计算几何

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

using i64 = long long;
using l64 = long double;

template<class T>
struct Point {
    T x;
    T y;

    Point<T>(T x_ = 0, T y_ = 0) : x(x_), y(y_) {}

    template<class U>
    operator Point<U>() {
        return Point<U>(U(x), U(y));
    }

    Point &operator+=(Point p) &{
        x += p.x;
        y += p.y;
        return *this;
    }

    Point &operator-=(Point p) &{
        x -= p.x;
        y -= p.y;
        return *this;
    }

    Point &operator*=(T v) &{
        x *= v;
        y *= v;
        return *this;
    }

    Point operator-() const {
        return Point<T>(-x, -y);
    }
```

```
    friend Point operator+(Point a, Point b) {
        return a += b;
    }

    friend Point operator-(Point a, Point b) {
        return a -= b;
    }

    friend Point operator*(Point a, T b) {
        return a *= b;
    }

    friend Point operator*(T a, Point b) {
        return b *= a;
    }

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

    friend std::istream &operator>>(std::istream &is, Point &p) {
        return is >> p.x >> p.y;
    }

    friend std::ostream &operator<<(std::ostream &os, Point p) {
        return os << p.x << " " << p.y;
    }
};

//a*b==0,两个向量垂直
template<class T>
T dot(Point<T> a, Point<T> b) {//计算两个点之间的点积。
    return a.x * b.x + a.y * b.y;
}

//a*b==0,两个向量平行
template<class T>
T cross(Point<T> a, Point<T> b) {//计算两个点之间的叉积
    return a.x * b.y - a.y * b.x;
}

template<class T>
T square(Point<T> p) {//计算一个点与其自身的点积
```

```
    return dot(p, p);
}

template<class T>
double length(Point<T> p) {//计算一个点表示的向量的长度
    return std::sqrt(double(square(p)));
}

long double length(Point<long double> p) {
    return std::sqrt(square(p));
}
//cos=a*b/(|a|*|b|)

template<class T>
struct Line {
    Point<T> a;
    Point<T> b;

    Line<T>(Point<T> a_ = Point<T>(), Point<T> b_ = Point<T>()) : a(a_), b(b_) {}

    friend std::istream &operator>>(std::istream &is, Line &l) {
        return is >> l.a.x >> l.a.y >> l.b.x >> l.b.y;
    }
};

template<class T>
T dist(Point<T> p, Line<T> line) {// 计算点道线段的距离
    if (line.a == line.b) {
        return dist(p, line.a);
    }
    Point<T> p1 = line.b - line.a, p2 = p - line.a, p3 = p - line.b;
    if (dot(p1, p2) < 0) return length(p2);
    if (dot(p1, p3) > 0) return length(p3);

    return fabs(cross(line.b - line.a, p - line.a) / length(line.b - line.a));
}

template<class T>
T dist(Point<T> a, Point<T> b) {//计算两点之间的距离
    return std::hypot(a.x - b.x, a.y - b.y);
}

template<class T>
```

```
T dist(Line<T> line) {//计算直线的距离
    return std::hypot(line.a.x - line.b.x, line.a.y - line.b.y);
}

template<class T>
Point<T> rotate(Point<T> a) {//将一个点绕原点旋转 90 度（逆时针）
    return Point<T>(-a.y, a.x);
}

//根据点的位置（相对于原点）返回一个符号值。如果点在 x 轴上方（或在 x 轴上但 y = 0 且 x > 0），则返回
template<class T>
int sgn(Point<T> a) {
    return a.y > 0 || (a.y == 0 && a.x > 0) ? 1 : -1;
}

template<class T>
bool pointOnLineLeft(Point<T> p, Line<T> l) {//判断点 p 是否在线段 l 的左侧（不包括线段上）。
    return cross(l.b - l.a, p - l.a) > 0;
}

template<class T>
Point<T> lineIntersection(Line<T> l1, Line<T> l2) {//计算两条线段 l1 和 l2 的交点。
    return l1.a + (l1.b - l1.a) * (cross(l2.b - l2.a, l1.a - l2.a) / cross(l2.b - l2.a, l1.a - l
}

