The Reference

Sponsored by: Alichos, itan, el monstruo, el tío Ed, el itan x2, el garo y Jhon Cena.

Contenido

STL

C++

Vector <vector>:

```
vector<T> V     // O(1)
V.push_back(T) // O(1)
V.pop_back() // O(1)
V.erase( V.begin(), V.end() ) // O(n)
V.insert(i, T) // O(n)
V.size()       // O(1)
V.clear()      // O(n)
V[i]           // O(1)
```

Stack <stack>:

```
stack<T> S // O(1)
S.push(T)  // O(1)
S.pop()    // O(1)
S.size()   // O(1)
S.top()    // O(1)
```

Queue <queue>:

```
queue<T> Q // O(1)
Q.push(T)  // O(1)
Q.pop()    // O(1)
Q.size()   // O(1)
Q.front()  // O(1)
```

Double Ended Queue <deque>:

```
deque<T> D       // O(1)
D.push_back(T)  // O(1)
D.push_front(T) // O(1)
D.pop_back()    // O(1)
D.pop_front()   // O(1)
D.size()        // O(1)
D.back()        // O(1)
D.front()       // O(1)
D.clear()       // O(n)
```

Set <set>:

```
set<T> S     // O(1)
S.insert(T) // O(log n)
S.erase(T)  // O(log n)
S.count(T)  // O(log n)
set<T>::iterator low=S.lower_bound( val )//O(log n)
S.upper_bound( val ) // O(log n)
S.size() // O(1)
S.clear()   // O(n)
```

Map <map>:

```
map<K, V> M // O(1)
M[K] = V    // O(log n)
M.count(K)  // O(log n)
M.clear()   // O(n)
```

Priority Queue <queue>:

```
priority_queue<T> PQ // O(1)
PQ.push(T)           // O(log n)
PQ.pop()             // O(log n)
PQ.top()             // O(1)
PQ.empty()           // O(1)
```

Bit Set <bitset>:

```
bitset<n> B  // O(1)
B.count() //Count bits set
B.any() // Test if any bit is set
B.set() //Set bits to 1
B.set(int pos, bool val = true); // foo.set(2,0)
B.reset() // Resets bits to zero
B.to_ulong() // Convert to unsigned long integer.
B[i] = 0 | 1 // O(1)
```

Pair <utility>:

```
pair<T, T> P       // O(1)
P.first | P.second // O(1)
make_pair(T, T);   // O(1)
```

Override Comparison Operator:

```
struct T{ // Data
        bool operator <(T cmp) const{// Comparison
    }
};
```

C++11

Unordered Map <unordered_map>

```
unordered_map <K,V> M; //O(1)
M[K] = V // O(1)
M.count(K) //O(n)
M.clear() //O(n)
```

Unordered Set <unordered_set>

```
unordered_set <T> S // O(1)
S.insert( T ) // O(1)
S.erase( T ) // O(n)
S.clear() //O(n)
S.find(T) // O(1)
S.count(T) // O(n)
S.size() // O(1)
```

C++ STL Algorithms

Sort:

```
sort(array, array + n)              // O(n log n)
sort(vector.begin(), vector.end()) // O(n log n)
```

Reverse:

```
reverse(array, array + n)              // O(n)
reverse(vector.begin(), vector.end()) // O(n)
```

Fill:

```
fill(array, array + n, T);              // O(n)
fill(vector.begin(), vector.end(), T); // O(n)
```

Lower / Upper Bound:

```
lower_bound(array, array + n, T) // O(log n)
lower_bound(vector.begin(), vector.end(), T)

upper_bound(array, array + n, T) // O(log n)
upper_bound(vector.begin(), vector.end(), T)

set.lower_bound(T) // O(log n)
set.upper_bound(T) // O(log n)
```

Next / Previous Permutation:

```
next_permutation(array, array + n) // O(n)
prev_permutation(array, array + n) // O(n)
```

```
Minimum / Maximum:
min(T, T) // O(1)
max(T, T) // O(1)
```

Geometría

```
// Definiciones iniciales.
typedef long long Long;

const double ERROR = 1e-9;
const double M_2PI = 2 * M_PI;

// Tolerancia en flotantes.
bool Igual(double a, double b) {
    return fabs(a - b) < ERROR;
}
```

Estructura Punto

```
// Punto en 2D.
struct Punto {

    double x, y;
    Punto() : x(), y() {}
    Punto(double X, double Y) : x(X), y(Y) {}

    // Izquierda a derecha, abajo a arriba.
    bool operator<(const Punto& cmp) const {
        if (!Igual(x, cmp.x)) return x < cmp.x;
        return Igual(y, cmp.y)? false: y < cmp.y;
    }

    bool operator==(const Punto& cmp) const {
        return Igual(x, cmp.x) && Igual(y, cmp.y);
    }
};
```

double Distancia(const Punto& p, const Punto& q)

```
// Distancia entre dos puntos p y q.
double Distancia(const Punto& p, const Punto& q) {
    return hypot(p.x - q.x, p.y - q.y);
}
```

double Magnitud(const Punto& v)

```
// Magnitud de un vector v.
double Magnitud(const Punto& v) {
    return hypot(v.x, v.y);
}
```

double Dot(const Punto& v, const Punto& w)

```
// Producto punto entre vectores v y w.
double Dot(const Punto& v, const Punto& w) {
    return v.x * w.x + v.y * w.y;
}
```

double Cruz(const Punto& v, const Punto& w)

```
// Producto cruz entre vectores v y w.
double Cruz(const Punto& v, const Punto& w) {
    return v.x * w.y - v.y * w.x;
}
```

double GradARad(double grd)

```
// Conversion de grados a radianes.
double GradARad(double grd) {
    return (grd * M_PI) / 180;
}
```

double RadAGrad(double rad)

```
// Conversion de radianes a grados.
double RadAGrad(double rad) {
    return (rad * 180) / M_PI;
}
```

Punto Rotar(const Punto& p, double rad)

```
// Rotar un punto respecto al origen.
// La rotación se hace en orden CCW, para
// rotar en CW llamar Rotar(p, M_2PI - rad).
Punto Rotar(const Punto& p, double rad) {
    return Punto(p.x*cos(rad) - p.y*sin(rad),
                 p.x*sin(rad) + p.y*cos(rad));
}
```

Punto Trasladar(const Punto& o, const Punto& p)

```
// Trasladar p tomando como origen al punto o .
Punto Trasladar(const Punto& o, const Punto& p) {
    return Punto(p.x - o.x, p.y - o.y);
}
```

Punto Escalar(const Punto& v, double s)

```
// Escalar un vector v por un factor s.
Punto Escalar(const Punto& v, double s) {
    return Punto(v.x * s, v.y * s);
}
```

Punto Opuesto(const Punto& v)

```
// Obtener vector opuesto.
Punto Opuesto(const Punto& v) {
    return Punto(-v.x, -v.y);
}
```

double Angulo(const Punto& v, const Punto& w)

```
// Angulo entre vectores v y w.
double Angulo(const Punto& v, const Punto& w) {
    return acos(Dot(v, w) / (Magnitud(v) *
                Magnitud(w)));
}
```

int ManoDerecha(const Punto& o, const Punto& p, const Punto& q)

```
// Test de mano derecha: CCW = 1, CW=-1,Colineal=0.
int ManoDerecha(const Punto& o, const Punto& p,
                const Punto& q) {
    double ccw = Cruz(Trasladar(o, p),
                Trasladar(o, q));
    return Igual(ccw, 0)? 0: (ccw < 0)? -1: 1;
}
```

pair<Punto, Punto> ParMasCercano(vector<Punto> P)

```
// Par de puntos mas cercanos en un conjunto de
puntos P.
pair<Punto, Punto> ParMasCercano(vector<Punto> P) {
    // Si ya esta ordenado, no usar sort.
    sort(P.begin(), P.end());
    set<Punto> rect;
    pair<Punto, Punto> par;
    int prev = 0; double delta = 1e9;
    for (int i = 0; i < P.size(); ++i) {
        while (P[i].x - P[prev].x > delta)
            rect.erase(Punto(P[prev].y,
                             P[prev++].x));
    set<Punto>::iterator it = rect.lower_bound(
                Punto( P[i].y - delta, P[0].x));
    for (; it != rect.end()
        && it->x <= P[i].y + delta; ++it) {
        double dist = hypot(P[i].x - it->y,
                            P[i].y - it->x);
        if (dist < delta)
            delta = dist, par = make_pair(
                Punto(it->y, it->x), P[i]);
        }
        rect.insert(Punto(P[i].y, P[i].x));
    }
    return par;
    // Alternativamente puede devolver delta.
}
```

Estructura Linea

```
// Linea en 2D.
// Si los puntos no aseguran coordenadas
// enteras usar version double. ¡CUIDADO!
// Verifiquen los tags <comment> <uncomment>
```

```
struct Linea {
    Punto p, q;
    Long a, b, c; // <comment/>
    //double a, b, c; // <uncomment/>
    Linea() : p(), q(), a(), b(), c() {}
    Linea(Long A, Long B, Long C)
        : p(), q(), a(A), b(B), c(C) {
        if (Igual(a, 0)) {
            c /= -b; b = -1;
            p = Punto(0, c);
            q = Punto(1, c);
        } else if (Igual(b, 0)) {
            c /= -a; a = -1;
            p = Punto(c, 0);
            q = Punto(c, 1);
        } else {
            p = Punto(-c/a, 0);
            q = Punto(-(b+c)/a, 1);
        }
        if (q < p) swap(p, q);
    }
    Linea(const Punto& P, const Punto& Q)
        : p(P), q(Q), a(), b(), c() {
        // Asegura p como punto menor.
        if (q < p) swap(p, q);
        a = q.y - p.y;
        b = p.x - q.x;
        if (!a) c = p.y, b = -1;
        else if (!b) c = p.x, a = -1;
        else {
            // <comment>
            c = abs(__gcd(a, b));
            a /= c, b /= c;
            // </comment>
            c = -a*p.x - b*p.y;
        }
    }
    // ¡PELIGRO! Ordena por ecuacion de recta.
    bool operator<(const Linea& cmp) const {
        if (!Igual(a, cmp.a)) return a < cmp.a;
        if (!Igual(b, cmp.b)) return b < cmp.b;
        return Igual(c, cmp.c)? false: c < cmp.c;
    }
};
```

bool PuntoEnRecta(const Punto& p, const Linea& r)

```
// Saber si un punto p esta en la recta r.
bool PuntoEnRecta(const Punto& p, const Linea& r) {
    return !ManoDerecha(r.p, r.q, p);
}
```

bool PuntoEnSegmento(constPunto p,const Linea&s)

```
// Saber si un punto p esta en el segmento s.
bool PuntoEnSegmento(const Punto& p,
                     const Linea& s){
    return PuntoEnRecta(p, s) &&
           !(p < s.p || s.q < p);
}
```

bool LineasParalelas(const Linea&l, const Linea&m)

```
// Saber si dos lineas l y m son paralelas.
bool LineasParalelas(const Linea& l,
                     const Linea& m) {
    return l.a == m.a && l.b == m.b; // <comment/>
    // <uncomment>
    //if (Igual(l.b, 0) || Igual(m.b, 0))
    //    return Igual(l.a, m.a) && Igual(l.b,m.b);
    //return Igual(l.a/l.b, m.a/m.b);
    // </uncomment>
}
```

bool LineasIguales(const Linea& l, const Linea& m)

```
// Saber si dos lineas l y m son iguales.
bool LineasIguales(const Linea& l, const Linea& m){
    return LineasParalelas(l, m) && Igual(l.c,m.c);
}
```

bool LineasPerpendiculares(const Linea& l ,const Linea& m)

```
// Saber si dos lineas l y m son perpendiculares.
bool LineasPerpendiculares(const Linea& l,
                           const Linea& m) {
    return (l.a == m.b && l.b == -m.a) ||
           (m.a==l.b && m.b == -l.a); //<comment/>
    // <uncomment>
    //if (Igual(l.b, 0) || Igual(l.a, 0))
    //    return Igual(l.a, m.b) && Igual(l.b,m.a);
    //return Igual(-l.a/l.b, m.b/m.a);
    // </uncomment>
}
```

Linea ParalelaEnPunto(const Linea& l, const Punto& p)

```
// Obtener una linea paralela a l que pase por p.
Linea ParalelaEnPunto(const Linea& l,
                      const Punto& p) {
    return Linea(p, Punto(p.x - l.b, p.y + l.a));
}
```

Linea PerpendicularEnPunto(const Linea& l, const Punto& p)

```
//Obtener una linea perpendicular a l que pase por
p.
Linea PerpendicularEnPunto(const Linea& l,
                           const Punto& p) {
    return Linea(p, Punto(p.x + l.a, p.y + l.b));
}
```

int InterseccionRectas(const Linea&r,const Linea&s)

```
// Saber si dos rectas r y s se intersectan.
// No intersectan = 0, Interseccion en un punto =1,
// Interseccion paralela en infinitos puntos = -1.
int InterseccionRectas(const Linea& r,
                       const Linea& s) {
    if (LineasIguales(r, s)) return -1;
    return LineasParalelas(r, s)? 0: 1;

}
```

int IntersecRectaSegmen(const Linea& r, const Linea& s)

```
// Saber si una recta r y un segmento s se
intersectan.
// No intersectan = 0, Interseccion en un punto= 1,
// Interseccion paralela en infinitos puntos = -1.
int IntersecRectaSegmen(const Linea& r,
                        const Linea& s) {
    if (LineasIguales(r, s)) return -1;
    if (LineasParalelas(r, s)) return 0;
    int t1 = ManoDerecha(r.p, r.q, s.p);
    int t2 = ManoDerecha(r.p, r.q, s.q);
    return (t1 != t2)? 1: 0;
}
```

int InterseccionSegmentos(const Linea& s, const Linea& t)

```
// Saber si dos segmentos s y t se intersectan.
// No intersectan = 0, Interseccion en un punto =1,
// Interseccion paralela en infinitos puntos = -1.
int InterseccionSegmentos(const Linea& s,
                          const Linea& t) {
    int t1 = ManoDerecha(s.p, s.q, t.p);
    int t2 = ManoDerecha(s.p, s.q, t.q);
    if (t1 == t2) return t1? 0:
                 (PuntoEnSegmento(s.p, t) ||
                  PuntoEnSegmento(s.q, t) ||
                  PuntoEnSegmento(t.p, s) ||
                  PuntoEnSegmento(t.q, s))? -1: 0;
    return (ManoDerecha(t.p, t.q, s.p) !=
            ManoDerecha(t.p, t.q, s.q))? 1: 0;
}
```

Punto PuntoInterseccion(const Linea& l, const Linea& m)

```
//Obtener punto de interseccion entre lineas l y m.
Punto PuntoInterseccion(const Linea& l,
                        const Linea& m) {
    assert(!LineasParalelas(l, m));
    //Si son paralelas KABOOM!
    if (Igual(l.a, 0)) return
        Punto((double)(l.c*m.b + m.c) / -m.a, l.c);
    double y = (double)(m.a*l.c - l.a*m.c) /
                       (m.b*l.a - m.a*l.b);
    return Punto((double)(l.c + l.b * y)/ -l.a, y);
}
```

