矩阵快速幂

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
using ll = long long;
const int MOD = 1e9 + 7;

struct Matrix {
    int n, m;
    vector<vector<ll>> a;
    Matrix(int _n, int _m) : n(_n), m(_m), a(_n + 1, vector<ll>(_m + 1, 0))
{}

    static Matrix identity(int n) {
        Matrix I(n, n);
        for (int i = 1; i <= n; ++i) {
            I.a[i][i] = 1;
        }
        return I;
    }
};

Matrix operator*(const Matrix& A, const Matrix& B) {
    Matrix C(A.n, B.m);
    for (int i = 1; i <= A.n; ++i) {
        for (int j = 1; j <= B.m; ++j) {
            for (int k = 1; k <= A.m; ++k) {
                C.a[i][j] = (C.a[i][j] + A.a[i][k] * B.a[k][j]) % MOD;
            }
        }
    }
    return C;
}

Matrix power(Matrix A, ll p) {
    Matrix res = Matrix::identity(A.n);
    while (p) {
        if (p & 1) res = res * A;
        A = A * A;
        p >>= 1;
    }
    return res;
}

void solve() {
    ll n;
    cin >> n;
    if (n <= 3) {
        cout << 1 << endl;
```

```
            return;
        }
        Matrix m(3, 3);
        // 正确1-indexed赋值转移矩阵
        m.a[1][2] = 1;  // 第1行第2列
        m.a[2][3] = 1;  // 第2行第3列
        m.a[3][1] = 1;  // 第3行第1列
        m.a[3][3] = 1;  // 第3行第3列

        Matrix a(3, 1);
        a.a[1][1] = 1;  // f1 = 1
        a.a[2][1] = 1;  // f2 = 1
        a.a[3][1] = 1;  // f3 = 1

        Matrix ans = power(m, n - 3) * a;
        // 输出结果矩阵的第3行第1列（fₙ）
        cout << ans.a[3][1] << endl;
    }

    signed main() {
        ios::sync_with_stdio(false);
        cin.tie(0), cout.tie(0);
        int t;
        cin >> t;
        while (t--) solve();
        return 0;
    }
```

背包问题模板（C++）

以下是01背包、完全背包和多重背包的C++实现模板：

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

// 01背包问题模板
int knapsack_01(int V, vector<int>& weights, vector<int>& values) {
    int n = weights.size();
    vector<int> dp(V + 1, 0);

    for (int i = 0; i < n; i++) {
        for (int j = V; j >= weights[i]; j--) {
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
        }
    }

    return dp[V];
}
```

```cpp
// 完全背包问题模板
int knapsack_complete(int V, vector<int>& weights, vector<int>& values) {
    int n = weights.size();
    vector<int> dp(V + 1, 0);

    for (int i = 0; i < n; i++) {
        for (int j = weights[i]; j <= V; j++) {
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
        }
    }

    return dp[V];
}

// 多重背包问题模板（二进制优化）
int knapsack_multiple(int V, vector<int>& weights, vector<int>& values,
vector<int>& counts) {
    // 二进制优化处理
    vector<int> new_weights, new_values;
    int n = weights.size();

    for (int i = 0; i < n; i++) {
        int count = counts[i];
        for (int k = 1; k <= count; k *= 2) {
            new_weights.push_back(weights[i] * k);
            new_values.push_back(values[i] * k);
            count -= k;
        }
        if (count > 0) {
            new_weights.push_back(weights[i] * count);
            new_values.push_back(values[i] * count);
        }
    }

    // 转换为01背包问题
    return knapsack_01(V, new_weights, new_values);
}

int main() {
    // 示例使用
    int V = 10; // 背包容量

    // 01背包示例
    vector<int> weights_01 = {2, 3, 4, 5};
    vector<int> values_01 = {3, 4, 5, 6};
    cout << "01背包最大价值: " << knapsack_01(V, weights_01, values_01) <<
endl;

    // 完全背包示例
    vector<int> weights_complete = {2, 3, 4, 5};
    vector<int> values_complete = {3, 4, 5, 6};
    cout << "完全背包最大价值: " << knapsack_complete(V, weights_complete,
values_complete) << endl;
```

```
    // 多重背包示例
    vector<int> weights_multiple = {2, 3, 4, 5};
    vector<int> values_multiple = {3, 4, 5, 6};
    vector<int> counts = {1, 2, 3, 2}; // 每个物品的数量
    cout << "多重背包最大价值: " << knapsack_multiple(V, weights_multiple,
values_multiple, counts) << endl;

    return 0;
}
```

模板说明

1. 01背包问题

- 特点：每种物品只有一件，只能选择放或不放
- 关键代码：内层循环从大到小遍历容量

```
for (int j = V; j >= weights[i]; j--) {
    dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
}
```

