArea += computePolygonArea(points);
        points.clear();
        break;}
      iter.next();}
    return totArea;}
  // notice that the main() throws an Exception -- necessary to
  // avoid wrapping the Scanner object for file reading in a
  // try { ... } catch block.
  public static void main(String args[]) throws Exception {
    Scanner scanner = new Scanner(new File("input.txt"));
    // also,
    //   Scanner scanner = new Scanner (System.in);
    double[] pointsA = readPoints(scanner.nextLine());
    double[] pointsB = readPoints(scanner.nextLine());
    Area areaA = makeArea(pointsA);
    Area areaB = makeArea(pointsB);
    areaB.subtract(areaA);
    // also,
    //   areaB.exclusiveOr (areaA);
    //   areaB.add (areaA);
    //   areaB.intersect (areaA);
    // (1) determine whether B - A is a single closed shape (as
    //     opposed to multiple shapes)
    boolean isSingle = areaB.isSingular();
    // also,
    //   areaB.isEmpty();
    if (isSingle)
      System.out.println("The␣area␣is␣singular.");
    else
      System.out.println("The␣area␣is␣not␣singular.");
    // (2) compute the area of B - A
    System.out.println("The␣area␣is␣" + computeArea(areaB) + ".");
    // (3) determine whether each p[i] is in the interior of B - A
    while (scanner.hasNextDouble()) {
      double x = scanner.nextDouble();
      assert(scanner.hasNextDouble());
      double y = scanner.nextDouble();
      if (areaB.contains(x,y)) {
        System.out.println ("Point␣belongs␣to␣the␣area.");
      } else {
        System.out.println ("Point␣does␣not␣belong␣to␣the␣area.");}}
    // Finally, some useful things we didn't use in this example:
    //
    //   Ellipse2D.Double ellipse = new Ellipse2D.Double (double x, double y,
    //
    //                                                     double w, double h);
    //
    //     creates an ellipse inscribed in box with bottom-left corner (x,y)
    //     and upper-right corner (x+y,w+h)
    //
    //   Rectangle2D.Double rect = new Rectangle2D.Double (double x, double y,
    //
    //                                                     double w, double h);
    //
    //     creates a box with bottom-left corner (x,y) and upper-right
    //     corner (x+y,w+h)
    //
    // Each of these can be embedded in an Area object (e.g., new Area (rect)).
  }
}
```

33.9 Geom3D

```
public class Geom3D {
  // distance from point (x, y, z) to plane aX + bY + cZ + d = 0
  public static double ptPlaneDist(double x, double y, double z,
      double a, double b, double c, double d) {
    return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a + b*b + c*c);
  }
  // distance between parallel planes aX + bY + cZ + d1 = 0 and
  // aX + bY + cZ + d2 = 0
  public static double planePlaneDist(double a, double b, double c,
      double d1, double d2) {
    return Math.abs(d1 - d2) / Math.sqrt(a*a + b*b + c*c);
  }
  // distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)
  // (or ray, or segment; in the case of the ray, the endpoint is the
  // first point)
  public static final int LINE = 0;
  public static final int SEGMENT = 1;
  public static final int RAY = 2;
  public static double ptLineDistSq(double x1, double y1, double z1,
     double x2, double y2, double z2, double px, double py, double pz,
     int type) {
    double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);
    double x, y, z;
    if (pd2 == 0) {
      x = x1;
      y = y1;
      z = z1;
    } else {
      double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) / pd2;
      x = x1 + u * (x2 - x1);
      y = y1 + u * (y2 - y1);
      z = z1 + u * (z2 - z1);
      if (type != LINE && u < 0) {
        x = x1;y = y1;z = z1;}
      if (type == SEGMENT && u > 1.0) {
        x = x2;y = y2;z = z2;}}
    return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz);}
  public static double ptLineDist(double x1, double y1, double z1,
      double x2, double y2, double z2, double px, double py, double pz,
      int type) {
    return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz, type));}}
```

33.10 Delaunay

```
// 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>
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;
}
```

33.11 Simplex