Punto ProyeccionEnRecta(const Punto& v, const Linea& r)

```
// Obtener proyeccion del vector v en la recta r.
Punto ProyeccionEnRecta(const Punto& v,
                        const Linea& r) {
    Punto a = Trasladar(r.p, v),
          b = Trasladar(r.p, r.q);
    return Trasladar(Opuesto(r.p),
         Escalar(b,Dot(a, b)/pow(Magnitud(b), 2)));
}
```

double DistanciaPuntoRecta(const Punto& p, const Linea& r)

```
// Distancia entre un punto p y una recta r.
double DistanciaPuntoRecta(const Punto& p,
                           const Linea& r) {
    return Distancia(ProyeccionEnRecta(p, r), p);
}
```

double DistanciaPuntoSegmento(const Punto& p, const Linea& s)

```
// Distancia entre un punto p y un segmento s.
double DistanciaPuntoSegmento(const Punto& p,
                              const Linea& s) {
    Punto proy = ProyeccionEnRecta(p, s);
    if (proy < s.p) return Distancia(s.p, p);
    if (s.q < proy) return Distancia(s.q, p);
    return Distancia(proy, p);
}
```

double DistanciaRectaRecta(const Linea& l, const Linea& m)

```
// Distancia entre dos lineas l y m.
double DistanciaRectaRecta(const Linea& l,
                           const Linea& m) {
    return LineasParalelas(l, m)?
           DistanciaPuntoRecta(l.p, m): 0;
}
```

double DistanciaSegmenSegmen(const Linea& s, const Linea& r)

```
// Distancia entre dos segmentos s y r.
double DistanciaSegmenSegmen(const Linea& s,
                             const Linea& r) {
    if (InterseccionSegmentos(s, r)) return 0;
    return min(min(DistanciaPuntoSegmento(s.p, r),
               DistanciaPuntoSegmento(s.q, r)),
               min(DistanciaPuntoSegmento(r.p, s),
               DistanciaPuntoSegmento(r.q, s)));
}
```

Definición Polígono

```
// Un poligono es una serie de
// vertices conectados por aristas.
// P = p1 -> p2 -> p3 -> ... -> pn -> p1.

typedef vector<Punto> Poligono;
```

bool PuntoEnPerimetro(const Punto& p, const Poligono& P)

```
// Saber si un punto esta en el perimetro de un
poligono.
bool PuntoEnPerimetro(const Punto& p,
                      const Poligono& P) {
    for (int i = 1; i < P.size(); ++i) {
        Punto l = min(P[i - 1], P[i]);
        Punto r = max(P[i - 1], P[i]);
        if (ManoDerecha(l, r, p) == 0 &&
           !(p < l || r < p)) return true;
    }
    return false;
}
```

int PuntoEnConvexo(const Punto& p, const Poligono& P)

```
// Prueba de punto en poligono convexo.
// En el perimetro = -1, Fuera = 0, Dentro = 1.
int PuntoEnConvexo(const Punto& p,
                   const Poligono& P) {
    if (PuntoEnPerimetro(p, P)) return -1;
    int dir = ManoDerecha(P[0], P[1], p);
    for (int i = 2; i < P.size(); ++i)
        if (ManoDerecha(P[i - 1], P[i], p) != dir)
            return 0; // Fuera.
    return 1; // Dentro.
}
```

int RayCasting(const Punto& p, const Poligono& P)

```
// Punto en poligono concavo por ray casting.
// En el perimetro = -1, Fuera = 0, Dentro = 1.
int RayCasting(const Punto& p, const Poligono& P) {
    if (PuntoEnPerimetro(p, P)) return -1;
    Punto o = *min_element(P.begin(), P.end());
    Linea rayo(p, Punto(o.x - M_PI, o.y - M_E));
    int cruces = 0;
    for (int i = 1; i < P.size(); ++i)
        if (InterseccionSegmentos(rayo,
            Linea(P[i - 1], P[i]))) ++cruces;
    return cruces & 1;
}
```

int AngleSummation(const Punto& p, const Poligono& P)

```
// Punto en poligono concavo por angle summation.
// En el perimetro = -1, Fuera = 0, Dentro = 1.
int AngleSummation(const Punto& p,
                   const Poligono& P) {
    if (PuntoEnPerimetro(p, P)) return -1;
    double angulo = 0;
    for (int i = 1; i < P.size(); ++i)
        angulo += ManoDerecha(p, P[i - 1], P[i]) *
                  Angulo(Trasladar(p, P[i - 1]),
                  Trasladar(p, P[i]));
    return (fabs(angulo) > M_PI)? 1: 0;
}
```

double Area(const Poligono& P)

```
// Area de un poligono.
double Area(const Poligono& P) {
    double area = 0;
    for (int i = 1; i < P.size(); ++i)
        area += Cruz(P[i - 1], P[i]);
    return fabs(area) / 2.0;
}
```

double Perimetro(const Poligono& P)

```
// Perimetro de un poligono.
double Perimetro(const Poligono& P) {
    double perimetro = 0;
    for (int i = 1; i < P.size(); ++i)
        perimetro += Distancia(P[i - 1], P[i]);
    return perimetro;
}
```

Poligono CercoConvexo(vector<Punto> P)

```
// Cerco convexo de un conjunto de puntos.
Poligono CercoConvexo(vector<Punto> P){
    // Si ya esta ordenado, no usar sort.
    sort(P.begin(), P.end());
    Poligono arriba, abajo;
    for (int i = 0; i < P.size(); ++i) {
        while (arriba.size() > 1) {
            int p = arriba.size() - 1;
            // Permitir colineales: <=
            if (ManoDerecha(arriba[p - 1],
                arriba[p], P[i]) < 0) break;
            arriba.pop_back();
        }
        arriba.push_back(P[i]);
    }
    arriba.pop_back();
    for (int i = P.size() - 1; i >= 0; --i) {
        while (abajo.size() > 1) {
            int p = abajo.size() - 1;
            // Permitir colineales: <=
            if (ManoDerecha(abajo[p - 1],
                abajo[p], P[i]) < 0) break;
            abajo.pop_back();
        }
        abajo.push_back(P[i]);
    }
    arriba.insert(arriba.end(),
                  abajo.begin(), abajo.end());
    return arriba; // Convex hull.
}
```

Punto Centroide(const Poligono& P)

```
// Centroide de un poligono.
Punto Centroide(const Poligono& P) {
    double x = 0, y = 0, k = 0;
    for (int i = 1; i < P.size(); ++i) {
        double cruz = Cruz(P[i - 1], P[i]);
        x += cruz * (P[i - 1].x + P[i].x);
        y += cruz * (P[i - 1].y + P[i].y);
        k += cruz * 3;
    }
    return Punto(x/k, y/k);
}
```

bool RectaCortaPoligono(const Linea& r, const Poligono& P)

```
// Saber si una recta corta un poligono.
bool RectaCortaPoligono( const Linea& r,
                         const Poligono& P) {
    for (int i = 0, prim = 0; i < P.size(); ++i) {
        int lado = ManoDerecha(r.p, r.q, P[i]);
        if (!lado) continue;
        if (!prim) prim = lado;
        else if (lado != prim) return true;
    }
    return false;
}
```

vector<Poligono> CortarPoligono(const Poligono& P, const Linea& r)

```
// Obtiene los poligonos resultantes de
// cortar un poligono convexo con una recta.
vector<Poligono> CortarPoligono(const Poligono& P,
                                const Linea& r) {
    if (!RectaCortaPoligono(r, P))
        return vector<Poligono>(1, P);
    int ind = 0;
    vector<Poligono> Ps(2);
    for (int i = 1; i < P.size(); ++i) {
        Linea s(P[i - 1], P[i]);
        if (IntersecRectaSegmen(r, s)) {
            Punto p = PuntoInterseccion(r, s);
            if (P[i - 1] == p) continue;
            Ps[ind].push_back(P[i - 1]);
            Ps[1 - ind].push_back(p);
            Ps[ind].push_back(p);
            ind = 1 - ind;
        }
        else Ps[ind].push_back(P[i - 1]);
    }
    Ps[0].push_back(Ps[0][0]);
    Ps[1].push_back(Ps[1][0]);
    return Ps;
}
```

Estructura Círculo

```
// Circulo en 2D.
struct Circulo {
    Punto c; double r;
    Circulo() : c(), r() {}
    Circulo(const Punto& C, double R) : c(C),r(R){}
    bool operator<(const Circulo& cmp) const {
        if (!(c == cmp.c)) return c < cmp.c;
        return Igual(r, cmp.r)? false: r < cmp.r;
    }
};
```

double Circuferencia(const Circulo& c)

```
// Circunferencia de un circulo.
double Circuferencia(const Circulo& c) {
    return M_2PI * c.r;
}
```

double Area(const Circulo& c)

```
// Area de un circulo.
double Area(const Circulo& c) {
    return M_PI * c.r * c.r;
}
```

int PuntoEnCirculo(const Punto& p,const Circulo& c)

```
// Saber si un punto esta dentro de un circulo.
// En circunferencia = -1, Fuera = 0, Dentro = 1.
int PuntoEnCirculo(const Punto& p,
                   const Circulo& c) {
    double dist = Distancia(p, c.c);
    if (Igual(dist, c.r)) return -1;
    return (dist < c.r)? 1: 0;
}
```

double DistanciaPuntoCirculo(const Punto& p, const Circulo& c)

```
// Distancia de un punto p a un circulo c
double DistanciaPuntoCirculo(const Punto& p,
                             const Circulo& c) {
    double dist = Distancia(p, c.c) - c.r;
    return (dist < 0)? 0: dist;
}
```

Punto ProyPuntoCircunferencia(const Punto& p, const Circulo& c)

```
// Proyecta un punto fuera de un circulo en su
circunferencia.
Punto ProyPuntoCircunferencia(const Punto& p,
                              const Circulo& c) {
    Punto v = Trasladar(p, c.c);
    double prop = DistanciaPuntoCirculo(p, c) /
                  Magnitud(v);
    return Trasladar(Opuesto(p), Escalar(v, prop));
}
```

Linea ProyTangentes(const Punto& p, const Circulo& c)

```
// Obtiene dos puntos que, desde el punto p, forman
//lineastangentesa la circunferencia del circulo c.
Linea ProyTangentes(const Punto& p,
                    const Circulo& c) {
    double a = acos(c.r / Distancia(p, c.c));
    Punto p_ = Trasladar(c.c,
               ProyPuntoCircunferencia(p, c));
    return Linea(Trasladar(Opuesto(c.c),
                 Rotar(p_, M_2PI - a)),
                 Trasladar(Opuesto(c.c),
                 Rotar(p_, a)));
}
```

int IntersecCirculoRecta(const Circulo& c, const Linea& r)

```
// Saber si se intersecta un circuloC y una rectaR.
// Tangente = -1, No se intersectan = 0, Cuerda= 1.
int IntersecCirculoRecta(const Circulo& c,
                         const Linea& r) {
    double dist = DistanciaPuntoRecta(c.c, r);
    if (Igual(dist, c.r)) return -1;
    return (dist < c.r)? 1: 0;
}
```

Punto Chicharronera(double a, double b, double c)

```
// Soluciones a ecuaciones cuadraticas.
Punto Chicharronera(double a, double b, double c) {
    double sq = sqrt(b*b - 4*a*c);
    return Punto((-b + sq) / (2*a),
                 (-b - sq) / (2*a));
}
```

Linea CuerdaInterseccion(const Circulo& c, const Linea& r)

```
//Cuerda de interseccion entre 1 circulo y 1 recta.
Linea CuerdaInterseccion(const Circulo& c,
                         const Linea& r) {
    assert(IntersecCirculoRecta(c, r)); // KABOOM!
    Punto p, q;
    if (!Igual(r.b, 0)) {
        Linea R = Linea(Trasladar(c.c, r.p),
                        Trasladar(c.c, r.q));
        p = Chicharronera(R.a*R.a + R.b*R.b,
                          2*R.a*R.c,
                        R.c*R.c - R.b*R.b*c.r*c.r);
        q = Punto(p.y, (R.c + R.a*p.y) / -R.b);
        p.y = (R.c + R.a*p.x) / -R.b;
        p = Trasladar(Opuesto(c.c), p);
        q = Trasladar(Opuesto(c.c), q);
    }
    else {
        double sq = sqrt(c.r*c.r –
                    pow(r.p.x - c.c.x, 2));
        p = Punto(r.p.x, c.c.y + sq);
        q = Punto(r.p.x, c.c.y - sq);
    }
    return Linea(p, q);
}
```

bool CirculoEnCirculo(const Circulo& c, const Circulo& d)

```
//Saber si 1 circulo c esta dentro de un circulo d.
bool CirculoEnCirculo(const Circulo& c,
                      const Circulo& d) {
    return Distancia(c.c, d.c) < d.r - c.r;
}
```

int IntersecCirculoCirculo(const Circulo& c, const Circulo& d)

```
//Saber si circulo c intersecta con el circulo d.
// Uno dentro del otro = -1, Disjuntos = 0,
Intersectan = 1.
int IntersecCirculoCirculo(const Circulo& c,
                           const Circulo& d) {
    double dist = Distancia(c.c, d.c);
    if (dist < fabs(c.r - d.r)) return -1;
    return (dist > c.r + d.r)? 0: 1;
}
```

int TangenteExtCirculoCirculo

```
// Obtiene tangentes exteriores (las que NO se
cruzan) entre dos círculos.
Int TangenteExtCirculoCirculo(const Circulo& a,
const Circulo& b, Linea &s, Linea &t) {
// Circulos identicos. Tangentes infinitas (o
ninguna a discrecion)
    if (Igual(a.r, b.r) && a.c == b.c) return 0;
// Uno es círculo interior del otro. Comparten una
tangente.
// EL CALCULO PUEDE COPIARSE A TangenteInt SI SE
REQUIERE.
    Punto u;
    bool unico = false;
    if(b.r<a.r && Igual(Distancia(a.c, b.c)
                    + b.r, a.r)){
        u = Trasladar(Opuesto(a.c),
            Escalar(Trasladar(a.c, b.c),
            a.r/Distancia(a.c, b.c)));
        unico = true;
    }
    if (a.r < b.r && Igual(Distancia(a.c, b.c)
                     + a.r, b.r)) {
        u = Trasladar(Opuesto(b.c),
            Escalar(Trasladar(b.c, a.c),
            b.r / Distancia(a.c, b.c)));
        unico = true;
    }
    if (unico) {
        s = t = PerpendicularEnPunto(
                Linea(a.c, b.c), u);
```

```
// Recta de tangencia; un punto es referencia.
//s = t = Linea(u, u); // Punto de tangencia.
        return 1;
    }
// Circulo en circulo. No hay tangentes.
    if (CirculoEnCirculo(a, b) ||
        CirculoEnCirculo(b, a)) {
        return 0;
    }
// Calcular las 2 rectas tangentes.
    Linea proy;
    Punto v;
    if (Igual(a.r, b.r)) {
        proy = Linea(a.c, a.c);
        Linea perp = PerpendicularEnPunto(
                    Linea(a.c, b.c), a.c);
        u = Escalar(Trasladar(perp.q, perp.p),
                b.r / Distancia(perp.p, perp.q));
        v = Opuesto(u);
    } else {
        Circulo c(a.c, abs(a.r - b.r));
        proy = ProyTangentes(b.c, c);
        u = Escalar(Trasladar(a.c, proy.p),
            b.r / (a.r - b.r));
        v = Escalar(Trasladar(a.c, proy.q),
            b.r / (a.r - b.r));
    }
    s = Linea(Trasladar(Opuesto(proy.p), u),
              Trasladar(Opuesto(b.c), u));
    t = Linea(Trasladar(Opuesto(proy.q), v),
              Trasladar(Opuesto(b.c), v));
    return 2;
}
```

