Notebook - Maratonas de Programação

Gabriel Moretti

Contents

1 Geometria

1.1 Circulos

```
#define PI 2*acos(0.0)

// A unidade de medida do angulo é radianos
template<typename T>
struct Circle {
    Point<T> C;
    T r;

    enum { IN, ON, OUT } PointPosition;

    double perimeter() const {
        return 2.0 * PI * r;
    }

    double area() const {
        return PI * r * r;
    }

    double arc(double theta) const {
        return theta * r;
    }

    double sector(double theta) const {
        return (theta * r * r)/2;
    }

    double chord(double theta) const {
        return 2 * r * sin(theta/2);
    }

    double segment(double a) const {
        return ((a - sin(a))*r*r)/2.0;
    }

    PointPosition position(const Point& P) const
    {
        auto d = dist(P, C);

        return equals(d, r) ? ON : (d < r ? IN : OUT);
    }


    // O ócdigo abaixo foi adaptado do livro Competitive Programming 3.  A
    çãfuno retorna um dos
    // ícrculos: o outro pode ser encontrado invertendo os âparmetros P e Q na
    chamada da çãfuno

    static std::optional<Circle>
    from_2_points_and_r(const Point<T>& P, const Point<T>& Q, T r)
    {
        double d2 = (P.x - Q.x) * (P.x - Q.x) + (P.y - Q.y) * (P.y - Q.y);
        double det = r * r / d2 - 0.25;
        if (det < 0.0)
            return { };
        double h = sqrt(det);
        auto x = (P.x + Q.x) * 0.5 + (P.y - Q.y) * h;
        auto y = (P.y + Q.y) * 0.5 + (Q.x - P.x) * h;
        return Circle<T>{ Point<T>(x, y), r };
    }

    static std::experimental::optional<Circle>
    from_3_points(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
    {
        auto a = 2*(Q.x - P.x);
        auto b = 2*(Q.y - P.y);
        auto c = 2*(R.x - P.x);
        auto d = 2*(R.y - P.y);
        auto det = a*d - b*c;
        // Pontos colineares
        if (equals(det, 0))
            return { };
        auto k1 = (Q.x*Q.x + Q.y*Q.y) - (P.x*P.x + P.y*P.y);
        auto k2 = (R.x*R.x + R.y*R.y) - (P.x*P.x + P.y*P.y);
        // çãSoluo do sistema por Regra de Cramer
        auto cx = (k1*d - k2*b)/det;
        auto cy = (a*k2 - c*k1)/det;
        Point<T> C { cx, cy };
        auto r = distance(P, C);
        return Circle<T>(C, r);
    }
};

template<typename T> std::variant<int, std::vector<Point<T>>>
intersection(const Circle<T>& c1, const Circle<T>& c2)
{
    double d = distance(c1.C, c2.C);
    if (d > c1.r + c2.r or d < fabs(c1.r - c2.r)) return 0;
    if (equals(d, 0.0) and equals(c1.r, c2.r)) return oo;
    auto a = (c1.r * c1.r - c2.r * c2.r + d * d)/(2 * d);
    auto h = sqrt(c1.r * c1.r - a * a);
    auto x = c1.C.x + (a/d)*(c2.C.x - c1.C.x);
    auto y = c1.C.y + (a/d)*(c2.C.y - c1.C.y);
    auto P = Point<T> { x, y };
    x = P.x + (h/d)*(c2.C.y - c1.C.y);
    y = P.y - (h/d)*(c2.C.x - c1.C.x);
    auto P1 = Point<T> { x, y };
    x = P.x - (h/d)*(c2.C.y - c1.C.y);
    y = P.y + (h/d)*(c2.C.x - c1.C.x);
    auto P2 = Point<T> { x, y };
    return P1 == P2 ? std::vector<Point<T>> { P1 } : std::vector<Point<T>> {
    P1, P2 };
}

// çãInterseo entre o ícrculo c e a reta que passa por P e Q
template<typename T> std::vector<Point<T>>
intersection(const Circle<T>& c, const Point<T>& P, const Point<T>& Q)
{
    auto a = pow(Q.x - P.x, 2.0) + pow(Q.y - P.y, 2.0);
    auto b = 2*((Q.x - P.x) * (P.x - c.C.x) + (Q.y - P.y) * (P.y - c.C.y));
    auto d = pow(c.C.x, 2.0) + pow(c.C.y, 2.0) + pow(P.x, 2.0)
        + pow(P.y, 2.0) + 2*(c.C.x * P.x + c.C.y * P.y);
```

```
    auto D = b * b - 4 * a * d;
    if (D < 0)
        return { };
    else if (equals(D, 0))
    {
        auto u = -b/(2*a);
        auto x = P.x + u*(Q.x - P.x);
        auto y = P.y + u*(Q.y - P.y);
        return { Point { x, y } };
    }
    auto u = (-b + sqrt(D))/(2*a);
    auto x = P.x + u*(Q.x - P.x);
    auto y = P.y + u*(Q.y - P.y);
    auto P1 = Point { x, y };
    u = (-b - sqrt(D))/(2*a);
    x = P.x + u*(Q.x - P.x);
    y = P.y + u*(Q.y - P.y);
    auto P2 = Point { x, y };
    return { P1, P2 };
}
```