```
// Two-phase simplex algorithm for solving linear programs 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 <limits>
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const DOUBLE EPS = 1e-9;
struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;
  LPSolver(const VVD &A, const VD &b, const VD &c) :
      m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
      for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
      for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }
      for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
      N[n] = -1; D[m+1][n] = 1;}
  void Pivot(int r, int s) {
      for (int i = 0; i < m+2; i++) if (i != r)
          for (int j = 0; j < n+2; j++) if (j != s)
      D[i][j] -= D[r][j] * D[i][s] / D[r][s];
      for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
      for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];
      D[r][s] = 1.0 / D[r][s];
      swap(B[r], N[s]);}
  bool Simplex(int phase) {
      int x = phase == 1 ? m+1 : m;
      while (true) {
          int s = -1;
          for (int j = 0; j <= n; j++) {
      if (phase == 2 && N[j] == -1) continue;
      if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
          }
          if (D[x][s] >= -EPS) return true;
          int r = -1;
          for (int i = 0; i < m; i++) {
      if (D[i][s] <= 0) continue;
      if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
            D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;
          }
          if (r == -1) return false;
          Pivot(r, s);}}
  DOUBLE Solve(VD &x) {
      int r = 0;
      for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
      if (D[r][n+1] <= -EPS) {
          Pivot(r, n);
          if (!Simplex(1) || D[m+1][n+1] < -EPS) return -numeric_limits<DOUBLE>::infinity();
          for (int i = 0; i < m; i++) if (B[i] == -1) {
      int s = -1;
      for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
      Pivot(i, s);}}
      if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
      x = VD(n);
      for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
      return D[m][n+1];}};
int main() {
    const int m = 4;
    const int n = 3;
    DOUBLE _A[m][n] = {
        { 6, -1, 0 },
        { -1, -5, 0 },
        { 1, 5, 1 },
        { -1, -5, -1 }
    };
    DOUBLE _b[m] = { 10, -4, 5, -5 };
    DOUBLE _c[n] = { 1, -1, 0 };
    VVD A(m);
    VD b(_b, _b + m);
    VD c(_c, _c + n);
    for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);
    LPSolver solver(A, b, c);
    VD x;
    DOUBLE value = solver.Solve(x);
    cerr << "VALUE:␣"<< value << endl;
    cerr << "SOLUTION:";
    for (size_t i = 0; i < x.size(); i++) cerr << "␣" << x[i];
    cerr << endl;
    return 0;
}
```

33.12 KDTree