int TangenteIntCirculoCirculo

```
// Obtiene tangentes interiores (las que SI se
cruzan) entre dos círculos.
int TangenteIntCirculoCirculo(const Circulo& a,
            const Circulo& b, Linea &s, Linea &t) {
// Circulos identicos. Tangentes infinitas (o
ninguna a discrecion)
    if (Igual(a.r, b.r) && a.c == b.c)
        return 0
// Uno es círculo interior del otro. Comparten una
tangente.
// CALCULO HECHO EN TangenteInt. Copiar de este si
se requiere.
Punto u;
// Circulos tangentes. Obtener recta tangente
unica.
    if (Igual(Distancia(a.c, b.c), a.r + b.r)) {
        u = Trasladar(Opuesto(b.c),
            Escalar(Trasladar(b.c, a.c),
            b.r / Distancia(a.c, b.c)));
        s = t = PerpendicularEnPunto(
                    Linea(a.c, b.c), u);
        // Recta de tangencia.
        //s = t = Linea(u, u);
        // Punto de tangencia.
        return 1;
    }
    // Circulos se traslapan. no hay tangentes.
    if (!(a.r + b.r < Distancia(a.c, b.c)))
        return 0;
    // Obtener 2 rectas tangentes.
    Linea proy;
    Punto v;
    Circulo c(a.c, a.r + b.r);
    proy = ProyTangentes(b.c, c);
    u = Escalar(Trasladar(a.c, proy.p),
                b.r / (a.r + b.r));
    v = Escalar(Trasladar(a.c, proy.q),
                b.r / (a.r + b.r));
    s =Linea(Trasladar(Opuesto(proy.p),Opuesto(u)),
             Trasladar(Opuesto(b.c), Opuesto(u)));
    t =Linea(Trasladar(Opuesto(proy.q),Opuesto(v)),
             Trasladar(Opuesto(b.c), Opuesto(v)));
    return 2;
}
```

vector< Punto > PuntosInterseccionCirculos(const Circulo& c, const Circulo& d)

```
vector< Punto > PuntosInterseccionCirculos( const
Circulo& c, const Circulo& d ){
    int mano;
    vector<Punto> ret;
    double angulo = 0.0, dist, X, Y;
    Circulo C = Circulo( Punto( 0, 0 ), c.r );
    Circulo D = Circulo(Trasladar( c.c, d.c ),d.r);
    mano = ManoDerecha(Punto(0,0),Punto(1,0), D.c);
    if( mano == 1 )
        angulo = M_2PI - Angulo( Punto(1,0),D.c );
    else if( mano == -1 )
        angulo = Angulo( Punto( 1, 0 ), D.c );
    D.c = Rotar( D.c, angulo );
    dist = Distancia( D.c, C.c );
    if( Igual( dist, C.r + D.r ) ){
        ret.push_back( Punto( C.r, 0 ) );
        ret[0] = Rotar( ret[0], M_2PI - angulo );
        ret[0] = Trasladar(
                 Punto( -c.c.x, -c.c.y), ret[0] );
    }
    else if( dist < (C.r + D.r) &&
             dist > fabs(C.r - D.r)) {
        X = (dist*dist - D.r*D.r + C.r*C.r) /
            (2*dist);
        Y = sqrt( C.r*C.r - X*X );
        ret.push_back( Punto( X, Y ) );
        ret.push_back( Punto( X, -Y ) );
        for( int i = 0; i < 2; i++ ){
            ret[i] = Rotar( ret[i], M_2PI-angulo );
            ret[i] = Trasladar( Punto( -c.c.x,
                                -c.c.y ), ret[i] );
        }
    }
    return ret;
}
```

Grafos

Definiciones Iniciales

```
typedef int Costo;
typedef pair<int, int> Arista;
const Costo INF = 1 << 30;
```

Estructura Grafo

```
// Grafos no ponderados.
// Nodos indexados de 0 a n - 1.
// Grafo(n,  true) -> Bidireccional.
// Grafo(n, false) -> Dirigido.
struct Grafo {
    int n; bool bi;
    vector<vector<int>> ady;
    Grafo(int N, bool B = true): n(N), bi(B),
         ady(N) {}
    void AgregarArista(int u, int v) {
        if (bi) ady[v].push_back(u);
        ady[u].push_back(v);
    }
```

Detectar Ciclos

```
// Detecta ciclos en un grafo o multigrafo.
// Llamar a DetectarCiclos() devuelve un
// vector de vectores; cada vector interno es
// una lista que representa un ciclo del grafo.
// NOTA: Solo detecta un ciclo por componente.
vector<int> ciclo;
vector<char> color;
void DetectarCiclo(int u, int p) {
    int retorno = bi? 0: 1;
    color[u] = ciclo.empty()? 'G': 'N';
    for (int v : ady[u]) {
        if (v == p && !retorno++) continue;
        if (ciclo.empty() && color[v] == 'G') {
            color[v] = 'A', ciclo.push_back(v);
            if (u != v) color[u] = 'R',
                        ciclo.push_back(u);
        }
        if (color[v] != 'B') continue;
        DetectarCiclo(v, u);
        if (color[u] == 'G' && color[v] == 'R')
            color[u] = 'R', ciclo.push_back(u);
    }
    if (color[u] == 'G') color[u] = 'N';
}
vector<vector<int>> DetectarCiclos() {
    vector<vector<int>> ciclos;
    color = vector<char>(n, 'B');
    for (int u = 0; u < n; ++u) {
        if (color[u] != 'B') continue;
        ciclo.clear(); DetectarCiclo(u, n);
        reverse(ciclo.begin(), ciclo.end());
        if (!ciclo.empty())
            ciclos.push_back(ciclo);
    }
    return ciclos;
}
```

Puentes y Puntos de Articulación

```
// Deteccion de puentes y puntos de articulacion
// en 1 grafo o multigrafo bidireccional.Los puentes
// quedan guardados en un vector de aristas. Los puntos
// de articulacion son marcados como true o false
// respectivamente en un vector de booleanos.
int tiempo;
vector<int> label, low;
vector<Arista> puentes; // <- Resultado
vector<bool> articulacion; // <- Resultado
int PuentesArticulacion(int u, int p) {
    label[u] = low[u] = ++tiempo;
    int hijos = 0, retorno = 0;
    for (int v : ady[u]) {
        if (v == p && !retorno++) continue;
        if (!label[v]) {
            ++hijos; PuentesArticulacion(v, u);
            if (label[u] <= low[v])
                articulacion[u] = true;
            if (label[u] <  low[v])
                puentes.push_back(Arista(u, v));
            low[u] = min(low[u], low[v]);
        }
        low[u] = min(low[u], label[v]);
    }
    return hijos;
}
```

```
void PuentesArticulacion() {
    low = vector<int>(n);
    label = vector<int>(n);
    tiempo = 0, puentes.clear();
    articulacion = vector<bool>(n);
    for (int u = 0; u < n; ++u)
      if (!label[u])
          articulacion[u] =
                    PuentesArticulacion(u, n) > 1;
}
```

Componentes Fuertemente Conexas

```
// Deteccion de componentes fuertemente conexas
// en un grafo dirigido. Las componentes quedan
// guardadas en un vector de vectores, donde
// cada vector interno contiene los nodos
// de una componente fuertemente conexa.
vector<vector<int>> scc; // <- Resultado
int top; vector<int> pila;

void FuertementeConexo(int u) {
    label[u] = low[u] = ++tiempo;
    pila[++top] = u;
    for (int v : ady[u]) {
        if (!label[v]) FuertementeConexo(v);
        low[u] = min(low[u], low[v]);
    }
    if (label[u] == low[u]) {
        vector<int> componente;
        while (pila[top] != u) {
            componente.push_back(pila[top]);
            low[pila[top--]] = n + 1;
        }
        componente.push_back(pila[top--]);
        scc.push_back(componente);
        low[u] = n + 1;
    }
}
```

```
void FuertementeConexo() {
    low = vector<int>(n);
    label = vector<int>(n);
    tiempo = 0, scc.clear();
    top = -1, pila = vector<int>(n);
    for (int u = 0; u < n; ++u)
        if (!label[u]) FuertementeConexo(u);
}
```

EsBipartito

```
// Determina si un grafo bidireccional
// es bipartito (o bien, es bicoloreable).
bool EsBipartito() {
    vector<char> color(n, -1);
    for (int u = 0; u < n; ++u) {
        if (color[u] >= 0) continue;
        color[u] = 0;
        queue<int> q; q.push(u);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int v : ady[u]) {
                if (color[v] < 0) q.push(v),
                        color[v] = 1 - color[u];
                if (color[u] == color[v])
                    return false;
            }
        }
    }
    return true;
}
```

Orden Topológico

```
// Obtiene el orden topologico de los nodos
// de un grafo dirigido. Orden ascendente
// respecto al numero de dependencias.
vector<bool> vis;
vector<int> ordenados;
void OrdenTopologico(int u) {
    vis[u] = true;
    for (int v : ady[u])
        if (!vis[v]) OrdenTopologico(v);
    ordenados.push_back(u);
}
```

```
void OrdenTopologico() {
    ordenados.clear();
    vis = vector<bool>(n);
    for (int u = 0; u < n; ++u)
        if (!vis[u]) OrdenTopologico(u);
}
```

BFS

```
// Busqueda en amplitud desde el nodo s.
// Devuelve el vector de distancias a todos
// los nodos desde s. Un valor INF indica que
// no es posible ir de s al respectivo nodo.
vector<Costo> BFS(int s) {
    queue<int> q;
    vector<Costo> d(n, INF);
    q.push(s), d[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : ady[u])
            if (d[u] + 1 < d[v])
                d[v] = d[u] + 1, q.push(v);
    }
    return d;
}
}; //Fin Estructura Grafo
```

Union-Find

```
// Conjuntos disjuntos con Union-Find.
struct UnionFind {
    int n; vector<int> padre, tam;
    UnionFind(int N) : n(N), tam(N, 1), padre(N) {
        while (--N) padre[N] = N;
    }

    int Raiz(int u) {
        if (padre[u] == u) return u;
        return padre[u] = Raiz(padre[u]);
    }

    bool SonConexos(int u, int v) {
        return Raiz(u) == Raiz(v);
    }

    void Unir(int u, int v) {
        int Ru = Raiz(u); int Rv = Raiz(v);
        if (Ru == Rv) return;
        --n, padre[Ru] = Rv;
        tam[Rv] += tam[Ru];
    }

    int Tamano(int u) {
        return tam[Raiz(u)];
    }
};
```

GrafoCosto

```
typedef pair<Costo, int> CostoNodo;
typedef pair<Costo, Arista> Ponderada;

// Grafos con ponderacion.
// Nodos indexados de 0 a n - 1.
// GrafoPonderado(n,  true) -> Bidireccional.
// GrafoPonderado(n, false) -> Dirigido.

struct GrafoPonderado {
    int n; bool bi;
    vector<vector<CostoNodo>> ady;
    GrafoPonderado(int N, bool B = true): n(N),
        bi(B), ady(N) {}

    void AgregarArista(int u, int v, Costo c) {
        if (bi) ady[v].push_back(CostoNodo(c, u));
        ady[u].push_back(CostoNodo(c, v));
    }
```

Kruskal

```
// Obtiene el arbol de expansion minima de un
// grafo bidireccional. Para obtener el arbol
// de expansion maxima descomentar el reverse.
// En caso de tener varias componentes conexas,
// obtiene el bosque de expansion minima.
vector<Ponderada> Kruskal() {
    vector<Ponderada> todas;
    for (int u = 0; u < n; ++u)
        for (CostoNodo arista : ady[u])
            todas.push_back(Ponderada(arista.first,
                        Arista(u, arista.second)));
    sort(todas.begin(), todas.end());
    // reverse(todas.begin(), todas.end());
    vector<Ponderada> mst;
    UnionFind componentes(n);
    for (Ponderada arista : todas) {
        int u = arista.second.first;
        int v = arista.second.second;
        if (!componentes.SonConexos(u, v))
            componentes.Unir(u, v),
                    mst.push_back(arista);
    }
    return mst;
}
```

Dijkstra

```
// Algoritmo de dijkstra desde el nodo s.
// Devuelve el vector de distancias a todos
// los nodos desde s. Un valor INF indica que
// no es posible ir de s al respectivo nodo.
vector<Costo> Dijkstra(int s) {
    vector<Costo> dist(n, INF);
    priority_queue<CostoNodo> pq;
    pq.push(CostoNodo(0, s)), dist[s] = 0;
    while (!pq.empty()) {
        Costo p = -pq.top().first;
        int u = pq.top().second; pq.pop();
        if (dist[u] < p) continue;
        for (CostoNodo arista : ady[u]) {
            int v = arista.second;
            p = dist[u] + arista.first;
            if (p < dist[v]) dist[v] = p,
                        pq.push(CostoNodo(-p, v));
        }
    }
    return dist;
}
```

BellmanFerrari

```
// Algoritmo de Bellman-Ford optimizado, desde
// el nodo s. Devuelve un booleano indicando si
// existe un ciclo negativo en un digrafo.
// Obtiene el vector de distancias a todos.
vector<Costo> dist; // <- Resultado
bool BellmanFerrari(int s) {
    queue<int> q;
    vector<bool> en_cola(n);
    vector<int> procesado(n);
    dist = vector<Costo>(n, INF);
    q.push(s), dist[s] = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop(), en_cola[u] = false;
        if (++procesado[u] == n) return true;
        for (CostoNodo arista : ady[u]) {
            int v = arista.second;
            Costo p = arista.first;
            if (dist[u] + p < dist[v]) {
                if (!en_cola[v]) q.push(v);
                dist[v] = dist[u] + p;
                en_cola[v] = true;
            }
        }
    }
    return false;
}
}; //Fin estructura GrafoCosto
```

Árboles

LCA

```
#define MAXV 105
#define LOGV 10
#define mp make_pair
using namespace std;
int level[ MAXV ], parent[ MAXV ], N;
int P[ MAXV ][ LOGV ], maximo[ MAXV ][ LOGV ],
peso[ MAXV ];

void PRE(){
    for(int u = 0; u < N; u++){
        P[u][0] = parent[u];
        maximo[u][0] = peso[u];
    }
    for(int i = 1; (1 << i) <= N; i++){
        for(int u = 0; u < N; u++){
            P[u][i] = P[P[u][i-1]][i-1];
            maximo[u][i] = max( maximo[u][i - 1],
                       maximo[P[u][i - 1]][i - 1] );
        }
    }
}
int LCA(int p , int q){
    if(level[p] < level[q]) swap(p,q);
    int lg;
    for( lg = 1 ; (1 << lg) <= level[p] ; ++lg );
        lg--;
    int maxi = 0;
    for( int i = lg ; i >= 0 ; i--){
        if( level[p] - (1 << i) >= level[q] ){
            maxi = max( maxi, maximo[p][i] );
            p = P[p][i];
        }
    }
    if( p == q ) return maxi;
    for( int i = lg ; i >= 0 ; i-- ){
        if( P[p][i] != -1 && P[p][i] != P[q][i]){
            maxi = max( maxi, max( maximo[q][i],
                                   maximo[p][i] ));
            p = P[p][i];
            q = P[q][i];
        }
    }
    maxi = max(max( maximo[p][0], maxi ) ,
                    maximo[q][0]);
    return maxi;
}
```

Ejemplo LCA

```
int main(){
    int A, Q, B;
    lli L;
    while( 1 ){
        cin >> N;
        if( !N ) break;
        for( int i = 0; i <= N; i++ ){
            dist[ i ] = 0;
            level[ i ] = 1;
            parent[ i ] = i;
        }
        for( int i = 1; i < N; i++ ){
            cin >> A >> L;
            parent[ i ] = A;
            dist[ i ] = ( L + dist[ A ] );
            level[ i ] += level[ A ];
        }
        PRE();
        cin >> Q;
        for( int i = 0; i < Q; i++ ){
            cin >> A >> B;
            if( i ) cout << ' ';
            cout << dist[ A ] + dist[ B ] -
                   ( 2ll * dist[ LCA( A, B ) ] );
        }
        cout << '\n';
    }
}
```