1.2 Sweep Line

```
vector<int> max_intersection(const vector<pll>& is)
{
    vector<pll> es;

    for (size_t i = 0; i < is.size(); ++i)
    {
        auto [a, b] = is[i];

        es.emplace_back(a, i + 1);       // Evento de íincio
        es.emplace_back(b, -(i + 1));    // Evento de fim
    }

    sort(es.begin(), es.end());

    set<int> active, max_set;

    for (const auto& [_, i] : es)
    {
        if (i > 0)
            active.emplace(i);
        else
            active.erase(-i);

        if (active.size() >= max_set.size())
            max_set = active;
    }

    return { max_set.begin(), max_set.end() };
}

// bf

template<typename T>
set<Point<T>> intersections(int N, const vector&lt;Segment&lt;T&gt;&gt;& segments)
{
    set<Point<T>> ans;

    for (int i = 0; i < N; ++i)
    {
        auto s = segments[i];

        for (int j = i + 1; j < N; ++j)
        {
            auto r = segments[j];
            auto P = s.intersection(r);

            if (P) ans.insert(P.value());
        }
    }

    return ans;
}

// Closest

pair<Point, Point> closest_pair(int N, vector<Point>& ps)
{
    sort(ps.begin(), ps.end());

    // Este ócdigo assume que N > 1
    auto d = dist(ps[0], ps[1]);
    auto closest = make_pair(ps[0], ps[1]);

    set<ii> S;
    S.insert(ii(ps[0].y, ps[0].x));
    S.insert(ii(ps[1].y, ps[1].x));

    for (int i = 2; i < N; ++i)
    {
        auto P = ps[i];
        auto it = S.lower_bound(Point(P.y - d, 0));

        while (it != S.end())
        {
            auto Q = Point(it->second, it->first);

            if (Q.x < P.x - d)
            {
                it = S.erase(it);
                continue;
            }

            if (Q.y > P.y + d)
                break;

            auto t = dist(P, Q);

            if (t < d)
            {
                d = t;
                closest = make_pair(P, Q);
            }
```

```
            ++it;
        }

        S.insert(ii(P.y, P.x));
    }

    return closest;
}

// Bentley Ottman

void add_neighbor_intersections(const Segment& s, const set&lt;Segment&gt;& sl,
    set<Point>& ans, priority_queue<Event>& events)
{
    // TODO: garantir que a busca identifique unicamente o elemento s,
    // éatravs do ajuste fino da ávarivel Segment::sweep_x
    auto it = sl.find(s);

    if (it != sl.begin())
    {
        auto L = *prev(it);
        auto P = s.intersection(L);

        if (P and ans.count(P.value()) == 0)
        {
            events.push(Event { P.value(), Event::INTERSECTION, s.idx } );
            ans.insert(P.value());
        }
    }

    if (next(it) != sl.end())
    {
        auto U = *next(it);
        auto P = s.intersection(U);

        if (P and ans.count(P.value()) == 0)
        {
            events.push(Event { P.value(), Event::INTERSECTION, s.idx } );
            ans.insert(P.value());
        }
    }
}

set<Point> bentley_ottman(vector&lt;Segment&gt;& segments)
{
    set<Point> ans;
    priority_queue<Event> events;

    for (size_t i = 0; i < segments.size(); ++i)
    {
        events.push(Event { segments[i].A, Event::OPEN, i });
        events.push(Event { segments[i].B, Event::CLOSE, i });
    }

    set&lt;Segment&gt; sl;

    while (not events.empty())
    {
        auto e = events.top();
        events.pop();

        Segment::sweep_x = e.P.x;

        switch (e.type) {
        case Event::OPEN:
        {
            auto s = segments[e.i];
            sl.insert(s);

            add_neighbor_intersections(s, sl, ans, events);
        }
        break;

        case Event::CLOSE:
        {
            auto s = segments[e.i];
            auto it = sl.find(s);       // TODO: aqui étambm

            if (it != sl.begin() and it != sl.end())
            {
                auto L = *prev(it);
                auto U = *next(it);
                auto P = L.intersection(U);

                if (P and ans.count(P.value()) == 0)
                    events.push( Event { P.value(), Event::INTERSECTION, L.idx
    } );
            }

            sl.erase(it);
        }
        break;

        default:
            auto r = segments[e.i];
            auto p = sl.equal_range(r);

            vector&lt;Segment&gt; range(p.first, p.second);

            // Remove os segmentos que se interceptam
            sl.erase(p.first, p.second);

            // Reinsere os segmentos
            Segment::sweep_x += 0.1;

            sl.insert(range.begin(), range.end());

            // Procura çõintersees com os novos vizinhos
            for (const auto& s : range)
                add_neighbor_intersections(s, sl, ans, events);
        }
    }

    return ans;
}
```