template<class T>
bool pointOnSegment(Point<T> p, Line<T> l) {//判断点 p 是否在线段 l 上（包括端点）。
    return cross(p - l.a, l.b - l.a) == 0 && std::min(l.a.x, l.b.x) <= p.x && p.x <= std::max(l
           && std::min(l.a.y, l.b.y) <= p.y && p.y <= std::max(l.a.y, l.b.y);
}

template<class T>
bool pointInPolygon(Point<T> a, std::vector<Point<T>> p) {//判断点 a 是否在多边形 p 内部。
    int n = p.size();
    for (int i = 0; i < n; i++) {
        if (pointOnSegment(a, Line<T>(p[i], p[(i + 1) % n]))) {
            return true;
        }
    }

    int t = 0;
    for (int i = 0; i < n; i++) {
        auto u = p[i];
```

```
        auto v = p[(i + 1) % n];
        if (u.x < a.x && v.x >= a.x && pointOnLineLeft(a, Line<T>(v, u))) {
            t ^= 1;
        }
        if (u.x >= a.x && v.x < a.x && pointOnLineLeft(a, Line<T>(u, v))) {
            t ^= 1;
        }
    }

    return t == 1;
}

template<class T>
std::vector<Point<T>> Andrew(std::vector<Point<T>> p) {//求凸包
    std::sort(p.begin(), p.end(), [&](Point<T> x, Point<T> y) {
        return x.x != y.x ? x.x < y.x : x.y < y.y;
    });
    std::vector<Point<T>> stk;
    int n = p.size();
    for (int i = 0; i < n; i++) {
        while (stk.size() > 1 && cross(stk.back() - stk[stk.size() - 2], p[i] - stk[stk.size() -
            stk.pop_back();
        stk.push_back(p[i]);
    }
    int tmp = stk.size();
    for (int i = n - 2; i >= 0; i--) {
        while (stk.size() > tmp && cross(stk.back() - stk[stk.size() - 2], p[i] - stk[stk.size()
            stk.pop_back();
        stk.push_back(p[i]);
    }
    stk.pop_back();
    return stk;
}

template<class T>
std::pair<Point<T>, Point<T>> rotatingCalipers(std::vector<Point<T>> &p) {//旋转卡壳求最远点对距离
    T res = 0;
    std::pair<Point<T>, Point<T>> ans;
    int n = p.size();
    for (int i = 0, j = 1; i < n; i++) {
        while (cross(p[i + 1] - p[i], p[j] - p[i]) < cross(p[i + 1] - p[i], p[j + 1] - p[i])) j
        if (dist(p[i], p[j]) > res) {
            ans = {p[i], p[j]};
```

```
            res = dist(p[i], p[j]);
        }
        if (dist(p[i + 1], p[j]) > res) {
            ans = {p[i + 1], p[j]};
            res = dist(p[i + 1], p[j]);
        }
    }
    return ans;
}