```
//  - constructs from n points in O(n lg^2 n) time
//  - handles nearest-neighbor query in O(lg n) if points are well distributed
//  - worst case for nearest-neighbor may be linear in pathological case
// ----------------------------------------
#include <limits>
// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();
// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}};
bool operator==(const point &a, const point &b) {
    return a.x == b.x && a.y == b.y;}
// sorts points on x-coordinate
bool on_x(const point &a, const point &b) {
    return a.x < b.x;}
// sorts points on y-coordinate
bool on_y(const point &a, const point &b) {
    return a.y < b.y;}
// squared distance between points
ntype pdist2(const point &a, const point &b) {
    ntype dx = a.x-b.x, dy = a.y-b.y;
    return dx*dx + dy*dy;}
// bounding box for a set of points
struct bbox{
  ntype x0, x1, y0, y1;
  bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}
  // computes bounding box from a bunch of points
  void compute(const vector<point> &v) {
    for (int i = 0; i < v.size(); ++i) {
      x0 = min(x0, v[i].x);   x1 = max(x1, v[i].x);
      y0 = min(y0, v[i].y);   y1 = max(y1, v[i].y);}}
  // squared distance between a point and this bbox, 0 if inside
  ntype distance(const point &p) {
    if (p.x < x0) {
        if (p.y < y0)       return pdist2(point(x0, y0), p);
        else if (p.y > y1)  return pdist2(point(x0, y1), p);
        else                return pdist2(point(x0, p.y), p);}
    else if (p.x > x1) {
        if (p.y < y0)       return pdist2(point(x1, y0), p);
        else if (p.y > y1)  return pdist2(point(x1, y1), p);
        else                return pdist2(point(x1, p.y), p);}
    else {
        if (p.y < y0)       return pdist2(point(p.x, y0), p);
        else if (p.y > y1)  return pdist2(point(p.x, y1), p);
        else                return 0;}}};
// stores a single node of the kd-tree, either internal or leaf
struct kdnode {
  bool leaf;      // true if this is a leaf node (has one point)
  point pt;       // the single point of this is a leaf
  bbox bound;     // bounding box for set of points in children
  kdnode *first, *second; // two children of this kd-node
  kdnode() : leaf(false), first(0), second(0) {}
  ~kdnode() { if (first) delete first; if (second) delete second; }
  // intersect a point with this node (returns squared distance)
  ntype intersect(const point &p) {
     return bound.distance(p);}
  // recursively builds a kd-tree from a given cloud of points
  void construct(vector<point> &vp){
    // compute bounding box for points at this node
    bound.compute(vp);
    // if we're down to one point, then we're a leaf node
    if (vp.size() == 1) {
      leaf = true;
      pt = vp[0];}
    else {
      // split on x if the bbox is wider than high (not best heuristic...)
      if (bound.x1-bound.x0 >= bound.y1-bound.y0)
          sort(vp.begin(), vp.end(), on_x);
      // otherwise split on y-coordinate
      else
          sort(vp.begin(), vp.end(), on_y);
      // divide by taking half the array for each child
      // (not best performance if many duplicates in the middle)
      int half = vp.size()/2;
      vector<point> vl(vp.begin(), vp.begin()+half);
      vector<point> vr(vp.begin()+half, vp.end());
      first = new kdnode();   first->construct(vl);
      second = new kdnode();  second->construct(vr);}}};
// simple kd-tree class to hold the tree and handle queries
struct kdtree{
  kdnode *root;
  // constructs a kd-tree from a points (copied here, as it sorts them)
  kdtree(const vector<point> &vp) {
      vector<point> v(vp.begin(), vp.end());
      root = new kdnode();
      root->construct(v);}
  ~kdtree() { delete root; }
  // recursive search method returns squared distance to nearest point
  ntype search(kdnode *node, const point &p){
      if (node->leaf) {
          // commented special case tells a point not to find itself
//          if (p == node->pt) return sentry;
//          else
              return pdist2(p, node->pt);}
      ntype bfirst = node->first->intersect(p);
      ntype bsecond = node->second->intersect(p);
      // choose the side with the closest bounding box to search first
      // (note that the other side is also searched if needed)
      if (bfirst < bsecond) {
          ntype best = search(node->first, p);
          if (bsecond < best)
              best = min(best, search(node->second, p));
          return best;}
      else {
          ntype best = search(node->second, p);
          if (bfirst < best)
              best = min(best, search(node->first, p));
          return best;}}
  // squared distance to the nearest
  ntype nearest(const point &p) {
      return search(root, p);}};
// --------------------------------------------
// some basic test code here
int main() {
  // generate some random points for a kd-tree
  vector<point> vp;
  for (int i = 0; i < 100000; ++i) {
          vp.push_back(point(rand()%100000, rand()%100000));
  }
  kdtree tree(vp);
  // query some points
  for (int i = 0; i < 10; ++i) {
    point q(rand()%100000, rand()%100000);
    cout << "Closest␣squared␣distance␣to␣(" << q.x << ",␣" << q.y << ")"
    << "␣is␣" << tree.nearest(q) << endl;}
  return 0;}
// --------------------------------------------
```

33.13 LogLan

```
// Code which demonstrates the use of Java's regular expression libraries.
// This is a solution for
//
//   Loglan: a logical language
//   http://acm.uva.es/p/v1/134.html
//
// In this problem, we are given a regular language, whose rules can be
// inferred directly from the code.  For each sentence in the input, we must
// determine whether the sentence matches the regular expression or not.  The
// code consists of (1) building the regular expression (which is fairly
// complex) and (2) using the regex to match sentences.
import java.util.*;
import java.util.regex.*;
public class LogLan {
  public static String BuildRegex () {
    String space = "␣+";
    String A = "([aeiou])";
    String C = "([a-z&&[^aeiou]])";
    String MOD = "(g" + A + ")";
    String BA = "(b" + A + ")";
    String DA = "(d" + A + ")";
    String LA = "(l" + A + ")";
    String NAM = "([a-z]*" + C + ")";
    String PREDA = "(" + C + C + A + C + A + "|" + C + A + C + C + A + ")";
    String predstring = "(" + PREDA + "(" + space + PREDA + ")*)";
    String predname = "(" + LA + space + predstring + "|" + NAM + ")";
    String preds = "(" + predstring + "(" + space + A + space + predstring + ")*)";
    String predclaim = "(" + predname + space + BA + space + preds + "|" + DA + space +
                  preds + ")";
    String verbpred = "(" + MOD + space + predstring + ")";
    String statement = "(" + predname + space + verbpred + space + predname + "|" +
                  predname + space + verbpred + ")";
    String sentence = "(" + statement + "|" + predclaim + ")";
    return "^" + sentence + "$";}
  public static void main (String args[]) {
    String regex = BuildRegex();
    Pattern pattern = Pattern.compile (regex);
    Scanner s = new Scanner(System.in);
    while (true) {
      // In this problem, each sentence consists of multiple lines, where the last
      // line is terminated by a period.  The code below reads lines until
      // encountering a line whose final character is a '.'.  Note the use of
      //
      //    s.length() to get length of string
      //    s.charAt() to extract characters from a Java string
      //    s.trim() to remove whitespace from the beginning and end of Java string
      //
      // Other useful String manipulation methods include
      //
      //    s.compareTo(t) < 0 if s < t, lexicographically
      //    s.indexOf("apple") returns index of first occurrence of "apple" in s
      //    s.lastIndexOf("apple") returns index of last occurrence of "apple" in s
      //    s.replace(c,d) replaces occurrences of character c with d
      //    s.startsWith("apple) returns (s.indexOf("apple") == 0)
      //    s.toLowerCase() / s.toUpperCase() returns a new lower/uppercased string
      //
      //    Integer.parseInt(s) converts s to an integer (32-bit)
      //    Long.parseLong(s) converts s to a long (64-bit)
      //    Double.parseDouble(s) converts s to a double
      String sentence = "";
      while (true) {
        sentence = (sentence + "␣" + s.nextLine()).trim();
        if (sentence.equals("#")) return;
        if (sentence.charAt(sentence.length()-1) == '.') break;
      }
      // now, we remove the period, and match the regular expression
      String removed_period = sentence.substring(0, sentence.length()-1).trim();
      if (pattern.matcher (removed_period).find()) {
        System.out.println ("Good");
      } else {
        System.out.println ("Bad!");}}}}
```

33.14 IO