1.3 Reta Dois

```cpp
template<typename T>
struct Line {
    bool vertical;
    T m, b;

    Line(const Point<T>& P, const Point<T>& Q) : vertical(false)
    {
        if (equals(P.x, Q.x))
        {
            vertical = true;
            b = P.x;
        } else
        {
            m = (Q.y - P.y)/(Q.x - P.x)
            b = P.y - m * P.x
        }
    }

    bool operator==(const Line<T>& r) const    // Verdadeiro se coincidentes
    {
        if (vertical != r.vertical || !equals(m, r.m)) return false;

        return equals(b, r.b);
    }

    bool parallel(const Line<T>& r) const      // Verdadeiro se paralelas
    {
        if (vertical && r.vertical) return b != r.b;
        if (vertical || r.vertical) return false;

        return equals(m, r.m) && !equals(b, r.b);
    }

    bool orthogonal(const Line& r) const // Verdadeiro se perpendiculares
    {
        if (vertical && r.vertical)
            return false;

        if ((vertical && equals(r.m, 0)) || (equals(m, 0) && r.vertical))
            return true;

        if (vertical || r.vertical)
            return false;

        return equals(m * r.m, -1.0);
    }
};
```

1.4 Retas

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

template<typename T>
struct Point {
    T x = 0, y = 0;
};

template<typename T>
struct Line {
    T a, b, c;

    Line(const Point<T>& P, const Point<T>& Q)
        : a(P.y - Q.y), b (Q.x - P.x), c(P.x * Q.y - Q.x * P.y)
    {
    }

    bool contains(const Point<T>& P) const
    {
        return equals(a*P.x + b*P.y + c, 0);
    }

    bool operator==(const Line<T>& r) const
    {
        auto k = a ? a : b;
        auto s = r.a ? r.a : r.b;

        return equals(a*s, r.a*k) && equals(b*s, r.b*k) && equals(c*s, r.c*k);
    }

    bool parallel(const Line<T>& r) const
    {
        auto det = a*r.b - b*r.a;

        return det == 0 and !(*this == r);
    }

    double distance(const Point<T>& p) const          // âDistncia de p à reta
    {
        return fabs(a*p.x + b*p.y + c)/hypot(a, b);
    }

    Point<T> closest(const Point<T>& p) const         // Ponto mais óprximo de p
    {
        auto den = (a*a + b*b);

        auto x = (b*(b*p.x - a*p.y) - a*c)/den;
        auto y = (a*(-b*p.x + a*p.y) - b*c)/den;

        return Point<T> { x, y };
    }

    bool orthogonal(const Line& r) const   // Verdadeiro se perpendiculares
    {
        return equals(a * r.a + b * r.b, 0);
    }
};

template<typename T>
T absolute_value(T x)
{
    if constexpr (std::is_floating_point_v<T>)
        return fabs(x);
    else
        return llabs(static_cast<long long>(x));
```

```
}

template<typename T>
double dist(const Point<T>& P, const Point<T>& Q) {
    return hypot(static_cast<double>(P.x - Q.x), static_cast<double>(P.y - Q.y
    ));
}

template<typename T>
T dist2(const Point<T>& P, const Point<T>& Q) {
    return (P.x - Q.x)*(P.x - Q.x) + (P.y - Q.y)*(P.y - Q.y);
}

template<typename T>
T taxicab(const Point<T>& P, const Point<T>& Q) {
    return absolute_value(P.x - Q.x) + absolute_value(P.y - Q.y);
}

template<typename T>
T max_norm(const Point<T>& P, const Point<T>& Q) {
    return std::max(absolute_value(P.x - Q.x), absolute_value(P.y - Q.y));
}

int main()
{
    Point<int> P, Q { 2, 3 };

    std::cout << "Euclidiana: " << dist(P, Q) << '\n';
    std::cout << "Quadrado: " << dist2(P, Q) << '\n';
    std::cout << "Motorista de átxi: " << taxicab(P, Q) << '\n';
    std::cout << "Norma do ámximo: " << max_norm(P, Q) << '\n';

    return 0;
}
```

1.5 Triangulos

```
template<typename T>
struct Triangle {
    Point<T> A, B, C;