Heavy-Light Decomposition

```
// Definiciones iniciales.
typedef vector<int> Lista;

// Arbol con Heavy-Light Decomposition.
// Los nodos estan indexados de 0 a n - 1.
struct HeavyLight {
    int n, conteo;
    Lista nivel, tamano, up;
    Lista indice, super, top;
    vector<Lista> aristas;
    HeavyLight(int N) : n(N), conteo(),top(N),
            nivel(N), tamano(N), up(N), indice(N),
            super(N), aristas(N) {}

    void AgregarArista(int u, int v) {
        aristas[u].push_back(v);
        aristas[v].push_back(u);
    }

    void CalcularNivel(int u, int p) {
        for(int i = 0; i < aristas[u].size(); ++i){
            int v = aristas[u][i];
            if (p == v) continue;
            if (super[u] == super[v])
                nivel[v] = nivel[u];
            else nivel[v] = nivel[u] + 1;
            CalcularNivel(v, u);
        }
    }
    // Construir realiza todas las operaciones para
    // trabajar con Heavy-Light. Por defecto, la
    raiz del     // arbol se establece como el nodo
    0. Si quieren definir una raiz diferente,
    llamen Construir(r) donde el parametro r indica
    cual sera la raiz del arbol.
    int Construir(int u = 0, int p = -1) {
        int tam_subarbol = 0;
        up[u] = p, super[u] = -1;
        for(int i = 0; i < aristas[u].size(); ++i){
            int v = aristas[u][i];
            if (p == v) continue;
            tam_subarbol += Construir(v, u);
        }
        for(int i = 0; i < aristas[u].size(); ++i){
            int v = aristas[u][i];
            if (p == v) continue;
            if (tamano[v] > tam_subarbol / 2)
                indice[u] = indice[v] + 1,
                super[u] = super[v],
                top[super[v]] = u;
        }
        if (super[u] == -1)
            super[u] = conteo, top[conteo++] = u;
        if (p == -1) CalcularNivel(u, p);
        return tamano[u] = tam_subarbol + 1;
    }
    int LCA(int u, int v) {
        if (nivel[v] > nivel[u]) swap(u, v);
        while (nivel[u] > nivel[v])
            u = up[top[super[u]]];
        while (super[u] != super[v])
            u = up[top[super[u]]],
            v = up[top[super[v]]];
        return (indice[u] > indice[v])? u: v;
    }
};
```

Grafos Guanajuato – Ejemplo

```
/* Ejemplo de Grafos guanajuato (Distancias
minimas):
MAXN - cantidad máxima de nodos
LOGN - logaritmo base 2 de MAXN
N - cantidad de nodos actual
M - cantidad de aristas
nxt[i] - continene el nodo con el que esta unido i
now - nodo con el que se inicia la busqueda del
ciclo
cntc - conteo/'mapeo'/id's de los ciclos
Rciclo[i] - contiene el id del ciclo en el que esta
el nodo 'i' (si es que pertenece)
Tciclo[i] - contiene el numero de nodos en el ciclo
con id 'i'
Nciclo[i] - al numerar los nodos en un ciclo, este
arreglo contiene el numero que se le dio al nodo
'i' en el ciclo (si es que pertenece a uno)
Pciclo[i] - contiene la raiz (nodo que es parte de
un ciclo) del arbol en el que esta el nodo 'i'
```

```
P[][] - Arreglo para el preproceso y recorrido para
LCA
pfsvis[] - visitados para la pfs
vis[] - visitados de la dfs
level[] - nivel del LCA */
const int MAXN = 100010, LOGN = 18;
int N, M, nxt[MAXN], now, cntc, Rciclo[MAXN],
Tciclo[MAXN], Nciclo[MAXN], P[MAXN][LOGN],
Pciclo[MAXN];
int pfsvis[MAXN], vis[MAXN], level[MAXN];

void pfs(int u) {
    pfsvis[u] = now;
    if (!pfsvis[nxt[u]]) pfs(nxt[u]);
    else if (pfsvis[nxt[u]] == now) {
        Rciclo[u] = ++cntc;
        Tciclo[cntc] = 1;
        Nciclo[u] = 1;
        for(int i=nxt[u], last=u; i!=u; last=i,
                                        i=nxt[i]){
            Rciclo[i] = Rciclo[last];
            Nciclo[i] = Nciclo[last] + 1;
            Tciclo[cntc]++;
        }
    }
}
void dfs(int u) {
    vis[u] = 1;
    if (!Rciclo[u]) {
        P[u][0] = nxt[u];
        if (!vis[nxt[u]]) dfs(nxt[u]);
        Pciclo[u] = Pciclo[nxt[u]];
        level[u] = level[nxt[u]] + 1;
    }
    else {
        P[u][0] = 0;
        Pciclo[u] = u;
        level[u] = 0;
    }
}
void PRE() {
    for (int d = 1; d < LOGN; ++d)
        for (int i = 1; i <= N; ++i)
    P[i][d] = P[P[i][d - 1]][d – 1];
}
```

```
int lca(int u, int v) {
    if (level[u] > level[v]) swap(u, v);
    for (int i = 0, j = level[v]-level[u]; i < LOGN
                            && j > 0; ++i, j >>= 1)
        if (j & 1) v = P[v][i];
    if (u == v) return u;
    for (int i = LOGN - 1; i >= 0; --i)
        if (P[u][i] != P[v][i]) {
        u = P[u][i];
        v = P[v][i];
    }
    return P[u][0];
}
int dist(int u, int v) {
    return level[u]+level[v]-(level[lca(u, v)]<<1);
}
int main() {
    while (cin >> N) {
        cntc = 0;
        memset(Rciclo, 0, sizeof Rciclo);
        memset(level, 0, sizeof level);
        memset(pfsvis, 0, sizeof pfsvis);
        memset(Nciclo, 0, sizeof Nciclo);
        memset(vis, 0, sizeof vis);
        for (int i = 1; i <= N; ++i)cin >> nxt[i];
        for (int i = 1; i <= N; ++i)if (!pfsvis[i])
                                    now = i, pfs(i);
        for(int i=1; i<=N; ++i) if(!vis[i]) dfs(i);
        PRE(); cin >> M;
        for (int i = 0; i < M; ++i) {
            int u, v; cin >> u >> v;
           if(Rciclo[Pciclo[u]]!=Rciclo[Pciclo[v]])
                cout << -1 << '\n';
            else if (Pciclo[u] == Pciclo[v])
                cout << dist(u, v) << "\n";
            else {
                int tmp = abs(Nciclo[Pciclo[u]] –
                              Nciclo[Pciclo[v]]);
        tmp=min(tmp,Tciclo[Rciclo[Pciclo[u]]]–tmp);
                cout << level[u]+level[v]+tmp << "\n";
            }
        }
    }
return 0;}
```

Flujos

```
// Definiciones iniciales.
typedef int Flujo;
// Ajustable.
typedef vector<int> Lista;
typedef pair<int, int> Par;
typedef vector<Flujo> Flujo1D;
typedef vector<Flujo1D> Flujo2D;
const Flujo FINF = 1 << 30;
```

Emparejamiento Bipartito

```
// Nodos indexados de 0 a n - 1.
struct Bipartito {
    int n; Lista pareja;
    vector<Lista> aristas;
    vector<bool> lado, visitado;
    Bipartito(int N) : lado(N), pareja(N),
            visitado(N), aristas(N), n(N) {}

    void AgregarArista(int u, int v) {
        aristas[u].push_back(v);
        aristas[v].push_back(u);
    }

    void AgregarIzq(int u) { lado[u] = true; }
    void AgregarDer(int u) { lado[u] = false; }

    int CaminoIncremental(int u) {
        visitado[u] = true;
        for (int i = 0; i < aristas[u].size(); ++i)
            if (pareja[aristas[u][i]] == -1)
                return pareja[aristas[u][i]] = u;
        for(int i = 0; i < aristas[u].size(); ++i){
            int v = aristas[u][i];
            if (visitado[pareja[v]]) continue;
            if (CaminoIncremental(pareja[v]) != -1)
                return pareja[v] = u;
        }
        return -1;
    }
    vector<Par> MaxEmparejamiento() {
        fill(pareja.begin(), pareja.end(), -1);
        for (int i = 0; i < n; ++i) {
            if (!lado[i]) continue;
            CaminoIncremental(i);
            fill(visitado.begin(),
                 visitado.end(), false);
        }
        vector<Par> pares;
        for (int i = 0; i < n; ++i)
            if (!lado[i] && pareja[i] != -1)
                pares.push_back(Par(pareja[i], i));
            return pares;//Cardinalidad =
                         pares.size()
    }
};
```

Flujo Máximo

```
// Nodos indexados de 0 a n - 1.
struct GrafoFlujo {
    int n; vector<Lista> aristas;
    Flujo2D cap, flujo;
    Lista padre, dist;
    GrafoFlujo(int N) : dist(N), padre(N),
            aristas(N), cap(N, Flujo1D(N)),
            flujo(N, Flujo1D(N)), n(N) {}
    void AgregarArista(int u, int v, Flujo c) {
        flujo[v][u] += c; // Solo dirigidas!
        cap[u][v] += c, cap[v][u] += c;
        aristas[u].push_back(v);
        aristas[v].push_back(u);
    }
```

Edmonds Karp

```
// Flujo maximo mediante Edmonds-Karp O(VE^2).
Flujo ActualizarFlujo(int u, Flujo f) {
    int p = padre[u];
    if (p == u) return f;
    f = ActualizarFlujo(p, min(f,
                    cap[p][u] - flujo[p][u]));
    flujo[p][u] += f;
    flujo[u][p] -= f;
    return f;
}
```

```
Flujo AumentarFlujo(int s, int t) {
    fill(padre.begin(), padre.end(), -1);
    queue<int> q; q.push(s); padre[s] = s;
    while (!q.empty()) {
        int u = q.front();
        q.pop(); if (u == t) break;
        for(int i=0; i<aristas[u].size(); ++i){
            int v = aristas[u][i];
            if (flujo[u][v] == cap[u][v] ||
                padre[v] != -1) continue;
            padre[v] = u, q.push(v);
        }
    }
    if (padre[t] == -1) return 0;
    return ActualizarFlujo(t, FINF);
}
Flujo EdmondsKarp(int s, int t) {
    Flujo flujo_maximo = 0, f;
    while (f = AumentarFlujo(s, t))
       flujo_maximo += f;
    return flujo_maximo;
}
```

Dinic

```
// Flujo maximo mediante Dinic O(V^2E).
Flujo FlujoBloqueante(int u, int t, Flujo f) {
    if (u == t) return f; Flujo fluido = 0;
    for(int i = 0; i < aristas[u].size(); ++i){
        if (fluido == f) break;
        int v = aristas[u][i];
        if (dist[u] + 1 > dist[v]) continue;
        Flujo fv = FlujoBloqueante(v, t,
                   min(f - fluido,
                     cap[u][v] - flujo[u][v]));
        flujo[u][v] += fv, fluido += fv;
        flujo[v][u] -= fv;
    }
    return fluido;
}
```

```
Flujo Dinic(int s, int t) {
    Flujo flujo_maximo = dist[t] = 0;
    while (dist[t] < INT_MAX) {
        fill(dist.begin(), dist.end(),INT_MAX);
        queue<int> q; q.push(s); dist[s] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for(int i = 0;i<aristas[u].size();++i){
                int v = aristas[u][i];
                if (flujo[u][v] == cap[u][v] ||
                    dist[v] <= dist[u]+1) continue;
                dist[v] = dist[u]+1, q.push(v);
            }
        }
        if (dist[t] < INT_MAX) flujo_maximo +=
                        FlujoBloqueante(s, t, FINF);
    }
    return flujo_maximo;
}
}; //Fin estructura GrafoFLujo
```

Definiciones Adicionales

```
// Definiciones adicionales.
typedef int Costo;
// Ajustable.
typedef vector<Costo> Costo1D;
typedef vector<Costo1D> Costo2D;
typedef pair<Costo, int> CostoNodo;
typedef pair<Flujo, Costo> FlujoCosto;
const double ERROR = 1e-9;
const Costo CINF = 1 << 30;
// Tolerancia en flotantes.
bool Igual(double a, double b) {
    return fabs(a - b) < ERROR;
}
```

Emparejamiento Bipartito de costo MAX/MIN

```
// EMPAREJAMIENTO BIPARTITO DE COSTO MAX/MIN
// Nodos indexados de 0 a n - 1, diferencia
// entre nodos en el conjunto izquierdo y derecho.
// Es posible que alguna variable se desborde y se
// cicle, para evitarlo cambien Dato a long long.
struct BipartitoCosto {
    Lista pareja, retorno; vector<bool> visitado;
    int n, s;
    Costo1D slack, etiqueta;
    Costo2D costo;
    // Emparejamiento de costo maximo S = 1
    // Emparejamiento de costo minimo S = -1
    BipartitoCosto(int N, int S = 1):
        costo(N, Costo1D(N, S * -CINF)), s(S),
        slack(2 * N), etiqueta(2 * N), pareja(2*N),
        retorno(2 * N), visitado(2 * N), n(N) {}
    void AgregarArista(int u, int v, Costo c) {
        costo[u][v] = c * s;
    }

vector<Par> EmparejamientoOptimo() {
    fill(pareja.begin(), pareja.end(), -1);
    fill(etiqueta.begin(), etiqueta.end(), 0);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            etiqueta[i] = max(etiqueta[i],
                              costo[i][j]);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            slack[j + n] = etiqueta[i] +
                 etiqueta[j + n] - costo[i][j];
        fill(visitado.begin(),
             visitado.end(),false);
        fill(retorno.begin(),retorno.end(), i);
        visitado[i] = true;
        bool emparejado = false;
        for (int j = 0; !emparejado; ++j) {
            int t = n;
            for (; t < 2 * n; ++t) {
                if (visitado[t]) continue;
                if (Igual(slack[t], 0)) break;
            }
            if (t < 2 * n) {
                visitado[t] = true;
                if (pareja[t] == -1) {
                    emparejado = true;
                    for (int p; ; t = p) {
                        pareja[t] = retorno[t];
                        p = pareja[retorno[t]];
                        pareja[retorno[t]] = t;
                        if(retorno[t]==i)break;
                    }
                } else {
                    visitado[t=pareja[t]]=true;
                    for (int k = 0; k<n; ++k){
                        Costo new_slack =
                                etiqueta[t] +
                                etiqueta[k + n]
                                - costo[t][k];
                        if (!Igual(new_slack,
                                 slack[k + n])
                                 && new_slack <
                                 slack[k+ n]){
                            slack[k + n]=new_slack;
                            retorno[k + n] = t;
                        }
                    }
                }
            } else {
                Costo d = CINF;
                for (int k = n; k < 2 * n; ++k)
                    if (!Igual(slack[k], 0))
                        d = min(d, slack[k]);
                for (int k = 0; k < n; ++k)
                    if(visitado[k]) etiqueta[k]-=d;
                    for (int k = n; k < 2 * n; ++k)
                       if(!visitado[k])slack[k]-=d;
                       else etiqueta[k] += d;
            }
        }
    }
    vector<Par> pares;
    for (int i = 0; i < n; ++i)
      if (!Igual(costo[i][pareja[i] - n], s*-CINF))
            pares.push_back(Par(i, pareja[i] - n));
    return pares; // Emparejamiento optimo.
}};// Fin bipartito costo
```