```
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;}
```

33.15 LatLong

```
/*
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;}
```

34 Edmonds Blossom

```
struct edge {
   int v, nx;
};
const int MAXN = 1000, MAXE = 2000;
edge graph[MAXE];
int last[MAXN], match[MAXN], px[MAXN], base[MAXN], N, edges;
bool used[MAXN], blossom[MAXN], lused[MAXN];
inline void add_edge(int u, int v) {
   graph[edges] = (edge) {v, last[u]};
   last[u] = edges++;
   graph[edges] = (edge) {u, last[v]};
   last[v] = edges++;
}
void mark_path(int v, int b, int children) {
   while (base[v] != b) {
      blossom[base[v]] = blossom[base[match[v]]] = true;
      px[v] = children;
      children = match[v];
      v = px[match[v]];
   }
}
int lca(int a, int b) {
   memset(lused, 0, N);
   while (1) {
      lused[a = base[a]] = true;
      if (match[a] == -1)
         break;
      a = px[match[a]];
   }
   while (1) {
      b = base[b];
      if (lused[b])
         return b;
      b = px[match[b]];
   }
}
int find_path(int root) {
   memset(used, 0, N);
   memset(px, -1, sizeof(int) * N);
   for (int i = 0; i < N; ++i)
      base[i] = i;
   used[root] = true;
   queue<int> q;
   q.push(root);
   int v, e, to, i;
   while (!q.empty()) {
      v = q.front(); q.pop();
      for (e = last[v]; e >= 0; e = graph[e].nx) {
         to = graph[e].v;
         if (base[v] == base[to] || match[v] == to)
            continue;
         if (to == root || (match[to] != -1 && px[match[to]] != -1)) {
            int curbase = lca(v, to);
            memset(blossom, 0, N);
            mark_path(v, curbase, to);
            mark_path(to, curbase, v);
            for (i = 0; i < N; ++i)
               if (blossom[base[i]]) {
                  base[i] = curbase;
                  if (!used[i]) {
                     used[i] = true;
                     q.push(i);
                  }
               }
         } else if (px[to] == -1) {
            px[to] = v;
            if (match[to] == -1)
               return to;
            to = match[to];
            used[to] = true;
            q.push(to);
         }
      }
   }
   return -1;
}
void build_pre_matching() {
   int u, e, v;
   for (u = 0; u < N; ++u)
      if (match[u] == -1)
         for (e = last[u]; e >= 0; e = graph[e].nx) {
            v = graph[e].v;
            if (match[v] == -1) {
               match[u] = v;
               match[v] = u;
               break;
            }
         }
}
void edmonds() {
   memset(match, 0xff, sizeof(int) * N);
   build_pre_matching();
   int i, v, pv, ppv;
   for (i = 0; i < N; ++i)
      if (match[i] == -1) {
         v = find_path(i);
         while (v != -1) {
            pv = px[v], ppv = match[pv];
            match[v] = pv, match[pv] = v;
            v = ppv;
         }
      }
}
```