    enum Angles { RIGHT, ACUTE, OBTUSE };
    enum Sides { EQUILATERAL, ISOSCELES, SCALENE };

    double perimeter() const
    {
        auto a = dist(A, B), b = dist(B, C), c = dist(C, A);

        return a + b + c;
    }

    double area() const
    {
        Line<T> r(A, B);

        auto b = dist(A, B);
        auto h = r.distance(C);
        return (b * h)/2;
    }

    double area2() const
    {
        auto a = dist(A, B);
        auto b = dist(B, C);
        auto c = dist(C, A);
        auto s = (a + b + c)/2
        return sqrt(s)*sqrt(s - a)*sqrt(s - b)*sqrt(s - c);
    }

    double area3() const
    {
        double det = (A.x*B.y + A.y*C.x + B.x*C.y) - (C.x*B.y + C.y*A.x + B.x*
A.y);

        return 0.5 * fabs(det);
    }

    Point<T> barycenter() const
    {
        auto x = (A.x + B.x + C.x) / 3.0;
        auto y = (A.y + B.y + C.y) / 3.0;

        return Point<T> { x, y };
    }

    double inradius() const
    {
        return (2 * area()) / perimeter();
    }

    Point<double> incenter() const
    {
        auto a = dist(B, C), b = dist(A, C), c = dist(A, B);
        auto P = perimeter();
        auto x = (a*A.x + b*B.x + c*C.x)/P;
        auto y = (a*A.y + b*B.y + c*C.y)/P;

        return { x, y };
    }

    double circumradius() const
    {
        auto a = dist(B, C);
        auto b = dist(A, C);
        auto c = dist(A, B);

        return (a * b * c)/(4 * area());
    }

    Point<T> circumcenter() const
    {
        auto D = 2*(A.x*(B.y - C.y) + B.x*(C.y - A.y) + C.x*(A.y - B.y));

        auto A2 = A.x*A.x + A.y*A.y;
        auto B2 = B.x*B.x + B.y*B.y;
```

```
        auto C2 = C.x*C.x + C.y*C.y;

        auto x = (A2*(B.y - C.y) + B2*(C.y - A.y) + C2*(A.y - B.y))/D;
        auto y = (A2*(C.x - B.x) + B2*(A.x - C.x) + C2*(B.x - A.x))/D;

        return { x, y };
    }

    Point<T> orthocenter() const
    {
        Line<T> r(A, B), s(A, C);

        Line<T> u { r.b, -r.a, -(C.x*r.b - C.y*r.a) };
        Line<T> v { s.b, -s.a, -(B.x*s.b - B.y*s.a) };

        auto det = u.a * v.b - u.b * v.a;
        auto x = (-u.c * v.b + v.c * u.b) / det;
        auto y = (-v.c * u.a + u.c * v.a) / det;

        return { x, y };
    }

    Angles classification_by_angles() const
    {
        auto a = dist(A, B);
        auto b = dist(B, C);
        auto c = dist(C, A);

        auto alpha = acos((a*a - b*b - c*c)/(-2*b*c));
        auto beta = acos((b*b - a*a - c*c)/(-2*a*c));
        auto gamma = acos((c*c - a*a - b*b)/(-2*a*b));

        auto right = PI / 2.0;

        if (equals(alpha, right) || equals(beta, right) || equals(gamma, right
    ))
            return RIGHT;

        if (alpha > right || beta > right || gamma > right)
            return OBTUSE;

        return ACUTE;
    }

    Sides classification_by_sides() const
    {
        auto a = dist(A, B), b = dist(B, C), c = dist(C, A);

        if (equals(a, b) and equals(b, c))
            return EQUILATERAL;

        if (equals(a, b) or equals(a, c) or equals(b, c))
            return ISOSCELES;

        return SCALENE;
    }
};
```

1.6 Retas Algoritmos

```
// Ângulo entre os segmentos de reta PQ e RS
template<typename T>
double angle(const Point<T>& P, const Point<T>& Q, const Point<T>& R, const
    Point<T>& S)
{
    auto ux = P.x - Q.x;
    auto uy = P.y - Q.y;

    auto vx = R.x - S.x;
    auto vy = R.y - S.y;

    auto num = ux * vx + uy * vy;
    auto den = hypot(ux, uy) * hypot(vx, vy);
    // Caso especial: se den == 0, algum dos vetores é degenerado: os dois
    // pontos ãso iguais. Neste caso, o ângulo ãno áest definido
    return acos(num / den);
}

template<typename T>
Line<T> perpendicular_bisector(const Point<T>& P, const  Point<T>& Q)
{
    auto a = 2*(Q.x - P.x);
    auto b = 2*(Q.y - P.y);
    auto c = (P.x * P.x + P.y * P.y) - (Q.x * Q.x + Q.y * Q.y);
    return { a, b, c };
}