Flujo Memoria Optimizada(Dinic)

```
// FLUJO MEMORIA OPTIMIZADA Y
// FLUJO MAXIMO DE COSTO MINIMO
// Nodos indexados de 0 a n - 1.
// No utiliza matrices de adyacencia.
struct GrafoFlujoCosto {
    struct AristaFlujo {
        int dst; AristaFlujo* residual;
        Flujo cap, flujo; Costo peso, npeso;
        AristaFlujo(int d, Flujo f, Flujo c)
                : dst(d), flujo(f), cap(c) {}
        Costo AumentarFlujo(Flujo f) {
            residual->flujo -= f;
            this->flujo += f;
            return peso * f;
        }
    };
    int n; vector<Par> prv; Lista dist;
    vector< vector<AristaFlujo*> > aristas;
    GrafoFlujoCosto(int N) : n(N), aristas(N),
                             prv(N), dist(N) {}
    ~GrafoFlujoCosto() { for (int i = 0; i<n; ++i)
        for (int j = 0; j < aristas[i].size(); ++j)
        delete aristas[i][j];
        // NO OMITIR
    }
// Para aristas bidireccionales agreguen dos
// aristas dirigidas. Si las aristas no son
ponderadas dejen el ultimo parametro con el valor
por defecto.

void AgregarArista(
    int u, int v, Flujo c, Costo p = 0) {
        AristaFlujo* uv = new AristaFlujo(v, 0, c);
        AristaFlujo* vu = new AristaFlujo(u, c, c);
        uv->residual = vu, vu->residual = uv;
        uv->peso = uv->npeso =  p;
        vu->peso = vu->npeso = -p;
        aristas[u].push_back(uv);
        aristas[v].push_back(vu);
}
```

```
// Dinic para flujo maximo con memoria optimizada.
// Prefieran esta version solo cuando n > 5,000.
Flujo FlujoBloqueante(int u, int t, Flujo f) {
    if (u == t) return f; Flujo fluido = 0;
    for (int i = 0; i < aristas[u].size(); ++i) {
        if (fluido == f) break;
        AristaFlujo* v = aristas[u][i];
        if (dist[u] + 1 == dist[v->dst]) {
            Flujo fv = FlujoBloqueante(v->dst, t,
               min(f - fluido, v->cap - v->flujo));
            v->AumentarFlujo(fv), fluido += fv;
        }
    }
    return fluido;
}
Flujo Dinic(int s, int t) {
    Flujo flujo_maximo = dist[t] = 0;
    while (dist[t] < INT_MAX) {
        fill(dist.begin(), dist.end(), INT_MAX);
        queue<int> q; q.push(s); dist[s] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int i=0; i<aristas[u].size();++i){
                AristaFlujo* v = aristas[u][i];
                if (dist[v->dst]< INT_MAX)continue;
                if (v->flujo == v->cap) continue;
                dist[v->dst] = dist[u] + 1;
                q.push(v->dst);
            }
        }
        if (dist[t] < INT_MAX)
           flujo_maximo+=FlujoBloqueante(s,t,FINF);
    }
    return flujo_maximo;
}
```

Max-Flow Min-Cost

```
//Flujo de costo minimo en O(VElogV * flow). Si
dejan el valor por defecto del parametro k saca el
flujo maximo.
void RecalcularCosto(const Costo1D& pi) {
    for (int u = 0; u < n; ++u) {
        for(int i = 0; i < aristas[u].size(); ++i){
            AristaFlujo* v = aristas[u][i];
            v->npeso = v->npeso + pi[u]-pi[v->dst];
        }
    }
}
FlujoCosto ActualizarFlujo(int u, Flujo f) {
    int p = prv[u].first, i = prv[u].second;
    if (p == -1) return FlujoCosto(f, 0);
    AristaFlujo* pu = aristas[p][i];
    FlujoCosto res = ActualizarFlujo(p,
                     min(f, pu->cap - pu->flujo));
    res.second += pu->AumentarFlujo(res.first);
    return res;
}
FlujoCosto AumentarFlujo(int s, int t, Flujo f) {
    Costo1D dist(n, CINF);
    fill(prv.begin(), prv.end(), Par(-1, -1));
    priority_queue<CostoNodo, vector<CostoNodo>,
                   greater<CostoNodo> > pq;
    pq.push(FlujoCosto(0, s)); dist[s] = 0;
    while (!pq.empty()) {
        int u = pq.top().second;
        Costo p = pq.top().first; pq.pop();
        if (!Igual(dist[u], p)) continue;
        for (int i= 0; i < aristas[u].size(); ++i){
            AristaFlujo* v = aristas[u][i];
            if (v->flujo == v->cap) continue;
            Costo ndist = dist[u] + v->npeso;
            if (!Igual(ndist, dist[v->dst]) &&
                ndist < dist[v->dst]) {
                dist[v->dst] = dist[u] + v->npeso;
                pq.push(CostoNodo(ndist, v->dst));
                prv[v->dst].second = i;
                prv[v->dst].first = u;
            }
        }
    }
    if(Igual(dist[t], CINF))return FlujoCosto(0,0);
    RecalcularCosto(dist);
    return ActualizarFlujo(t, f);
}
FlujoCosto FlujoCostoMin(int s,int t, Flujok=FINF){
    Costo1D dist(n, CINF); dist[s] = 0;
    for (int i = 0; i < n; ++i) {
        for (int u = 0; u < n; ++u) {
            if (Igual(dist[u], CINF)) continue;
            for(int j=0; j<aristas[u].size(); ++j){
                AristaFlujo* v = aristas[u][j];
                if (v->flujo < v->cap)
                    dist[v->dst]= min(dist[v->dst],
                                  dist[u] + v->npeso);
            }
        }
    }
    RecalcularCosto(dist);
    FlujoCosto flujo_costo(0, 0);
    while (flujo_costo.first < k) {
        FlujoCosto fc = AumentarFlujo(s, t, k-
                        flujo_costo.first);
        flujo_costo.second += fc.second;
        flujo_costo.first += fc.first;
        if (!fc.first) break;
    }
    return flujo_costo;
}
}; //Fin estructura GrafoFlujoCosto
```

Estructuras de Datos

Sparse Table

```
#define ll long long
#define MAXN 100005
#define LOGN 20
using namespace std;
ll sparse [MAXN][LOGN], A[MAXN];
int logs[MAXN];

ll gcd(ll a, ll b){
    if(b==0)
        return a;
    ll GCD=gcd(b, a%b);
    return GCD;
}
void pre(int N){
    for(int i = 2; i <= N; i++)
        logs[i] = logs[i/2] + 1;
    for(int i = 0; i < N; i++)
        sparse[i][0] = A[i];
    for(int i = 0; i < logs[N]; i++)
        for(int j = 0; j < N; j++)
            sparse[j][i+1] = gcd(sparse[j][i],
                sparse[min(j + (1<<i), N-1)][i]);
}

ll query (int a, int b){
    if(a == b) return sparse[a][0];
    int L = logs[b-a+1];
    return gcd(sparse[a][L],
               sparse[b-(1 << L)+1][L]);
}
```

SegmentTree Dinámico

```
const int INF = 1 << 30;
// Segment Tree version dinamica. Para generar el
// arbol completo deben llamar a la funcion
Construir. CUIDADO: Para usarlo deben especificar
el tipo de dato a utilizar;
template<class T> struct SegTree {
    T dato; int i, d;
    SegTree* izq, *der;
    SegTree(int I, int D): izq(NULL), der(NULL),
                           i(I), d(D), dato() {}
    ~SegTree() {
        if (izq) delete izq;
        if (der) delete der;
    }
    T Construir() {
     if (i == d) return dato = T();
     int m = (i + d) >> 1;
     izq = new SegTree(i, m);
     der = new SegTree(m + 1, d);
     return dato=izq->Construir()+der->Construir();
    }
    T Actualizar(int a, T v) {
        if (a < i || d < a) return dato;
        if (a == i && d == a) return dato = v;
        if (!izq) {
            int m = (i + d) >> 1;
            izq = new SegTree(i, m);
            der = new SegTree(m + 1, d);
        }
        return dato = izq->Actualizar(a, v) +
                      der->Actualizar(a, v);
    }
    T Query(int a, int b) {
        if (b < i || d < a) return T();
        if (a <= i && d <= b) return dato;
        return izq? izq->Query(a, b) +
                    der->Query(a, b): T();
    }
};
```

```
// A continuación se ejemplifica como sobrecargar
// el operador + dentro de una estructura para
poder reutilizar el codigo del Segment Tree
facilmente. El ejemplo sobrecarga el + por la
funcion de maximo. Es MUY IMPORTANTE tener un
constructor por defecto.
struct MaxInt {
    int d; MaxInt(int D) : d(D) {}
    MaxInt() : d(-INF) {} // IMPORTANTE!
    MaxInt operator+(const MaxInt& o) {
        return MaxInt(max(d, o.d));
    }
};
```

Segment Tree (Single U, Range Q)

```
const int SIZE = (1 << 22) + 5;
const int MINF = -(1e9);
typedef pair<int, int> data;
//ST indexado desde uno
data ST[SIZE];

data Combina(data A, data B) {
    if (B.first > A.first) {
        swap(A.first, B.first);
        swap(B.first, A.second);
        if (B.second > A.second)
            swap(A.second, B.second);
    } else if (B.first > A.second)
        swap(B.first, A.second);
    return A;
}

void Update(int a,int b,int v,int n, int L, int R){
    if (L > b || R < a) return;
    if (L >= a && R <= b) {
        ST[n] = data(v, MINF); return;
    }
    int mid = (L + R) >> 1;
    Update(a, b, v, n << 1, L, mid);
    Update(a, b, v, (n << 1) + 1, mid + 1, R);
    ST[n] = Combina (ST[(n << 1)], ST[(n << 1)+1]);
}
```

```
data Query(int a, int b, int n, int L, int R) {
    if (L > b || R < a) return data(MINF, MINF);
    if (L >= a && R <= b) return ST[n];
    int mid = (L + R) >> 1;
    return Combina(Query(a, b, n << 1, L, mid),
            Query(a, b, (n << 1) + 1, mid + 1, R));
}
int main() {
    int N, Q, a, b;
    char opc;
    data ans;
    cin >> N;
    for (int i = 1; i <= N; i++) {
        cin >> a;
        //Update o cambio de valor en el Arreglo
          posicion i
        Update(i, i , a, 1, 1, N);
    }
    cin >> Q;
    for (int i = 0; i < Q; ++i) {
        cin >> opc >> a >> b;
        if (opc == 'Q') {
            //Query del rango a - b en el arreglo
            ans = Query(a, b, 1, 1, N);
            if (a == b) cout << ans.first<<'\n';
            else  cout<<ans.first+ans.second<<'\n';
        }
        else Update(a, a, b, 1, 1, N);
    }
}
```

Segment Tree Lazy (Single U, Range Q)

```
const int MAXN = 500009;
typedef long long lli;
lli ST[MAXN], Lazy[MAXN], arr[MAXN], N;
void Build(int n = 1, int L = 1, int R = N) {
    if (L == R) {
        ST[n] = arr[L];
        return;
    }
    int ls = (n << 1), rs = ls + 1, m =(L + R) >>1;
    Build(ls, L, m); Build(rs, m + 1, R);
    ST[n] = max(ST[ls], ST[rs]);
}
```

```
void Update(int r, lli v, int l = 1, int n = 1, int
L = 1, int R = N) {
    if ((L > r) || (R < l))  return;
    int ls = (n << 1), rs = ls + 1, m = (L + R)>>1;
    if (Lazy[n] && (L != R)) {
        ST[ls] = Lazy[n];
        ST[rs] = Lazy[n];
        Lazy[ls] = Lazy[n];
        Lazy[rs] = Lazy[n];
        Lazy[n] = 0;
    }
    if ((L >= l) && (R <= r)) {
        ST[n] = v;
        Lazy[n] = v;
        return;
    }
    Update(r, v, l, ls, L, m);
    Update(r, v, l, rs, m + 1, R);
    ST[n] = max(ST[ls], ST[rs]);
}
lli Query(int l, int r, int n = 1, int L = 1, int R
= N) {
    if ((L > r) || (R < l))  return -1LL;
    int ls = (n << 1), rs = ls + 1, m = (L + R)>>1;
    if (Lazy[n] && (L != R)) {
        ST[ls] = Lazy[n];
        ST[rs] = Lazy[n];
        Lazy[ls] = Lazy[n];
        Lazy[rs] = Lazy[n];
        Lazy[n] = 0;
    }
    if ((L >= l) && (R <= r))  return ST[n];
    return max(Query(l, r, ls, L, m),
               Query(l, r, rs, m + 1, R));
}
```

```
int main() {
    cin >> N;
    for (int i = 1; i <= N; ++i) cin >> arr[i];
    Build();
    int Q; cin >> Q;
    for (int i = 0; i < Q; ++i) {
        int I, J;
        cin >> I >> J;
        cout << Query(1, I) << '\n';

        Update(I, (Query(1, I) + J));
    }
    return 0;
}
```

Estructura Fenwick Tree

```
// Fenwick Tree. Indices de 1 a n.
struct FenTree {
    vector<int> tree;
    FenTree(int n) : tree(n + 1) {}

    void Actualizar(int i, int v) {
        while (i < tree.size()) {
            tree[i] += v;
            i += i & -i;
        }
    }
    int Query(int i) {
        int sum = 0;
        while (i > 0) {
            sum += tree[i]; i -= i & -i;
        }
        return sum;
    }
    int Rango(int i, int j) {
        return Query(j) - Query(i - 1);
    }
};
```

Pila Incremental

```
long long H[1000009], W[1000009];
int main(){
    int N; cin >> N;
    for (int i = 0; i < N; ++i)
        cin >> H[i], W[i] = 1;
    H[N] = -1;  W[N] = 1;
    stack<int> Q;
    long long res = -1, cnt;
    int qt;
    for (int i = 0; i <= N; ++i) {
        if (Q.empty()) Q.push(i);
        else {
            if (H[i] >= H[Q.top()]) Q.push(i);
            else {
                qt = Q.top(); cnt = 0;
                while (H[qt] > H[i]) {
                    cnt += W[qt];
                    res = max(res, H[qt] * cnt);
                    Q.pop();
                    if (Q.empty()) break;
                    qt = Q.top();
                }
                W[i] += cnt;
                Q.push(i);
            }
        }
    }
    cout << res << "\n";
}
```

Strings

Suffix Array

```
void BucketSort(vector<int>& sa,
                const vector<int>&rank, int ranks){
    vector<int> bucket(ranks, 0);
    vector<int> tmp_sa(sa.size());
    for (int i = 0; i < sa.size(); ++i)
        ++bucket[rank[sa[i]]];
    for (int i = 0, sum = 0; i < ranks; ++i)
        swap(bucket[i], sum), sum += bucket[i];
    for (int i = 0; i < sa.size(); ++i)
        tmp_sa[bucket[rank[sa[i]]]++] = sa[i];
    swap(sa, tmp_sa);
}
// Recuerden poner '$' al final de la cadena.
vector<int> SuffixArray(const string& str) {
    int ranks = 255; vector<int> sa(str.size());
    vector<int> nrank(str.size());
    vector<int> rank(str.size(), 0);
    vector<int> tmp_rank(str.size());
    for (int i = 0; i < str.size(); ++i)
        nrank[i] = str[i], sa[i] = i;
    for (int p = 0; true; ++p) {
        BucketSort(sa, nrank, ranks + 1);
        BucketSort(sa,  rank, ranks + 1);
        tmp_rank[0] = ranks = 0;
        for (int i = 1; i < str.size(); ++i)
            if (rank[sa[i]] != rank[sa[i - 1]] ||
                nrank[sa[i]] != nrank[sa[i - 1]])
                tmp_rank[i] = ++ranks;
            else tmp_rank[i] = ranks;
            if (ranks + 1 == str.size()) break;
            for (int i = 1; i <= 1 << p; ++i)
                nrank[str.size() - i] = 0;
            for (int i = 0; i < str.size(); ++i) {
                int prv = sa[i] - (1 << p);
              if (prv >= 0) nrank[prv]=tmp_rank[i];
                rank[sa[i]] = tmp_rank[i];
            }
    }
    return sa;
}
```

```
vector<int> LCP(const string& str,
                const vector<int>& sa) {
    vector<int> lcp(str.size());
    vector<int> plcp(str.size());
    vector<int> phi(str.size(), -1);
    for(int i = 1; i < str.size(); ++i)
        phi[sa[i]] = sa[i - 1];
    int len = 0;
    for(int i = 0; i < str.size(); ++i) {
        if (phi[i] == -1) continue;
        for (; str[phi[i] + len] ==
               str[i + len]; ++len) {}
        plcp[i] = len; len = max(len-1, 0);
    }
    for (int i = 0; i < str.size(); ++i)
        lcp[i] = plcp[sa[i]];
    return lcp;
}
```