// 0 : not intersect不相交
// 1 : strictly intersect严格相交
// 2 : overlap重叠
// 3 : intersect at endpoint在端点相交
//判断两条线段 l1 和 l2 是否相交，
template<class T>
std::tuple<int, Point<T>, Point<T>> segmentIntersection(Line<T> l1, Line<T> l2) {
    if (std::max(l1.a.x, l1.b.x) < std::min(l2.a.x, l2.b.x)) {
        return {0, Point<T>(), Point<T>()};
    }
    if (std::min(l1.a.x, l1.b.x) > std::max(l2.a.x, l2.b.x)) {
        return {0, Point<T>(), Point<T>()};
    }
    if (std::max(l1.a.y, l1.b.y) < std::min(l2.a.y, l2.b.y)) {
        return {0, Point<T>(), Point<T>()};
    }
    if (std::min(l1.a.y, l1.b.y) > std::max(l2.a.y, l2.b.y)) {
        return {0, Point<T>(), Point<T>()};
    }
    if (cross(l1.b - l1.a, l2.b - l2.a) == 0) {
        if (cross(l1.b - l1.a, l2.a - l1.a) != 0) {
            return {0, Point<T>(), Point<T>()};
        } else {
            auto maxx1 = std::max(l1.a.x, l1.b.x);
            auto minx1 = std::min(l1.a.x, l1.b.x);
            auto maxy1 = std::max(l1.a.y, l1.b.y);
            auto miny1 = std::min(l1.a.y, l1.b.y);
            auto maxx2 = std::max(l2.a.x, l2.b.x);
            auto minx2 = std::min(l2.a.x, l2.b.x);
            auto maxy2 = std::max(l2.a.y, l2.b.y);
            auto miny2 = std::min(l2.a.y, l2.b.y);
            Point<T> p1(std::max(minx1, minx2), std::max(miny1, miny2));
            Point<T> p2(std::min(maxx1, maxx2), std::min(maxy1, maxy2));
```

```
            if (!pointOnSegment(p1, l1)) {
                std::swap(p1.y, p2.y);
            }
            if (p1 == p2) {
                return {3, p1, p2};
            } else {
                return {2, p1, p2};
            }
        }
    }
    auto cp1 = cross(l2.a - l1.a, l2.b - l1.a);
    auto cp2 = cross(l2.a - l1.b, l2.b - l1.b);
    auto cp3 = cross(l1.a - l2.a, l1.b - l2.a);
    auto cp4 = cross(l1.a - l2.b, l1.b - l2.b);

    if ((cp1 > 0 && cp2 > 0) || (cp1 < 0 && cp2 < 0) || (cp3 > 0 && cp4 > 0) || (cp3 < 0 && cp4
        return {0, Point<T>(), Point<T>()};
    }

    Point p = lineIntersection<T>(l1, l2);
    if (cp1 != 0 && cp2 != 0 && cp3 != 0 && cp4 != 0) {
        return {1, p, p};
    } else {
        return {3, p, p};
    }
}

//判断一条线段 l 是否完全位于一个多边形 p 内部
template<class T>
bool segmentInPolygon(Line<T> l, std::vector<Point<T>> p) {
    int n = p.size();
    if (!pointInPolygon(l.a, p)) {
        return false;
    }
    if (!pointInPolygon(l.b, p)) {
        return false;
    }
    for (int i = 0; i < n; i++) {
        auto u = p[i];
        auto v = p[(i + 1) % n];
        auto w = p[(i + 2) % n];
        auto[t, p1, p2] = segmentIntersection(l, Line<T>(u, v));
```

```
if (t == 1) {
    return false;
}
if (t == 0) {
    continue;
}
if (t == 2) {
    if (pointOnSegment(v, l) && v != l.a && v != l.b) {
        if (cross(v - u, w - v) > 0) {
            return false;
        }
    }
} else {
    if (p1 != u && p1 != v) {
        if (pointOnLineLeft(l.a, Line<T>(v, u))
            || pointOnLineLeft(l.b, Line<T>(v, u))) {
            return false;
        }
    } else if (p1 == v) {
        if (l.a == v) {
            if (pointOnLineLeft(u, l)) {
                if (pointOnLineLeft(w, l)
                    && pointOnLineLeft(w, Line<T>(u, v))) {
                    return false;
                }
            } else {
                if (pointOnLineLeft(w, l)
                    || pointOnLineLeft(w, Line<T>(u, v))) {
                    return false;
                }
            }
        } else if (l.b == v) {
            if (pointOnLineLeft(u, Line<T>(l.b, l.a))) {
                if (pointOnLineLeft(w, Line<T>(l.b, l.a))
                    && pointOnLineLeft(w, Line<T>(u, v))) {
                    return false;
                }
            } else {
                if (pointOnLineLeft(w, Line<T>(l.b, l.a))
                    || pointOnLineLeft(w, Line<T>(u, v))) {
                    return false;
                }
            }
```