struct Segment {
    Point<T> A, B;
    // Verifica se o ponto P da reta r que écontm A e B  pertence ao segmento
    bool contains(const Point<T>& P) const {
        return equals(A.x, B.x) ?  min(A.y, B.y) <= P.y and P.y <= max(A.y, B.
    y)
               : min(A.x, B.x) <= P.x and P.x <= max(A.x, B.x);
    }
    // Esta abordagem ãno exige que P esteja sobre a reta AB
    bool contains2(const Point<T>& P) const {
        double dAB = dist(A, B), dAP = dist(A, P), dPB = dist(P, B);
        return equals(dAP + dPB, dAB);
    }
    // Ponto mais óprximo de P no segmento AB
    Point<T> closest(const Point<T>& P) {
        Line<T> r(A, B);
        auto Q = r.closest(P);
        if (this->contains(Q)) return Q;
        auto distA = P.distanceTo(A);
        auto distB = P.distanceTo(B);
        if (distA <= distB) return A;
        else return B;
    }

    bool intersect(const Segment<T>& s) const
    {
        auto d1 = D(A, B, s.A);
        auto d2 = D(A, B, s.B);
        if ((equals(d1, 0) && contains(s.A)) || (equals(d2, 0) && contains(s.B
```

```
    ))) return true;
        auto d3 = D(s.A, s.B, A);
        auto d4 = D(s.A, s.B, B);
        if ((equals(d3, 0) && s.contains(A)) || (equals(d4, 0) && s.contains(B
    ))) return true;
        return (d1 * d2 < 0) && (d3 * d4 < 0);
    }
}

// Verifica se o ponto P pertence ao segmento de reta AB
template<typename T>
bool contains(const Point<T>& A, const Point<T>& B, const Point<T>& P)
{
    // Verifica se P áest na ãregio retangular
    auto xmin = min(A.x, B.x);
    auto xmax = max(A.x, B.x);
    auto ymin = min(A.y, B.y);
    auto ymax = max(A.y, B.y);
    if (P.x < xmin || P.x > xmax || P.y < ymin || P.y > ymax)
        return false;
    // Verifica çãrelao de çsemelhana no âtringulo
    return equals((P.y - A.y)*(B.x - A.x), (P.x - A.x)*(B.y - A.y));
}

// D = 0: R pertence a reta PQ
// D > 0: R à esquerda da reta PQ
// D < 0: R à direita da reta PQ
template<typename T>
T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
{
    return (P.x * Q.y + P.y * R.x + Q.x * R.y) - (R.x * Q.y + R.y * P.x + Q.x
    * P.y);
}

template<typename T>
std::pair<int, Point<T>> intersections(const Line<T>& r, const Line<T>& s)
{
    auto det = r.a * s.b - r.b * s.a;
    if (equals(det, 0))      // Coincidentes ou paralelas
    { return { (r == s) ? oo : 0, {} };
    }else                      // Concorrentes
    {
        auto x = (-r.c * s.b + s.c * r.b) / det;
        auto y = (-s.c * r.a + r.c * s.a) / det;
        return { 1, { x, y } };
    }
}
```

1.7 Vetores

```
template<typename T>
struct Vector
{
    T x = 0, y = 0;

    Vector(const Point<T>& A, const Point<T>& B)
        : x(B.x - A.x), y(B.y - A.y) {}
};

template<typename T>
Vector<T> normalize(const Vector<T>& v)
{
    auto len = v.length();
    return { v.x / len, v.y / len };
}

template<typename T>
Point<T> rotate(const Point<T>& P, T angle)
{
    auto x = cos(angle) * P.x - sin(angle) * P.y;
    auto y = sin(angle) * P.x + cos(angle) * P.y;

    return { x, y };
}

template<typename T>
Point<T> rotate2(const Point<T>& P, T angle, const Point<T>& C)
{
    auto Q = translate(P, -C.x, -C.y);
    Q = rotate(Q, angle);
    Q = translate(Q, C.x, C.y);

    return Q;
}

template<typename T>
Vector<T> scale(const Vector<T>& v, T sx, T sy)
{
    return { sx * v.x, sy * v.y };
}

template<typename T>
Point<T> translate(const Point<T>& P, T dx, T dy)
{
    return { P.x + dx, P.y + dy };
}

template<typename T>
Vector<T> cross_product(const Vector<T>& u, const Vector<T>& v)
{
    auto x = u.y*v.z - v.y*u.z;
    auto y = u.z*v.x - u.x*v.z;
    auto z = u.x*v.y - u.y*v.x;

    return { x, y, z };
}

template<typename T>
T dot_product(const Vector<T>& u, const Vector<T>& v)
{
    return u.x * v.x + u.y * v.y;
}