Ejemplo

```
int main() {
    string cadena; cin >> cadena;
    cadena += '$';
    vector <int> sufix = SuffixArray(cadena);
    vector<int> lcp = LCP(cadena , sufix);
    for (int i = 0; i < cadena.size(); i++){
        cout << sufix[i] << "\t" <<
            cadena.substr(sufix[i])<< "\t" <<
            lcp[i] << endl;
    }
    return 0;
}
```

KMP

```
int F[1000009];
//Pattern and Text
string P, T;
void CF() {
    F[0] = -1;
    for (int i = 0, j = -1; P[i]; ) {
        while (~j && P[i] != P[j]) j = F[j];
        F[++i] = ++j;
    }
}
```

```
void KMPSearch() {
  int i = 9, j = 0;
  while (i < T.size()) {
    while (j >= 0 && T[i] != P[j]) j = F[j];
    i++; j++;
    if (j == P.size()) {
      printf("P is found at index %d in T\n", i-j);
      j = F[j];
    }
  }
}
```

Hashing

```
#define ull unsigned long long
const int MAXC = 20000005;
const ull MOD = 10000000000000003LL;
const ull MOD2 = 10000000000000013LL;

ull HB[MAXC], B = 71;
ull Multiplicar(ull a, ull b, ull m) {
    ull res = 0, p = a;
    for (; b; b >>= 1) {
        if (b & 1) res = (res + p) % m;
        p = (p + p) % m;
    }
    return res;
}

ull Subs(const vector<ull>& hasH, int a, int b) {
    return (hasH[b] - Multiplicar(hasH[a - 1]
                    , HB[b - a + 1], MOD)) % MOD;
}

ull F(char c) {
    return c - 'a' + 1;
}

void CB() {
    HB[0] = 1; HB[1] = B;
    for (int i = 2; i < MAXC; ++i)
        HB[i] = Multiplicar(HB[i - 1], B, MOD);
}
```

```
vector<ull> MhasH(string s) {
    vector<ull> hasH(s.size() + 1, 0);
    for (int i = 1; i <= s.size(); ++i)
        hasH[i] = (Multiplicar(hasH[i - 1], B, MOD)
    + F(s[i - 1])) % MOD;
    return hasH;
}
```

Minimum Rotating String & BinSearch (returns first inequality index)

```
    cin >> s; int N = s.size();
    ss = s + s;
    vector<ull> THash = Mhash(ss);
    int i = 0, k = 1, ans;
    for (; k < N; ++k) {
        //cout << Subs(THash, i + 1, i + N) << " -
          " << Subs(THash, k + 1, k + N) << '\n';
        if (Subs(THash, i + 1, i + N) ==
            Subs(THash, k + 1, k + N)) continue;
        int in = 1, fin = N + 1, mid;
        while (in < fin) {
            mid = (in + fin) / 2;
            if (Subs(THash, i + 1, i + mid) !=
                Subs(THash, k + 1, k + mid))
                fin = mid;
            else in = mid + 1;
        }
        //cout << "Comparamos: " << ss.substr(i, N)
          << " Con: " << ss.substr(k, N) << '\n';
        //cout << "Salieron diferentes del
          caracter: " << fin << '\n';
        if (ss[i + fin - 1] > ss[k + fin - 1]) {
            //cout<<ss[i + fin - 1] <<"-"<< ss[k +
              fin - 1] << " " << k << '\n';
            ans = k; i = k;
        }
    }
    cout << ans << '\n';
```

Trie

```
struct nodoTrie{
    int w;
  //El numero de palabras que terminan en este nodo
    int p;
//El número de palabras que tienen el camina de la
raíz a este nodo como prefijo
    int h[26];
    //Los hijos del nodo para cada letra
    void inicializa(){
        w = p = 0;
        for(int i = 0 ; i < 26 ; i++)
           h[i] = -1;//-1 para hijos aún no creados

    }
};
struct Trie{
    int inc;
    vector<nodoTrie>nodos;
    //Inicializa la cosa esta.
    //rep = 1 si hay repetidos. 0 si no hay.
    Trie(int rep){ inc = rep;
        nodos = vector<nodoTrie>(1);
        nodos[0].inicializa();
    }

    //Agrega una palabra al trie.
    void AgregaPalabra(string &cadena){
        int posS = 0, posT = 0 , sigN ;
        while(posS < cadena.size()){
          sigN = nodos[posT].h[cadena[posS]-'a'];
          if(sigN != -1) posT = sigN;
          else {
             sigN = nodos.size();
             nodos.resize(sigN+1);
             nodos[sigN].inicializa();
             nodos[posT].h[cadena[posS]-'a']= sigN;
             posT = sigN;
          }
          posS++; nodos[posT].p++;
        }
        if(!nodos[posT].w) nodos[posT].w = 1;
        else nodos[posT].w += inc;
    }
```

```
//Si hay repetidos dice cuantas veces aparece la
palabra. Si no hay repetido, dice si está o no está
esa palabra
int BuscaPalabra(string &cadena){
    int posS = 0;
    int posT = 0, sigN;
    while(posS < cadena.size()){
        sigN = nodos[posT].h[cadena[posS]-'a'];
        if(sigN == -1) return 0;
        posT = sigN;
        posS++;
    }
    return nodos[posT].w;
}

//Regresa la cantidad de palabras que tienen a
$cadena$ como prefijo
int Prefijo(string &cadena){
    int posS = 0; int posT = 0, sigN;
    while(posS < cadena.size()){
        sigN = nodos[posT].h[cadena[posS]-'a'];
        if(sigN == -1) return 0;
        posT = sigN;
        posS++;
    }
    return nodos[posT].p;
}
};
```

Matemagias

```
typedef long long Long;
```

Factores Primos

```
// Factores primos de un numero a.
typedef pair<int, int> Factor;
vector<Factor> FactoresPrimos(int a) {
    int conteo = 0;
    vector<Factor> factores;
    while (!(a & 1)) ++conteo, a >>= 1;
    if (conteo) factores.push_back(Factor(2,
                conteo)), conteo = 0;
    int raiz = sqrt(a);
    for (int i = 3; i <= raiz; i += 2) {
        while (!(a % i)) ++conteo, a /= i;
        if (conteo) factores.push_back(Factor(i,
                              conteo)), conteo = 0;
    }
    if (a > 1) factores.push_back(Factor(a, 1));
    return factores;
}
```

Criba

```
// Criba de Eratostenes de 1 a n.
vector<int> Criba(int n) {
    int raiz = sqrt(n);
    vector<int> criba(n + 1);
    for (int i = 4; i <= n; i += 2)
        criba[i] = 2;
    for (int i = 3; i <= raiz; i += 2)
        if (!criba[i])
            for (int j = i * i; j <= n; j += i)
                if (!criba[j]) criba[j] = i;
    return criba;
}
```

Factores del Factorial

```
// Factores primos de n factorial (n!). El vector
de primos debe estar ordenado.
vector<Factor> FactoresFactorial(int n,
                    const vector<int>& primos) {
    vector<Factor> factores;
    for (int i = 0; i < primos.size(); ++i) {
        if (n < primos[i]) break;
        int p = primos[i]; int reps = n / p;
        while (primos[i] <= n / p)
            p *= primos[i], reps += n / p;
        factores.push_back(Factor(primos[i],reps));
    }
    return factores;
}
```

Exponenciación Binaria

```
// Exponenciacion binaria a^n mod m.
Long Exponenciar(Long a, Long n, Long m) {
    Long resultado = 1;
    for (; n; n >>= 1) {
        if (n & 1) resultado = (resultado * a) % m;
        a = (a * a) % m;
    }
    return resultado;
}
```

Multiplicación Binaria

```
// Multiplicacion binaria a*b mod m.
Long Multiplicar(Long a, Long b, Long m) {
    Long resultado = 0;
    for (; b; b >>= 1) {
        if (b & 1) resultado = (resultado + a) % m;
        a = (a + a) % m;
    }
    return resultado;
}
```

Euclides Extendido

```
// Algoritmo de Euclides extendido entre a y b.
Además de devolver el gcd(a, b), resuelve la
identidad de Bezout con el par (x, y).
Si el parámetro mod no es especificado, se resuelve
con aritmetica normal; si mod se especifica, la
identidad se resuelve modulo mod.
Long Euclides(Long a, Long b, Long& x,
              Long& y, Long mod = 0) {
    if (!b) { x = 1, y = 0; return a; }
    Long gcd = Euclides(b, a % b, x, y, mod);
    x = !mod? x - y * (a / b):
             (mod + x - (y * (a / b)) % mod) % mod;
    swap(x, y);
    return gcd;
}
```

Estructura Fracción

```
// Tipo de dato para operar fracciones.
struct Fraccion {
    Long num, den;
    Fraccion() : num(0), den(1) {}
Fraccion(Long n, Long d) {
    if (d < 0) n = -n, d = -d;
    Long gcd = __gcd(abs(n), abs(d));
    num = n / gcd, den = d / gcd;
}
Fraccion operator-() const {
    return Fraccion(-num, den);
}
Fraccion operator+(const Fraccion& f) {
    Long gcd = __gcd(den, f.den);
    return Fraccion(num * (f.den / gcd)+
          f.num *(den / gcd), den * (f.den / gcd));
}
Fraccion operator-(const Fraccion& f) {
    return *this + -f; // a - b = a + (-b)
}
Fraccion operator*(const Fraccion& f) {
    return Fraccion(num * f.num, den * f.den);
}
Fraccion operator/(const Fraccion& f) {
    return Fraccion(num * f.den, den * f.num);
}
bool operator<(const Fraccion& cmp) {
    Long gcd = __gcd(den, cmp.den);
    return num * (cmp.den / gcd) <
           cmp.num * (den / gcd);
}
bool operator==(const Fraccion& cmp) {
    Long gcd = __gcd(den, cmp.den);
    return num * (cmp.den / gcd) ==
           cmp.num * (den / gcd);
}
}; //Fin estructura Fracción
```

Eliminación Gaussiana

```
// Eliminacion Gaussiana de matrices.
Definiciones iniciales para Gauss-Jordan.
typedef vector<double> Vector;
typedef vector<Vector> Matriz;
// Para eliminacion con fracciones.
Fraccion fabs(const Fraccion& f) {
    return Fraccion(abs(f.num), f.den);
}
bool EsCero(const Fraccion& f) {
    return f.num == 0;
}
// Para eliminacion con flotantes.
const double ERROR = 1e-9;
bool EsCero(double a) {
    return fabs(a) < ERROR;
}
```

Phi de Euler

```
long long EulerPhi(long long n){
    long long ans = n, i;
    for (i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
            while (n % i == 0) n /= i;
            ans -= ans / i;
        }
    }
    if (n > 1) ans -= ans / n
    return ans;
}
```

```
// En caso de no permitir el pivoteo (eg. cuando //
requieran sacar una matriz inversa) simplemente //
comenten o borren la seccion <comment>.
void EliminacionGaussiana(Matriz& m) {
  for (int i = 0; i < m.size(); ++i) {
    // <comment>
    int fila_mayor = i;
    for (int j = i + 1; j < m.size(); ++j)
        if (fabs(m[fila_mayor][i]) < fabs(m[j][i]))
            fila_mayor = j;
        swap(m[i], m[fila_mayor]);
    // </comment>
    if (EsCero(m[i][i])) continue;
    for (int j = m[i].size() - 1; j >= i; --j)
        m[i][j] = m[i][j] / m[i][i];
    for (int j = 0; j < m.size(); ++j) {
        if (i == j || EsCero(m[j][i])) continue;
        for (int k = m[j].size() - 1; k >= i; --k)
            m[j][k] = m[j][k] - m[i][k] * m[j][i];
    }
  }
}
```

Estructura Complejo

```
// Tipo de dato para operar numeros complejos.
struct Complejo {
    double real, imag;
    Complejo() : real(), imag() {}
    Complejo(double r, double i):real(r),imag(i) {}
    Complejo operator+(const Complejo& c) {
        return Complejo(real+c.real, imag+c.imag);
    }
    Complejo operator-(const Complejo& c) {
        return Complejo(real-c.real, imag-c.imag);
    }
    Complejo operator*(const Complejo& c) {
        return Complejo(real*c.real - imag*c.imag,
                    real * c.imag + imag * c.real);
    }
};
```

Fast And Fourier

```
// Transformada rapida de Fourier. Se tiene que
garantizar que el numero de elementos en el vector
sea una potencia de 2.
const double M_2PI = 2 * M_PI;
vector<Complejo> FastAndFourier(
            const vector<Complejo> &a, int k = 1) {
    int n = a.size();
    if (n == 1) return a;
    vector<Complejo> par, impar;
    for (int i = 0; i < n; ++i)
        if (i & 1) par.push_back(a[i]);
        else impar.push_back(a[i]);
    impar = FastAndFourier(impar, k);
    par = FastAndFourier(par, k);
    vector<Complejo> fourier(n);
    Complejo w(1, 0),
          wn(cos(-k * M_2PI/n), sin(-k * M_2PI/n));
    for (int i = 0; i < n/2; w = w * wn, ++i) {
        fourier[i + n/2] = impar[i] - w * par[i];
        fourier[i] = impar[i] + w * par[i];
    }
    return fourier;
}
```

Fast And Fourier Invertida

```
// Transformada inversa de Fourier. Se tiene que
garantizar que el numero de elementos en el vector
sea una potencia de 2.
vector<Complejo> InvFastAndFourier(
                    const vector<Complejo>& a) {
    vector<Complejo> ifft = FastAndFourier(a, -1);
    for (int i = 0; i < ifft.size(); ++i)
        ifft[i].real /= ifft.size(),
        ifft[i].imag /= ifft.size();
    return ifft; }
```

Convolución Discreta

```
// Convolucion discreta de dos vectores usando
transformada rapida de Fourier O(n log n).
Multiplica eficientemente dos polinomios.
Vector ConvolucionDiscreta( const Vector& a,
                            const Vector& b) {
    int n = a.size() + b.size() - 1;
    int p = 1; while (p < n) p <<= 1;
    vector<Complejo> A(p), B(p), C(p);
    for (int i = 0; i < a.size(); ++i)
        A[i] = Complejo(a[i], 0);
    for (int i = 0; i < b.size(); ++i)
        B[i] = Complejo(b[i], 0);
    A = FastAndFourier(A);
    B = FastAndFourier(B);
    for (int i = 0; i < p; ++i)
        C[i] = A[i] * B[i];
    C = InvFastAndFourier(C);
    Vector convolucion(n);
    for (int i = 0; i < n; ++i)
        convolucion[i] = C[i].real;
    return convolucion;
}
```

Tolerancia a flotantes

```
bool Igual(double a, double b) {
    return fabs(a - b) < ERROR;
}
```

Multiplicar Matrices

```
//Multiplica una matriz de tamano n*m por una
matriz de m*1, el resultado es una matriz de n*1
Matriz MultiplicarMatriz(const Matriz &a,
                         const Matriz &b){
    Matriz producto(a.size(),
            vector<double>(b[0].size(), 0.0000));
    for(int i = 0; i < a.size(); i++)
        for(int j = 0; j < b[0].size(); j++)
            for(int k = 0; k < b.size(); k++)
                producto[i][j] += a[i][k]*b[k][j];
    return producto;
}
```

Josephus Problem

//Se avanzan k pasos de forma circular durante n - 1 rondas hasta que solo uno quede vivo

g(n,k) = (g(n - 1, k) + k) mod n

g(1, k) = 0

Ley de Seno

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

$$Área = \frac{1}{2}bc(\sin(A)) = \frac{1}{2}ca(\sin(B)) = \frac{1}{2}ab(\sin(C))$$

Ley de Coseno

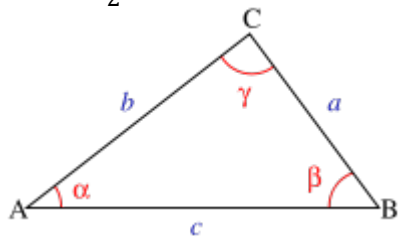
$$c^2 = a^2 + b^2 - 2ab * cos(C)$$
$$a^2 = b^2 + c^2 - 2bc * cos(A)$$
$$b^2 = a^2 + c^2 - 2ac * \cos(B)$$

Fórmula de Herón

$$Area = \sqrt{p * (p-a) * (p-b) * (p-c)}$$

Donde

$$p = \frac{a+b+c}{2}$$



Inverso modular

El inverso multiplicativo de un número a módulo m: x en $ax + my = 1$ o $a^{\varphi(m)-1}$ $modulo\ m$

Primos Grandes

10^9 + 9, 10^9 + 21, 10^9 + 33, 999999937, 999999929.