```
                } else {
                    if (pointOnLineLeft(u, l)) {
                        if (pointOnLineLeft(w, Line<T>(l.b, l.a))
                            || pointOnLineLeft(w, Line<T>(u, v))) {
                            return false;
                        }
                    } else {
                        if (pointOnLineLeft(w, l)
                            || pointOnLineLeft(w, Line<T>(u, v))) {
                            return false;
                        }
                    }
                }
            }
        }
    }
    return true;
}

template<class T>
std::vector<Point<T>> hp(std::vector<Line<T>> lines) {
    std::sort(lines.begin(), lines.end(), [&](auto l1, auto l2) {
        auto d1 = l1.b - l1.a;
        auto d2 = l2.b - l2.a;

        if (sgn(d1) != sgn(d2)) {
            return sgn(d1) == 1;
        }

        return cross(d1, d2) > 0;
    });

    std::deque<Line<T>> ls;
    std::deque<Point<T>> ps;
    for (auto l: lines) {
        if (ls.empty()) {
            ls.push_back(l);
            continue;
        }

        while (!ps.empty() && !pointOnLineLeft(ps.back(), l)) {
            ps.pop_back();
            ls.pop_back();
```

```
        }

        while (!ps.empty() && !pointOnLineLeft(ps[0], l)) {
            ps.pop_front();
            ls.pop_front();
        }

        if (cross(l.b - l.a, ls.back().b - ls.back().a) == 0) {
            if (dot(l.b - l.a, ls.back().b - ls.back().a) > 0) {

                if (!pointOnLineLeft(ls.back().a, l)) {
                    assert(ls.size() == 1);
                    ls[0] = l;
                }
                continue;
            }
            return {};
        }

        ps.push_back(lineIntersection(ls.back(), l));
        ls.push_back(l);
    }

    while (!ps.empty() && !pointOnLineLeft(ps.back(), ls[0])) {
        ps.pop_back();
        ls.pop_back();
    }
    if (ls.size() <= 2) {
        return {};
    }
    ps.push_back(lineIntersection(ls[0], ls.back()));

    return std::vector<T>(ps.begin(), ps.end());
}

void DAOQI() {
    std::cout << "0 0\n";
}

signed main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int T = 1;
```

```
    //std::cin >> T;
    while (T--) DAOQI();
    return 0;
}
```

FFT/快速傅里叶变换

```
//https://ac.nowcoder.com/acm/contest/108305/J

#include <iostream>
#include <vector>
#include <cmath>
#include <complex>
#include <algorithm>
#include <string>
using namespace std;

const double PI = acos(-1);

// FFT函数：快速傅里叶变换
void fft(vector<complex<double>>& a, int n, int op) {
    // 位反转重排
    for (int i = 0, j = 0; i < n; i++) {
        if (i < j) swap(a[i], a[j]);
        for (int k = n >> 1; (j ^= k) < k; k >>= 1);
    }
    // 蝶形运算
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * op;
        complex<double> wn(cos(ang), sin(ang)); // 旋转因子
        for (int i = 0; i < n; i += len) {
            complex<double> w(1, 0);
            for (int j = 0; j < len / 2; j++) {
                complex<double> u = a[i + j];
                complex<double> v = w * a[i + j + len / 2];
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wn; // 更新旋转因子
            }
        }
    }
    // 逆变换需缩放
    if (op == -1) {
        for (int i = 0; i < n; i++) {
            a[i] /= n;
        }
    }
```

```
}

// 卷积计算：使用FFT或暴力法（小规模数据）
vector<int> convolve(vector<int>& a, vector<int>& b) {
    if (a.empty() || b.empty()) {
        return vector<int>();
    }
    int n1 = a.size(), n2 = b.size();
    int res_size = n1 + n2 - 1;

    // 小规模数据使用暴力卷积
    if (n1 <= 128 || n2 <= 128) {
        vector<int> c(res_size, 0);
        for (int i = 0; i < n1; i++) {
            for (int j = 0; j < n2; j++) {
                c[i + j] += a[i] * b[j];
            }
        }
        return c;
    }