// O retorno áest no intervalo [0, pi]
template<typename T>
double angle(const Vector<T>& u, const Vector<T>& v)
```

```
{
    auto lu = u.length();
    auto lv = v.length();
    auto prod = dot_product(u, v);

    return acos(prod/(lu * lv));
}
```

1.8 Quadrilateros

```
struct Rectangle {
    Point<T> P, Q;
    T b, h;

    Rectangle(const Point<T>& p, const Point<T>& q) : P(p), Q(q)
    {
        b = max(P.x, Q.x) - min(P.x, Q.x);
        h = max(P.y, Q.y) - min(P.y, Q.y);
    }

    Rectangle(const T& base, const T& height)
        : P(0, 0), Q(base, height), b(base), h(height) {}

    T perimeter() const
    {
        return 2 * (b + h);
    }

    T area() const
    {
        return b * h;
    }

    Rectangle intersection(const Rectangle& r) const
    {
        using interval = pair<T, T>;

        auto I = interval(min(P.x, Q.x), max(P.x, Q.x));
        auto U = interval(min(r.P.x, r.Q.x), max(r.P.x, r.Q.x));

        auto a = max(I.first, U.first);
        auto b = min(I.second, U.second);

        if (b < a)
            return { {-1, -1}, {-1, -1} };

        I = interval(min(P.y, Q.y), may(P.y, Q.y));
        U = interval(min(r.P.y, r.Q.y), may(r.P.y, r.Q.y));

        auto c = max(I.first, U.first);
        auto d = min(I.second, U.second);

        if (d < c)
            return { {-1, -1}, {-1, -1} };

        inter = Rectangle(Point(a, c), Point(b, d));

        return { {a, c}, {b, d} };
    }
};

template<typename T>
struct Trapezium {
    T b, B, h;

    T area() const
    {
        return (b + B) * h / 2;
    }
};
```

1.9 Poligonos

```
template<typename T>
class Polygon {
    vector<Point<T>> vs;
    int n;

    // O âparmetro deve conter os n évrtices do ípolgono
    Polygon(const vector<Point<T>>& ps) : vs(ps), n(vs.size())
    {
        vs.push_back(vs.front());
    }

    T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R) const
    {
        return (P.x * Q.y + P.y * R.x + Q.x * R.y) - (R.x * Q.y + R.y * P.x +
    Q.x * P.y);
    }

    bool convex() const {
        // Um ípolgono deve ter, no minimo, 3 évrtices
        if (n < 3) return false;

        int P = 0, N = 0, Z = 0;

        for (int i = 0; i < n; ++i) {
            auto d = D(vs[i], vs[(i + 1) % n], vs[(i + 2) % n]);
            d ? (d > 0 ? ++P : ++N) : ++Z;
        }

        return P == n or N == n;
    }

    double distance(const Point<T>&P, const Point<T>& Q)
    {
        return hypot(P.x - Q.x, P.y - Q.y);
    }

    double perimeter() const
    {
        auto p = 0.0;
        for (int i = 0; i < n; ++i)
            p += distance(vs[i], vs[i + 1]);
        return p;
    }
```

```cpp
double area() const
{
    auto a = 0.0;
    for (int i = 0; i < n; ++i)
    {
        a += vs[i].x * vs[i + 1].y;
        a -= vs[i + 1].x * vs[i].y;
    }
    return 0.5 * fabs(a);
}

// Ângulo APB, em radianos
double angle(const Point<T>& P, const Point<T>& A, const Point<T>& B)
{
    auto ux = P.x - A.x;
    auto uy = P.y - A.y;
    auto vx = P.x - B.x;
    auto vy = P.y - B.y;
    auto num = ux * vx + uy * vy;
    auto den = hypot(ux, uy) * hypot(vx, vy);
    // Caso especial: se den == 0, algum dos vetores é degenerado: os
    // dois pontos ãso iguais. Neste caso, o ângulo ãno áest definido
    return acos(num / den);
}

bool equals(double x, double y) {
    static const double EPS { 1e-6 };
    return fabs(x - y) < EPS;
}

bool contains(const Point<T>& P) const
{
    if (n < 3) return false;
    auto sum = 0.0;
    for (int i = 0; i < n - 1; ++i) {
        auto d = D(P, vs[i], vs[i + 1]);
        auto a = angle(P, vs[i], vs[i + 1]);
        sum += d > 0 ? a : (d < 0 ? -a : 0);
    }
    static const double PI = acos(-1.0);
    return equals(fabs(sum), 2*PI);
}

// çãInterseo entre a reta AB e o segmento de reta PQ
Point<T> intersection(const Point<T>& P, const Point<T>& Q,
                      const Point<T>& A, const Point<T>& B)
{
    auto a = B.y - A.y;
    auto b = A.x - B.x;
    auto c = B.x * A.y - A.x * B.y;
    auto u = fabs(a * P.x + b * P.y + c);
    auto v = fabs(a * Q.x + b * Q.y + c);