Triángulo de pascal

```
const int mod = 1e9 + 7;
C[0][0] = 1LL;
for (int i = 1; i < MAXN; ++i) {
    C[i][0] = 1LL;
    for (int j = 1; j <= i; j++)
       C[i][j]=(C[i - 1][j - 1] + C[i -1][j])% mod;
}
```

Combinatoria

```
Long Combinatoria(Long n, Long r){
    if (r > n)  return 0;
    if( r == 0 ) return 1;
    Long v = n--, inverso, x, y;
    for (Long i = 2; i < r + 1; ++i, --n){
        Euclides( i, modulo, x, y, modulo );
        v = (((v * n) % modulo) * x ) % modulo;
    }
    return v;
}
```

Radio incentro

$$r = \sqrt{\frac{(s-a)(s-b)(s-c)}{s}}$$

Radio circuncentro

$$r = \frac{a * b * c}{4 * Area}$$

Incentro

$$\boldsymbol{P}(\boldsymbol{x},\boldsymbol{y}) = \left(\frac{a\,x_a + bx_b + cx_c}{a+b+c}, \quad \frac{ay_a + by_b + by_c}{a+b+c}\right)$$

Definiciones

Triángulos

- La recta que une los puntos medios de dos lados de un triángulo es paralela al tercer lado e igual a su mitad, y se llama paralela media correspondiente al tercer lado.
- Mediatriz: La mediatriz de un lado de un triángulo se **define** como la recta perpendicular a dicho lado que pasa por su punto medio.
- Los puntos de la mediatriz de un lado de un triángulo equidistan de los vértices que definen dicho lado.
- Altura: La altura de un triángulo, respecto de uno de sus lados, se **define** como la recta perpendicular a dicho lado que pasa por el vértice opuesto.
- En un triángulo isósceles, la altura correspondiente al lado desigual divide el triángulo en dos triángulos iguales.
- Mediana: La mediana de un triángulo, correspondiente a uno de sus vértices, se **define** como la recta que une dicho vértice del triángulo con el punto medio del lado opuesto.
- Las tres medianas de un triángulo son interiores al mismo, independientemente del tipo de triángulo que sea.
- Cada mediana de un triángulo divide a éste en dos triángulos de igual área.
- Bisectriz: La bisectriz de un triángulo, correspondiente a uno de sus vértices, se **define** como la recta que, pasando por dicho vértice, divide al ángulo correspondiente en dos partes iguales.
- Los puntos de la bisectriz equidistan de los lados del ángulo.
- El baricentro de un triángulo, es un punto interior al mismo, que dista el doble de cada vértice que del punto medio de su lado opuesto.
- El Ortocentro, Baricentro y Circuncentro están siempre alineados. **(**Recta de Euler**)**
- El baricentro está entre el ortocentro y circuncentro.
- La distancia del baricentro al circuncentro es la mitad que la distancia del baricentro al ortocentro. M
- Incentro: **Punto** en el que se cortan las 3 bisectrices, es el centro de la circunferencia inscrita. Equidista de sus 3 lados, siendo tangente a ellos.
- Circuncentro: Es el punto de corte de las 3 mediatrices.

Extra

Fórmula de los caballos

```
typedef long long int Long;
struct punto{
    Long x,y;
    punto(){ x = 0, y = 0;}
};
int precalc[5][5] = {{0},
                     {3,2},
                     {2,1,4},
                     {3,2,3,2},
                     {2,3,2,5,4}};
Long calcula(punto A,punto B){
    Long x,y,tam;
    Long a,st,nd;
    x = abs(A.x-B.x);
    y = abs(A.y-B.y);
    if(x > y) swap(x,y);
    if(y < 5) return precalc[y][x];
    else {
        if(y & 1) tam = y/2 + 4;
        else tam = y/2 + 2;
        if(x >= tam){
            a = y-x;
            x += a/2;
            y -= a/2;
            if( a & 1 ) return ((x+1)/3)*2 +1;
            return (((x-1)/3)+1)*2;
        } else {
            a = y/2;
            if(y&1){
                if(a&1) st = a+2;nd = a+1;
                else st = a+1;nd = a+2;
            } else {
                if(a&1) st = a+1;nd = a;
                else st = a; nd = a+1;
            }
            if(x&1) return nd;
            return st;
        }
    }
}
```

Precision cout

```
double res = algo;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(3); cout << res << '\n';
```

Big Integer / Java

```
import java.util.Scanner;
import java.math.BigInteger;
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int caseNo = 1;
        while (true) {
            int N = sc.nextInt(), F = sc.nextInt();
            if (N == 0 && F == 0) break;
            BigInteger sum = BigInteger.ZERO
            for (int i = 0; i < N; i++) {
                BigInteger V = sc.nextBigInteger();
                sum = sum.add(V);
            }
            System.out.println(sum.divide(
                        BigInteger.valueOf(F)));
        }
    }
}
```

Constructor

```
BigInteger(String val) // Translates the decimal
String representation of a BigInteger into a
BigInteger.
```

Constantes

```
Static BigInteger ONE //The BigInteger constant
one.
Static BigInteger TEN //The BigInteger constant
ten.
Static BigInteger ZERO //The BigInteger constant
zero.
```

Métodos

- **BigInteger** abs() //Returns a **BigInteger** whose value is absolute value of this **BigInteger**.
- **BigInteger** add(**BigInteger** val) //Returns **BigInteger** whose value is (this + val).
- **BigInteger** compareTo(**BigInteger** val) //Compares this **BigInteger** with the specified **BigInteger**.
- **BigInteger** divide(**BigInteger** val) //Returns a **BigInteger** whose value is (this / val).
- **BigInteger[]** divideAndRemainder(**BigInteger** val) //Returns an array of two BigIntegers containing (this / val) followed by (this % val).
- **double** doubleValue() //Converts this **BigInteger** to a **double**.
- float floatValue() //Converts this **BigInteger** to a float
- **BigInteger** gcd(**BigInteger** val) //Returns a **BigInteger** whose value is the gratest common divisor of abs(this) and abs(val).
- **int** intValue() //Converts this **BigInteger** to an **int**.
- **long** longValue() //Converts this **BigInteger** to a **long**.
- **BigInteger** **max**(**BigInteger** val) //Returns the maximum of this **BigInteger** and val.
- **BigInteger** **min**(**BigInteger** val) //Returns the minimum of this **BigInteger** and val.
- **BigInteger** mod(**BigInteger** m) // Returns a **BigInteger** whose value is (this mod m).
- **BigInteger** modInverse (**BigInteger** m) //Returns a **BigInteger** whose value is ($this^{-1}$ mod m).
- **BigInteger** modPow(**BigInteger** exponent, **BigInteger** m) //Returns a **BigInteger** whose value is ($this^{exponent}$ mod m).
- **BigInteger** multiply (**BigInteger** val) //Returns a **BigInteger** whose value is (this * val).
- **BigInteger** negate() //Returns a **BigInteger** whose value is (-this)
- **BigInteger** nextProbablePrime() // Returns the **first** integer greater than this **BigInteger** that is probably prime.
- **BigInteger** **pow**(**int** exponent) //Returns a **BigInteger** whose value is($this^{exponent}$).
- **BigInteger** remainder(**BigInteger** val) // Returns a **BigInteger** whose value is (this % val).
- **BigInteger** substract(**BigInteger** val) //Returns a **BigInteger** whose value is (this - val).
- String toString() //Returns the decimal String representation of this **BigInteger**.
- Static **BigInteger** valueOf(**long** val) // Returns a **BigInteger** whose value is equal to that of the specified **long**.

Primos hasta el 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			

Códigos Rivas

Robbing Gringotts

```
//Meet in the middle
#include <bits/stdc++.h>
using namespace std;
const int INF = (1 << 30);
const int MAXN = 200;
int Elementos[MAXN][MAXN], X[MAXN];
int MP[MAXN];
char vis[MAXN];
```

```
int NN, MM, dif;
typedef pair<int, int> Arista;
// Emparejamiento de costo maximo en grafo
bipartito ponderado.
// Nodos con indice del 0 al n - 1. Recibe los
mismos parametros
// que MaxEmparejamientoBipartito. Utiliza
algoritmo hungaro.
// Cuidado: Se ocupa una matriz de costo entre
nodos.

int slack[MAXN];
int retorno[MAXN];
int etiqueta[MAXN];
int pareja[MAXN];
bool visitado[MAXN];
int costo[MAXN][MAXN];

long long EmparejaCostoMaxBipartito(
    const vector<int>& l, const vector<int>& r) {
    // Si l.size() != r.size() KABOOM!
    assert(l.size() == r.size());

    int n = l.size() + r.size();
    fill(pareja, pareja + n, -1);
    fill(etiqueta, etiqueta + n, 0);
    for (int i = 0; i < l.size(); ++i)
        for (int j = 0; j < r.size(); ++j)
            etiqueta[l[i]] = max(etiqueta[l[i]],
                costo[l[i]][r[j]]);

    for (int i = 0; i < l.size(); ++i) {
        for (int j = 0; j < r.size(); ++j)
            slack[r[j]] = -costo[l[i]][r[j]] +
                etiqueta[l[i]] + etiqueta[r[j]];
        fill(visitado, visitado + n, false);
        fill(retorno, retorno + n, l[i]);
        visitado[l[i]] = true;

        bool emparejado = false;
        for (int j = 0; !emparejado; ++j) {
            int t = 0; for (; t < r.size(); ++t) {
                if (visitado[r[t]]) continue;
                if (!slack[r[t]]) break;
            }
            if (t < r.size()) {
                visitado[t = r[t]] = true;
                if (pareja[t] == -1) {
                    emparejado = true;
                    for (int p; ; t = p) {
                        pareja[t] = retorno[t];
                        p = pareja[retorno[t]];
                        pareja[retorno[t]] = t;
                        if (retorno[t] == l[i])
break;
                    }
                } else {
                    visitado[t = pareja[t]] = true;
                    for (int k = 0; k < r.size();
++k) {
                        int new_slack = etiqueta[t]
+
                            etiqueta[r[k]] -
costo[t][r[k]];
                        if (new_slack <
slack[r[k]]) {
                            slack[r[k]] =
new_slack;
                            retorno[r[k]] = t;
                        }
                    }
                }
            } else {
                int d = INF;
                for (int k = 0; k < r.size(); ++k)
                    if (slack[r[k]]) d = min(d,
slack[r[k]]);
                for (int k = 0; k < l.size(); ++k)
                    if (visitado[l[k]])
etiqueta[l[k]] -= d;
                for (int k = 0; k < r.size(); ++k)
                    if (!visitado[r[k]])
slack[r[k]] -= d;
                    else etiqueta[r[k]] += d;
            }
        }
    }
    long long tot = 0;
```

```
    for (int i = 0; i < l.size(); ++i)
        tot += (long
long)costo[l[i]][pareja[l[i]]];
    return tot;
}


//Obtiene el peso del subconjunto de peso maximo
tal que es menor o igual a un peso maximo
determinado
//Elementos de valor unitario y paso variable
// N - 2 ^ (N/2)
//Meet in the middle
int  DPA[(1 << 16)], DPB[(1 << 16)];
void MITM(int N, int M) {
      //Subarreglo A [0, (N + 1) / 2]
      //Subarreglo B [((N + 1) / 2) + 1, N]
      set<int> CombinacionesA;
      for (int i = 0; i < M; ++i) {
          DPA[0] = DPB[0] = 0;
          int mid_i = X[i] >> 1;
          int temp = 1 << mid_i, sum;
          CombinacionesA.clear();
          for (int j = 0; j < mid_i; ++j) DPA[1 <<
j] = Elementos[i][j];
          for (int j = 0; j < X[i] - mid_i; ++j)
DPB[1 << j] = Elementos[i][mid_i + j];
          for (int j = 0; j < temp; ++j) {
                int ind = j & (-j);
                DPA[j] = DPA[j ^ ind] + DPA[ind];
                CombinacionesA.insert(DPA[j]);
          }
          int mid_j = (X[i] - mid_i);
          temp = 1 << mid_j;
          char vis[55];
          fill(vis, vis + N, 0);
          for (int j = 0; j < temp; ++j) {
                int ind = j & (-j);
                DPB[j] = DPB[j ^ ind] + DPB[ind];
                for (int k = 0; k < N; ++k)
                    if ((CombinacionesA.find(MP[k]
- DPB[j]) != CombinacionesA.end()) && (vis[k] ==
0)) {
                        costo[k][NN + i] = MP[k];
                        vis[k] = 1;
                    }
          }
      }
}

int main(){
      cin.tie(0);
      ios_base::sync_with_stdio(0);
      int T, N, M;

      cin >> T;
      for (int nc = 0; nc < T; ++nc) {
            cin >> N >> M;
            vector<int> left, right;
          NN = max(N, M);
          for (int i = 0; i < NN; i++) {
              left.push_back(i);
              right.push_back(i + NN);
          }

            for (int i = 0; i < N; ++i) cin >>
MP[i];
            for (int i = 0; i < M; ++i) {
                  cin >> X[i];
                  for (int j = 0; j < X[i]; ++j)
cin >> Elementos[i][j];
            }
            for (int i = 0; i < (NN + NN); ++i)
                for (int j = 0; j < (NN + NN); ++j)
costo[i][j] = 0;
            MITM(N, M);
            cout << EmparejaCostoMaxBipartito(left,
right) << '\n';
      }
      return 0;
}
```