    // 大规模数据使用FFT
    int n = 1;
    while (n < res_size) n <<= 1; // 扩展到2的幂
    vector<complex<double>> fa(n), fb(n);
    // 填充数据
    for (int i = 0; i < n1; i++) fa[i] = a[i];
    for (int i = 0; i < n2; i++) fb[i] = b[i];

    // FFT计算
    fft(fa, n, 1);   // 正变换
    fft(fb, n, 1);
    for (int i = 0; i < n; i++) fa[i] *= fb[i]; // 点乘
    fft(fa, n, -1);  // 逆变换

    // 取整并返回结果
    vector<int> c(res_size);
    for (int i = 0; i < res_size; i++) {
        c[i] = round(fa[i].real());
    }
    return c;
}
```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int T;
    cin >> T;
    while (T--) {
        string A, B;
        cin >> A >> B;
        // 反转字符串：低位在索引0
        reverse(A.begin(), A.end());
        reverse(B.begin(), B.end());

        // 分离实部和虚部
        vector<int> A_real, A_img, B_real, B_img;
        for (int i = 0; i < A.size(); i++) {
            if (i % 2 == 0) A_real.push_back(A[i] - '0'); // 实部：偶数位
            else A_img.push_back(A[i] - '0');             // 虚部：奇数位
        }
        for (int i = 0; i < B.size(); i++) {
            if (i % 2 == 0) B_real.push_back(B[i] - '0');
            else B_img.push_back(B[i] - '0');
        }

        // 计算四个卷积
        vector<int> rr = convolve(A_real, B_real); // 实*实
        vector<int> ii = convolve(A_img, B_img);   // 虚*虚
        vector<int> ri = convolve(A_real, B_img);  // 实*虚
        vector<int> ir = convolve(A_img, B_real);  // 虚*实

        // 构造实部多项式：rr - 2*ii
        int len_real = max(rr.size(), ii.size());
        vector<int> real_poly(len_real, 0);
        for (int i = 0; i < rr.size(); i++) real_poly[i] = rr[i];
        for (int i = 0; i < ii.size(); i++) real_poly[i] -= 2 * ii[i];

        // 构造虚部多项式：ri + ir
        int len_img = max(ri.size(), ir.size());
        vector<int> img_poly(len_img, 0);
        for (int i = 0; i < ri.size(); i++) img_poly[i] = ri[i];
        for (int i = 0; i < ir.size(); i++) img_poly[i] += ir[i];

        // 合并多项式：实部在偶数位，虚部在奇数位
        int max_index = 2 * max(len_real, len_img) + 1000; // 确保足够大
```

```
        vector<int> d(max_index, 0);
        for (int i = 0; i < len_real; i++) d[2 * i] = real_poly[i]; // 实部
        for (int i = 0; i < len_img; i++) d[2 * i + 1] = img_poly[i]; // 虚部

        // 进位调整：从低位到高位
        for (int k = 0; k < max_index; k++) {
            if (k >= max_index - 2) break; // 防止越界
            int ck = d[k];
            // 调整到0或1：ck mod 2
            int r = ck % 2;
            if (r < 0) r += 2; // 确保非负
            r %= 2;
            // 计算进位量（基的递推关系）
            int carry = (r - ck) / 2;
            d[k] = r; // 当前位保留0或1
            d[k + 2] += carry; // 进位到高位
        }

        // 寻找最高非零位
        int last_non_zero = max_index - 1;
        while (last_non_zero >= 0 && d[last_non_zero] == 0) {
            last_non_zero--;
        }

        // 输出结果
        if (last_non_zero < 0) {
            cout << "0\n"; // 全零情况
        } else {
            string ans;
            // 从高位到低位生成字符串
            for (int i = last_non_zero; i >= 0; i--) {
                ans += ('0' + d[i]);
            }
            cout << ans << '\n';
        }
    }
    return 0;
}
```