    // éMdia ponderada pelas âdistncias de P e Q éat a reta AB
    return {(P.x * v + Q.x * u)/(u + v), (P.y * v + Q.y * u)/(u + v)};
}
```

```cpp
    // Corta o ípolgono com a reta r que passa por A e B
    Polygon cut_polygon(const Point<T>& A, const Point<T>& B) const
    {
        vector<Point<T>> points;
        const double EPS { 1e-6 };
        for (int i = 0; i < n; ++i)
        {
            auto d1 = D(A, B, vs[i]);
            auto d2 = D(A, B, vs[i + 1]);

            // éVrtice à esquerda da reta
            if (d1 > -EPS)
                points.push_back(vs[i]);

            // A aresta cruza a reta
            if (d1 * d2 < -EPS)
                points.push_back(intersection(vs[i], vs[i + 1], A, B));
        }
        return Polygon(points);
    }

    double circumradius() const
    {
        auto s = distance(vs[0], vs[1]);
        const double PI { acos(-1.0) };
        return (s/2.0)*(1.0/sin(PI/n));
    }

    double apothem() const
    {
        auto s = distance(vs[0], vs[1]);
        const double PI { acos(-1.0) };
        return (s/2.0)*(1.0/tan(PI/n));
    }
};
```

1.10 3d

```cpp
template<typename T>
struct Point3D { T x, y, z; };

template<typename T>
struct Sphere {
    Point3D<T> C;
    T r;

    double area() const
    {
        return 4.0*PI*r*r;
    }

    double volume() const
    {
        return 4.0*PI*r*r*r/3.0;
    }
};

template<typename T>
```

```
struct Cylinder {
    T r, h;

    double area() const
    {
        return 2*PI*r*(r + h);
    }

    double volume() const
    {
        return PI*r*r*h;
    }
};

template<typename T>
struct Cube {
    T L;

    double face_diagonal() const
    {
        return L*sqrt(2.0);
    }

    double space_diagonal() const
    {
        return L*sqrt(3.0);
    }

    double area() const
    {
        return 6.0*L*L;
    }

    double volume() const
    {
        return L*L*L;
    }
};

template<typename T>
struct Cone {
    T r, H;

    double volume() const
    {
        return PI*r*r*H/3.0;
    }

    double area() const
    {
        return PI*r*r + PI*r*sqrt(r*r + H*H);
    }

    // Volume do tronco do cone
    double frustum(double rm, double h) const
    {
        return PI*h*(r*r + r*rm + rm*rm)/3.0;
    }
};

template<typename T>
struct Parallelepiped {
    Vector3D<T> u, v, w;

    double volume() const
    {
        return fabs(u.x*v.y*w.z + u.y*v.z*w.x + u.z*v.x*w.y
                - (u.x*v.z*.wy + u.y*v.x*w.z + u.z*v.y*w.x);
    }

    double volume2() const
    {
        double a = u.lenght();
        double b = v.length();
        double c = w.length();

        double m = angle(u, v);
        double n = angle(u, w);
        double p = angle(v, w);

        return a*b*c*sqrt(1 + 2*cos(m)*cos(n)*cos(p)
            - cos(m)*cos(m) - cos(n)*cos(n) - cos(p)*cos(p));
    }

    double volume3() const
    {
        return fabs(dot_product(u, cross_product(v, w)));
    }

    double area() const
    {
        double uv = cross_product(u, v).length();
        double uw = cross_product(u, w).length();
        double vw = cross_product(v, w).length();

        return 2*(uv + uw + vw);
    }
};
```

1.11 Envoltorio Convexo

```
template<typename T>
class GrahamScan {
    static Point<T> pivot(vector<Point<T>>& P)
    {
        size_t idx = 0;
        for (size_t i = 1; i < P.size(); ++i)
            if (P[i].y < P[idx].y or (equals(P[i].y, P[idx].y) and P[i].x > P[
    idx].x))
                idx = i;
        swap(P[0], P[idx]);
        return P[0];
    }

    static void sort_by_angle(vector<Point<T>>& P)
    {
```

```
        auto P0 = pivot(P);
        sort(P.begin() + 1, P.end(), [&](const Point<T>& A, const Point<T>& B)
 {
            // pontos colineares: escolhe-se o mais óprximo do ôpiv
            if (equals(D(P0, A, B), 0)) return A.distance(P0) < B.distance(P0)
;
            auto alfa = atan2(A.y - P0.y, A.x - P0.x);
            auto beta = atan2(B.y - P0.y, B.x - P0.x);
            return alfa < beta;
        });
    }