Joutong Travels

```
//3329 - Compresion de grafo ponderado a subgrafo
ponderado con aristas pertecientes al camino mínimo
con Dijkstra
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 105;
const int INF = (1 << 30);
typedef pair<int, int> Arista;
vector<int> grafo[MAXN];
vector<Arista> grafo_peso[MAXN];
int dist[MAXN];

int cap[MAXN][MAXN];
int flujo[MAXN][MAXN];

void AgregarArista(int u, int v, int c){
    grafo[u].push_back(v);
    grafo[v].push_back(u);
    cap[u][v] += c; cap[v][u] += c;
    //flujo[v][u] += c; // Solo en dirigidas!
}

int FlujoBloqueante(int u, int t, int f) {
    if (u == t) return f; int fluido = 0;
    for (int i = 0; i < grafo[u].size(); ++i) {
        int v = grafo[u][i];
        if (dist[u] + 1 > dist[v]) continue;
        int fv = FlujoBloqueante(v, t,
            min(f - fluido, cap[u][v] -
flujo[u][v]));
        flujo[u][v] += fv; flujo[v][u] -= fv;
        fluido += fv; if (fluido == f) break;
    }
    return fluido;
}

int Dinic(int s, int t, int n) {
    int flujo_maximo = dist[t] = 0;
    while (dist[t] < INF) {
        fill(dist, dist + n, INF);
        queue<int> q; q.push(s); dist[s] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int i = 0; i < grafo[u].size();
++i) {
                int v = grafo[u][i];
                if (flujo[u][v] == cap[u][v] ||
                    dist[v] <= dist[u] + 1)
continue;
                dist[v] = dist[u] + 1, q.push(v);
            }
        }
        if (dist[t] < INF) flujo_maximo +=
            FlujoBloqueante(s, t, INF);
    }
    return flujo_maximo;
}

vector<int> Dijkstra(int o, int n) {
    vector<int> dista(n, INF);
    priority_queue<Arista, vector<Arista>,
                   greater<Arista> > pq;
    pq.push(Arista(0, o)); dista[o] = 0;
    while (!pq.empty()) {
        int u = pq.top().second;
        int p = pq.top().first; pq.pop();
        //if (dist[u] < p) continue;
        for (int i = 0; i < grafo_peso[u].size();
++i) {
            p = grafo_peso[u][i].first;
            int v = grafo_peso[u][i].second;
            if (dista[u] + p < dista[v]) {
                dista[v] = dista[u] + p;
                pq.push(Arista(dista[v], v));
            }
        }
    }
    return dista;
}
int main() {
    cin.tie(0);
    ios_base::sync_with_stdio(0);
    int N, M, A, Z, a, b, p, daz, dza;
    cin >> N >> M >> A >> Z;
    A--; Z--;
```

```
    for (int i = 0; i < M; ++i) {
        cin >> a >> b >> p;
        a--; b--;
        grafo_peso[a].push_back(Arista(p, b));
        grafo_peso[b].push_back(Arista(p, a));
    }
    vector<int> distAZ = Dijkstra(A, N);
    daz = distAZ[Z];
    vector<int> distZA = Dijkstra(Z, N);
    for (int i = 0; i < N; ++i) {
        int du = distAZ[i];
        for (int j = 0; j < grafo_peso[i].size();
++j) {
            int v = grafo_peso[i][j].second;
            int p = grafo_peso[i][j].first;
            if ((du + p) == (daz - distZA[v]))
AgregarArista(i, v, 1);
        }
    }
   /* cout << "Grafo original:\n";
    for (int i = 0; i < N; ++i) {
        cout << "NODO " << i + 1<< ": ";
        for (int j = 0; j < grafo_peso[i].size();
++j) {
            cout << "( " << grafo_peso[i][j].second
+ 1<< " | " << grafo_peso[i][j].first << " ) ";
        }
        cout << '\n';
    }
    cout << "Grafo reducido:\n";
    for (int i = 0; i < N; ++i) {
        cout << "NODO " << i + 1<< ": ";
        for (int j = 0; j < grafo[i].size(); ++j) {
            cout << grafo[i][j] + 1<< " ";
        }
        cout << '\n';
    }*/
    cout << Dinic(A, Z, N) << '\n';
    return 0;
}
```

LIS NON-DECREASING

```
// for reconstructing the answer just check the
last for in LIS()
//LIS NON-DECREASING <=
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef pair<int, int> pii;
typedef vector<pii> vpii;
int LIS (vi &v) {
      vpii best;
      vi dad(v.size(), -1);
      for (int i = 0; i < v.size(); i++) {
            pii item = pii(v[i], i);
            auto it = lower_bound(best.begin(),
best.end(), item);
            if (it == best.end()) {
                  dad[i] = ((best.size() == 0)? -1:
best.back().second);
                  best.push_back(item);
            } else {
                dad[i] = dad[it->second];
                  *it = item;
            }
      }
      int r, i = best.back().second;
      for (r = 0; i >= 0; ++r, i = dad[i]);
      return r;
}

int main() {
    cin.tie(0);
    ios_base::sync_with_stdio(0);
      int nc, N, a;
      //cin >> nc;
      //while (nc--) {
          cin >> N;
          vi v(N);
          for(int i = 0; i < N; i++) cin >> v[i];
        cout << LIS(v) << '\n';
    //}
      return 0;
}
```

LIS STRICTLY INCREASING

```
// for reconstructing the answer just check the
last for in LIS()
//LIS STRICTLY INCREASING
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef pair<int, int> pii;
typedef vector<pii> vpii;
int LIS (vi &v) {
      vpii best;
      vi dad(v.size(), -1);
      for (int i = 0; i < v.size(); i++) {
            pii item = pii(v[i], i);
            auto it = lower_bound(best.begin(),
best.end(), item);
            if (it == best.end()){
                if ((best.size()) && (item.first ==
best.back().first)) {
                    dad[i] =
dad[best.back().second];
                    best.back() = item;
                    continue;
                }
                  dad[i] = ((best.size() == 0)? -1:
best.back().second);
                  best.push_back(item);
            } else {
                auto itt = it;
                itt--;
                if ((it != best.begin()) &&
(item.first == (*itt).first)) {
                    dad[i] = dad[(*itt).second];
                    continue;
                }
                dad[i] = dad[it->second];
                  *it = item;
            }
      }
      int r, i = best.back().second;
      for (r = 0; i >= 0; ++r, i = dad[i]);
      return r;
}
```

Unique Path

```
//Componentes Biconexas en grafo bidireccional
#include <bits/stdc++.h>
using namespace std;

// Definiciones iniciales
typedef pair<int, int> Arista;
const int INF = 1 << 30;
const int MAXN = 100000;
vector<int> grafo[MAXN];
int numeracion;
int level[MAXN];
int parent[MAXN];
vector<int> puentes[MAXN];
int low[MAXN], num[MAXN];
set<Arista> Puentesp;
// Detecta los puentes y puntos de articulacion en
// un grafo bidireccional. Indices de 0 a n - 1.
void PuntosArtPuentes_(int u, int p) {
    //int hijos = 0;
    low[u] = num[u] = ++numeracion;
    for (int i = 0; i < grafo[u].size(); ++i) {
        int v = grafo[u][i];
        if (v == p) continue;
        if (!num[v]) {
            //++hijos;
            PuntosArtPuentes_(v, u);
            if (low[v] > num[u]) {
                puentes[u].push_back(v);
                puentes[v].push_back(u);
            }
            low[u] = min(low[u], low[v]);
            //punto_art[u] |= low[v] >= num[u];
        } else low[u] = min(low[u], num[v]);
    }
    //if (p == -1) punto_art[u] = hijos > 1;
}

void PuntosArtPuentes(int n) {
    numeracion = 0;
    fill(num, num + n, 0);
    fill(low, low + n, 0);
    //fill(punto_art, punto_art + n, false);
```

```
    for (int i = 0; i < n; ++i) puentes[i].clear();
    for (int i = 0; i < n; ++i)
        if (!num[i]) PuntosArtPuentes_(i, -1);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < puentes[i].size(); ++j)
            Puentesp.insert(Arista(i,
puentes[i][j]));
}
// Estructura de conjuntos disjuntos.
// Conjuntos indexados de 0 a n - 1.

struct UnionFind {
    int nconjuntos;
    vector<int> padre;
    vector<int> tamano;

    UnionFind(int n) : nconjuntos(n),
        padre(n), tamano(n, 1) {
        for(int i = 0; i < n; ++i)
            padre[i] = i;
    }

    int Encontrar(int u) {
        if (padre[u] == u) return u;
        return padre[u] = Encontrar(padre[u]);
    }

    void Unir(int u, int v) {
        int Ru = Encontrar(u);
        int Rv = Encontrar(v);
        if (Ru == Rv) return;
        -- nconjuntos, padre[Ru] = Rv;
        tamano[Rv] += tamano[Ru];
    }

    bool MismoConjunto(int u, int v) {
        return Encontrar(u) == Encontrar(v);
    }

    int TamanoConjunto(int u) {
        return tamano[Encontrar(u)];
    }
};
```

```
char TC[MAXN];
char PROC[MAXN];

void Limpia(int n){
    for (int i = 0; i < n; ++i) {
        grafo[i].clear();
        PROC[i] = 'N';
    }
    Puentesp.clear();
}

void Regiones(int n, UnionFind& UF) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < grafo[i].size(); ++j) {
            int v = grafo[i][j];
            if (Puentesp.find(Arista(i, v)) !=
Puentesp.end()) continue;
            UF.Unir(i, v);
        }
}
int main(){
    cin.tie(0);
    ios_base::sync_with_stdio(0);
    int T;
    cin >> T;
    for (int nc = 1; nc <= T; ++nc) {
        int N, M, a, b;
        cin >> N >> M;
        Limpia(N);
        for (int i = 0; i < M; ++i) {
            cin >> a >> b;
            a--; b--;
            grafo[a].push_back(b);
            grafo[b].push_back(a);
        }
        PuntosArtPuentes(N);
        UnionFind UFC(N);
        Regiones(N, UFC);
        for (int i = 0; i < N; ++i)
            if (UFC.TamanoConjunto(i) > 1) TC[i] =
'C';
            else TC[i] = 'S';
        UnionFind UF(N);
        for (int i = 0; i < N; ++i) {
```

```
            for (int j = 0; j < puentes[i].size();
++j) {
                int v = puentes[i][j];
                if ((TC[i] == 'C') || (TC[v] ==
'C')) continue;
                //cout << "Puente entre: " << i +1
<< " y " << v + 1 << '\n';
                UF.Unir(i, v);
            }
        }
        long long ans = 0, aux;
        //proc N = NOT DOMN, D = DONE;
        for (int i = 0; i < N; ++i) {
            int pi = UF.Encontrar(i);
            if ((TC[i] == 'S') && (PROC[pi] ==
'N')) {
                aux = UF.TamanoConjunto(i);
                ans += ((aux * (aux - 1)) / 2);
                PROC[pi] = 'D';
            }
        }
        cout << "Case #" << nc << ": " << ans <<
'\n';
    }
    return 0;
}
```

File Transmission

```
//Compresion de grafo bidireccional a arbol
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 10005;
const int LOGN = 18;
vector<int> grafo[MAXN];
typedef pair<int, int> Arista;
char vis[MAXN];
int numeracion;
int level[MAXN];
int parent[MAXN];
vector<int> puentes[MAXN];
int low[MAXN], num[MAXN], P[MAXN][LOGN];
set<Arista> Puentesp;
// Detecta los puentes y puntos de articulacion en
// un grafo bidireccional. Indices de 0 a n - 1.

void PuntosArtPuentes_(int u, int p) {
    //int hijos = 0;
    low[u] = num[u] = ++numeracion;
    for (int i = 0; i < grafo[u].size(); ++i) {
        int v = grafo[u][i];
        if (v == p) continue;
        if (!num[v]) {
            //++hijos;
            PuntosArtPuentes_(v, u);
            if (low[v] > num[u]) {
                puentes[u].push_back(v);
                puentes[v].push_back(u);
            }
            low[u] = min(low[u], low[v]);
            //punto_art[u] |= low[v] >= num[u];
        } else low[u] = min(low[u], num[v]);
    }
    //if (p == -1) punto_art[u] = hijos > 1;
}

void PuntosArtPuentes(int n) {
    numeracion = 0;
    fill(num, num + n, 0);
    fill(low, low + n, 0);
```

```
    //fill(punto_art, punto_art + n, false);
    for (int i = 0; i < n; ++i) puentes[i].clear();
    for (int i = 0; i < n; ++i)
        if (!num[i]) PuntosArtPuentes_(i, -1);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < puentes[i].size(); ++j)
            Puentesp.insert(Arista(i,
puentes[i][j]));
}

// Estructura de conjuntos disjuntos.
// Conjuntos indexados de 0 a n - 1.

struct UnionFind {
    int nconjuntos;
    vector<int> padre;
    vector<int> tamano;

    UnionFind(int n) : nconjuntos(n),
        padre(n), tamano(n, 1) {
        for(int i = 0; i < n; ++i)
            padre[i] = i;
    }

    int Encontrar(int u) {
        if (padre[u] == u) return u;
        return padre[u] = Encontrar(padre[u]);
    }

    void Unir(int u, int v) {
        int Ru = Encontrar(u);
        int Rv = Encontrar(v);
        if (Ru == Rv) return;
        -- nconjuntos, padre[Ru] = Rv;
        tamano[Rv] += tamano[Ru];
    }

    bool MismoConjunto(int u, int v) {
        return Encontrar(u) == Encontrar(v);
    }

    int TamanoConjunto(int u) {
        return tamano[Encontrar(u)];
    }
};


void PRE(int n) {
    for (int d = 1; d < LOGN; ++d)
        for (int i = 0; i < n; ++i)
            P[i][d] = P[P[i][d - 1]][d - 1];
}

int lca(int u, int v) {
    if (level[u] > level[v]) swap(u, v);
    for (int i = 0, j = level[v] - level[u]; i <
LOGN && j > 0; ++i, j >>= 1)
        if (j & 1) v = P[v][i];
    if (u == v) return u;
    for (int i = LOGN - 1; i >= 0; --i)
        if (P[u][i] != P[v][i]) {
            u = P[u][i];
            v = P[v][i];
        }
    return P[u][0];
}

void Limpia(int n) {
    for (int d = 0; d < LOGN; ++d)
        for (int i = 0; i < n; ++i) P[i][d] = 0;
    for (int i = 0; i < n; ++i) {
        vis[i] = 0;
        grafo[i].clear();
        P[i][0] = i;
        level[i] = 1;
    }
    Puentesp.clear();
}

void Regiones(int n, UnionFind& UF) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < grafo[i].size(); ++j) {
            int v = grafo[i][j];
            if (Puentesp.find(Arista(i, v)) !=
Puentesp.end()) continue;
            UF.Unir(i, v);
        }
}
```

```
void Enraizar(int o, UnionFind UF) {
    queue<int> Q;
    Q.push(o);
    vis[o] = 1;
    while (!Q.empty()) {
        int u = Q.front();
        int pu = UF.Encontrar(u);
        //cout << "sale " << u << '\n';
        Q.pop();
        for (int j = 0 ; j < grafo[u].size(); ++j)
{
            int v = grafo[u][j];
            int pv = UF.Encontrar(v);
            if (vis[v] == 1) continue;
            if (pv != pu) {
                level[pv] = level[pu] + 1;
                P[pv][0] = pu;
            }
            vis[v] = 1;
            Q.push(v);
       }
    }
}
int main(){
    cin.tie(0);
    ios_base::sync_with_stdio(0);
    int N, M;
    for (int nc = 1; ; ++nc) {
        cin >> N >> M;
        if (N + M == 0) break;
        Limpia(N);
        int a, b;
        for (int i = 0; i < M; ++i) {
            cin >> a >> b;
            a--; b--;
            grafo[a].push_back(b);
            grafo[b].push_back(a);
        }
        PuntosArtPuentes(N);
        UnionFind UF(N);
        Regiones(N, UF);
        //for (int i = 0; i < N; ++i) cout << "NODO
" << i + 1 << " REGION: " << low[i] << '\n';
        /*cout << "GRAFOOO!!\n";
        for (int i = 0; i < N; ++i) {
            cout << "NODO " << i + 1 << ": ";
            for (int j = 0; j < grafo[i].size();
++j) cout << grafo[i][j] + 1<< " ";
            cout << '\n';
        }*/
        //for (int i = 0; i < N; ++i) UF.Unir(i,
low[i] - 1);
        //for (int i = 0; i < N; ++i) cout <<
"Padre - Region de " << i + 1 << ": " <<
UF.Encontrar(i) + 1 << '\n';
        Enraizar(0, UF);
        /*for (int i = 0; i < N; ++i) {
            for (int j = 0 ; j < grafo[i].size();
++j) {
                int v = grafo[i][j];
                int pu = UF.Encontrar(i);
                int pv = UF.Encontrar(v);
                if ((pu == pv) || (vis[pv] == 1))
continue;
                level[pv] += level[pu];
                P[pv][0] = pu;
                vis[pv] = 1;
                vis[pu] = 1;
            }
        }*/
        //for (int i = 0; i < N; ++i) cout <<
"level de " << i + 1 << " cuyo padre es: " <<
UF.Encontrar(i) + 1 << " = " <<
level[UF.Encontrar(i)] << '\n';
        PRE(N); int Q; cin >> Q;
        if (nc > 1) cout << '\n';
        cout << "Case #" << nc << ":\n";
        for (int i = 0; i < Q; ++i) {
            cin >> a >> b;
            a--; b--;
            int pa = UF.Encontrar(a);
            int pb = UF.Encontrar(b);
            long long ans = (level[pa] + level[pb]
- ((level[lca(pa, pb)]) << 1)) * 50LL ;
            cout <<  ans << '\n';
        }
    }
    return 0;  }
```