扫描线

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

const int MAX_N = 150000;        // 最大商店数量
const int MAX_VAL = 1000000;     // 最大独特度值

// 事件结构体：用于表示扫描线中的事件
struct Event {
    int pos;   // 事件触发的位置（对应b1的位置）
    int l, r;  // 影响b2的区间[l, r]
    int type;  // 事件类型：+1表示添加贡献，-1表示移除贡献
};

// 线段树结构体：用于区间更新和最大值查询
struct SegmentTree {
    struct Node {
        int l, r;         // 节点代表的区间[l, r]
        int max_val;      // 区间最大值
        int idx;          // 最大值对应的位置（b2的值）
        int lazy;         // 懒惰标记，用于延迟更新
    };
    vector<Node> tree;    // 线段树节点数组
    int n;                // 线段树覆盖的总区间大小

    // 构造函数：初始化线段树
    SegmentTree(int size) {
        n = size;
        tree.resize(4 * n + 10);  // 分配4倍空间
        build(1, 1, n);           // 构建线段树
    }

    // 构建线段树
    void build(int id, int l, int r) {
        tree[id].l = l;
        tree[id].r = r;
        tree[id].lazy = 0;
        if (l == r) {
```

```
        tree[id].max_val = 0;  // 初始值设为0
        tree[id].idx = l;      // 记录位置
        return;
    }
    int mid = (l + r) / 2;
    build(id * 2, l, mid);       // 构建左子树
    build(id * 2 + 1, mid + 1, r); // 构建右子树
    push_up(id);                 // 更新父节点
}

// 更新父节点信息
void push_up(int id) {
    // 选择左右子树中较大的值
    if (tree[id * 2].max_val >= tree[id * 2 + 1].max_val) {
        tree[id].max_val = tree[id * 2].max_val;
        tree[id].idx = tree[id * 2].idx;
    } else {
        tree[id].max_val = tree[id * 2 + 1].max_val;
        tree[id].idx = tree[id * 2 + 1].idx;
    }
}

// 下推懒惰标记
void push_down(int id) {
    if (tree[id].lazy != 0) {
        int lazy_val = tree[id].lazy;
        // 更新左子树
        tree[id * 2].lazy += lazy_val;
        tree[id * 2].max_val += lazy_val;
        // 更新右子树
        tree[id * 2 + 1].lazy += lazy_val;
        tree[id * 2 + 1].max_val += lazy_val;
        tree[id].lazy = 0;  // 清除标记
    }
}

// 区间更新接口
void update(int l, int r, int val) {
    update(1, l, r, val);
}

// 实际更新操作
void update(int id, int L, int R, int val) {
```

```
    // 当前节点区间与更新区间无交集
    if (tree[id].l > R || tree[id].r < L) {
        return;
    }
    // 当前节点区间完全包含在更新区间内
    if (L <= tree[id].l && tree[id].r <= R) {
        tree[id].lazy += val;      // 更新懒惰标记
        tree[id].max_val += val;   // 更新节点值
        return;
    }
    push_down(id);  // 下推懒惰标记
    int mid = (tree[id].l + tree[id].r) / 2;
    if (L <= mid) {
        update(id * 2, L, R, val);  // 更新左子树
    }
    if (R > mid) {
        update(id * 2 + 1, L, R, val);  // 更新右子树
    }
    push_up(id);  // 更新父节点
}

// 区间查询接口
pair<int, int> query(int l, int r) {
    return query(1, l, r);
}

// 实际查询操作
pair<int, int> query(int id, int L, int R) {
    // 当前节点区间与查询区间无交集
    if (tree[id].l > R || tree[id].r < L) {
        return {-1000000000, 0};  // 返回极小值
    }
    // 当前节点区间完全包含在查询区间内
    if (L <= tree[id].l && tree[id].r <= R) {
        return {tree[id].max_val, tree[id].idx};
    }
    push_down(id);  // 下推懒惰标记
    int mid = (tree[id].l + tree[id].r) / 2;
    // 查询区间完全在左子树
    if (R <= mid) {
        return query(id * 2, L, R);
    }
    // 查询区间完全在右子树
```