    static vector<Point<T>> convex_hull(const vector<Point<T>>& points)
    {
        vector<Point<T>> P(points);
        auto N = P.size();
        // Corner case: com 3 évrtices ou menos, P é o óprprio convex hull
        if (N <= 3) return P;
        sort_by_angle(P);
        vector<Point<T>> ch;
        ch.push_back(P[N - 1]);
        ch.push_back(P[0]);
        ch.push_back(P[1]);
        size_t i = 2;
        while (i < N)
        {
            auto j = ch.size() - 1;
            if (D(ch[j - 1], ch[j], P[i]) > 0)
                ch.push_back(P[i++]);
            else
                ch.pop_back();
        }
        // O óenvoltrio é um caminho fechado: o primeiro ponto é igual ao
    último
        return ch;
    }
};


// Cadeia ómontona de Andrew
template<typename T>
vector<Point<T>> make_hull(const vector<Point<T>>& points, vector<Point<T>>&
    hull)
{
    for (const auto& p : points)
    {
        auto size = hull.size();

        while (size >= 2 and D(hull[size - 2], hull[size - 1], p) <= 0)
        {
            hull.pop_back();
            size = hull.size();
        }

        hull.push_back(p);
    }

    return hull;
}

template<typename T>
vector<Point<T>> monotone_chain(const vector<Point<T>>& points)
{
    vector<Point<T>> P(points);

    sort(P.begin(), P.end());

    vector<Point<T>> lower, upper;

    lower = make_hull(P, lower);

    reverse(P.begin(), P.end());

    upper = make_hull(P, upper);

    lower.pop_back();
    lower.insert(lower.end(), upper.begin(), upper.end());

    return lower;
}
```

2 Matematica

2.1 Mdc

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

using namespace std;

long long gcd(long long a, long long b)
{
    return b ? gcd(b, a % b) : a;
}

long long ext_gcd(long long a, long long b, long long& x, long long& y)
{
    if (b == 0)
    {
        x = 1;
        y = 0;
        return a;
    }

    long long x1, y1;
    long long d = ext_gcd(b, a % b, x1, y1);

    x = y1;
    y = x1 - y1*(a/b);

    return d;
}

int main()
{
    long long a, b;
```

```
    cin >> a >> b;

    cout << "(" << a << ", " << b << ") = " << gcd(a, b) << '\n';

    long long x, y;
    auto d = ext_gcd(a, b, x, y);

    cout << d << " = (" << a << ")(" << x << ") + (" << b << ")(" << y << ")\n
    ";

    return 0;
}
```

2.2 Primos

```
//(N ** fi de p) % p == 1 sempre
// sistema reduzido de íresduo é os diferentes restos que deixam (7 vai ter t
    =6) - pega todos os restos
// únmeros coprimos - únmero que mdc entre eles é 1
// coprimos de 6 = 1,4,5


// TEOREMA DE FERMAT
// a^p é congruente a a(mod p) - a é inteiro e p é primo
//

// TEOREMA DE EULER
// a^fi de m é congruente a 1 mod m
// ós de primo o fi é -1
// fatora em primo e sabe que é -1
// fi de qulquer valor é = fi de primo 1 * fi de primo 2

// Fatoracao em primos
#define ll long long

ll phi(){

}

ll fatp(int x){
    map<int, int> m;

    for(int i = 2; i * i < x; i++){
       while(x%i == 0){
        x/=i;
        m[i]++;
       }



    }


}

// verificar se é primo
bool is_p(int n){
    if(n < 2)
        return false;

    if(n == 2)
        return true;

    if(n%2 == 0)
        return false;

    for(int i = 3; i * i <= n; i+=2){
        if(n%i == 0)
            return false;
    }
    return true;
}

// crivo
vector<long, long> primes(ll N){
    bitset<MAX> sieve;
    vector<long long> ps{2};
    sieve.set();

    for(ll i = 3; i<=N; i+=2){
        if(sieve[i]){
            ps.push_back(i);
            for(ll j = i * i; j<=N; j+=2*i){
                sieve[j] = false;
            }
        }
    }
    return ps;
}
```

2.3 Fast Exp

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

using namespace std;

long long fast_exp(long long a, int n)
{
    if (n == 1)
        return a;

    auto x = fast_exp(a, n / 2);

    return x * x * (n % 2 ? a : 1);
}

long long fast_exp_it(long long a, int n)
{
    long long res = 1, base = a;

    while (n)
    {
        if (n & 1)
            res *= base;

        base *= base;
```

```
        n >>= 1;
    }

    return res;
}

int main()
{
    long long a;
    int n;

    cin >> a >> n;

    cout << a << "^" << n << " = " << fast_exp(a, n) << '\n';

    return 0;
}
```