```
        else if (L > mid) {
            return query(id * 2 + 1, L, R);
        }
        // 查询区间跨越左右子树
        else {
            pair<int, int> left_res = query(id * 2, L, R);
            pair<int, int> right_res = query(id * 2 + 1, L, R);
            // 返回左右子树中的最大值及其位置
            if (left_res.first >= right_res.first) {
                return left_res;
            } else {
                return right_res;
            }
        }
    }
};
```

SPFA

```
#include<bits/stdc++.h>
using namespace std;
int n,m,s,t,tot=0;
int head[10010];

struct node{
    int t,l,next;
}edge[20020];

void addedge(int x,int y,int z)
{
    edge[++ tot].l = z;
    edge[tot].t = y;
    edge[tot].next = head[x];
    head[x] = tot;
}

int dis[10010];
int vis[10010]={0};

queue<int> q;

int SPFA(int a,int b)
{
    memset(dis,0x3f,sizeof(dis));
    dis[a] = 0;
    vis[a] = 1;
    q.push(s);
    while(!q.empty())
    {
        int x = q.front();
        q.pop();
        vis[x] = 0;
        for(int i = head[x];i != -1;i = edge[i].next)
        {
            int y = edge[i].t;
            if(dis[y] > dis[x] + edge[i].l)
            {
                dis[y] = dis[x] + edge[i].l;
                if(!vis[y])
```

```
                {
                    q.push(y);
                    vis[y] = 1;
                }
            }
        }
    }
    if(dis[b] >= 0x3f3f3f3f)
    {
        return -1;
    }
    return dis[b];
}

int main()
{
    cin >> n >> m >> s >> t;
    memset(head,-1,sizeof(head));
    memset(edge,-1,sizeof(edge));
    while(m --)
    {
        int x,y,z;
        cin >> x >> y >> z;
        addedge(x,y,z);
        addedge(y,x,z);
    }
    int ans = SPFA(s,t);
    cout << ans << "\n";
    return 0;
}
```

迪杰斯特拉

```
#include<bits/stdc++.h>
using namespace std;
int n,m,s,t,tot=0;
int head[10010];

struct node{
    int t,l,next;
}edge[20020];


void addedge(int x,int y,int z)
{
    edge[++tot].l=z;
    edge[tot].t=y;
    edge[tot].next=head[x];
    head[x]=tot;
}

struct ty
{
    int x,dis;
    bool operator < (const ty &a) const
    {
        return dis>a.dis;
    }
};

priority_queue<ty> q;
int dis[10010];
int vis[10010]={0};

int dij(int a,int b)
{
    memset(dis,0x3f,sizeof(dis));
    dis[a]=0;
    ty tmp;
    tmp.x=a;
    tmp.dis=0;
    q.push(tmp);
    while(!q.empty())
```

```
    {
        ty ind=q.top();
        q.pop();
        if(vis[ind.x])
        {
            continue;
        }
        vis[ind.x]=1;
        for(int i=head[ind.x];i!=-1;i=edge[i].next)
        {
            int y=edge[i].t;
            if(vis[y])continue;
            if(dis[y]>dis[ind.x]+edge[i].l)
            {
                dis[y]=dis[ind.x]+edge[i].l;
                ty ind2;
                ind2.x=y;
                ind2.dis=dis[y];
                q.push(ind2);
            }

        }
    }
    if(dis[b]>=0x3f3f3f3f)return -1;
    return dis[b];
}

int main()
{
    cin>>n>>m>>s>>t;
    memset(edge,-1,sizeof(edge));
    memset(head,-1,sizeof(head));
    while(m--)
    {
        int x,y,z;
        cin>>x>>y>>z;
        addedge(x,y,z);
        addedge(y,x,z);
    }
    int ans=dij(s,t);
    cout<<ans<<"\n";
```

```
    return 0;
}
```