2. 完全背包问题

- 特点：每种物品有无限件，可以重复选择
- 关键代码：内层循环从小到大遍历容量

```
for (int j = weights[i]; j <= V; j++) {
    dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
}
```

3. 多重背包问题

- 特点：每种物品有数量限制
- 关键优化：使用二进制优化将多重背包转化为01背包
- 优化原理：将物品数量拆分为1, 2, 4, ..., 2^(k-1), c-2^k+1的形式，减少物品数量

使用提示

1. 确保输入的重量和价值数组大小一致
2. 对于多重背包，确保数量数组与重量和价值数组大小一致
3. 可以根据需要修改返回值为dp数组，以获取更详细的结果

这些模板可以直接在算法竞赛或面试中使用，涵盖了背包问题的三种基本变体。

主席树变式：查询区间第几大的数

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

struct Node {
    int count; // 存储的是计数，而不是原始值
    int left_child_idx;
    int right_child_idx;

    explicit Node(int cnt = 0, int lc = 0, int rc = 0) : count(cnt),
left_child_idx(lc), right_child_idx(rc) {}
};

class PersistentSegmentTree {
private:
    std::vector<Node> tree_nodes;
    std::vector<int> roots; // roots[i] 存储的是考虑了前 i 个元素后的权值线段树的
根
    int next_node_idx;
    int value_range_size; // 离散化后的值域大小

    // 构建一个空的线段树，所有计数为 0
    int build_empty_tree(const int l, const int r) {
        const int current_node_idx = next_node_idx++;
        if (l == r) {
            tree_nodes[current_node_idx].count = 0; // 初始计数为0
            return current_node_idx;
        }

        const int mid = l + (r - l) / 2;
        tree_nodes[current_node_idx].left_child_idx = build_empty_tree(l,
mid);
        tree_nodes[current_node_idx].right_child_idx = build_empty_tree(mid
+ 1, r);
        tree_nodes[current_node_idx].count = 0; // 空树总和为0
        return current_node_idx;
    }

    // 更新操作：在权值线段树中，将离散化后的值 p 的计数增加 c (这里 c 通常是 1)
    int update(int prev_node_idx, int l, int r, int p, int c_val) {
        int current_node_idx = next_node_idx++;
        tree_nodes[current_node_idx] = tree_nodes[prev_node_idx]; // 复制旧
节点信息

        // 仅增加当前节点的 count
        tree_nodes[current_node_idx].count += c_val;

        if (l == r) {
            return current_node_idx;
        }

        int mid = l + (r - l) / 2;
```

```
        if (p <= mid) {
            tree_nodes[current_node_idx].left_child_idx =
update(tree_nodes[prev_node_idx].left_child_idx, l, mid, p, c_val);
        } else {
            tree_nodes[current_node_idx].right_child_idx =
update(tree_nodes[prev_node_idx].right_child_idx, mid + 1, r, p, c_val);
        }
        return current_node_idx;
    }

    // 查询区间 [left_root, right_root] 的第 k 小值
    // l, r 是当前节点代表的值域区间
    int query_kth(int left_root_idx, int right_root_idx, int l, int r, int
k) {
        if (l == r) {
            return l; // 返回离散化后的值域下标
        }

        // 计算左子树在 right_root 中的总数
        int count_in_right_left =
tree_nodes[tree_nodes[right_root_idx].left_child_idx].count;
        // 计算左子树在 left_root 中的总数
        int count_in_left_left =
tree_nodes[tree_nodes[left_root_idx].left_child_idx].count;

        // 区间 [l, r] 中，左子树（值域 [l, mid]）内的元素数量
        int count_in_current_left_subtree = count_in_right_left -
count_in_left_left;

        int mid = l + (r - l) / 2;
        if (k <= count_in_current_left_subtree) {
            // 第 k 小值在左子树中
            return query_kth(tree_nodes[left_root_idx].left_child_idx,
tree_nodes[right_root_idx].left_child_idx, l, mid, k);
        } else {
            // 第 k 小值在右子树中，k 减去左子树中的数量
            return query_kth(tree_nodes[left_root_idx].right_child_idx,
tree_nodes[right_root_idx].right_child_idx, mid + 1, r, k -
count_in_current_left_subtree);
        }
    }


public:
    // N 这里代表的是离散化后的值域大小，而不是原始数组的长度
    explicit PersistentSegmentTree(int _value_range_size) :
        value_range_size(_value_range_size),
        tree_nodes(4500000 + 5), // 估算节点数量，每个版本增加 logN 个节点
        next_node_idx(1)
    {
        roots.reserve(200005); // N+M 个版本
    }

    // 提供给外部调用的 build_empty_tree
```

```
    int build_empty_tree_public() {
        return build_empty_tree(1, value_range_size);
    }

    // 提供给外部调用的 update
    int update_public(int prev_root_idx, int p, int c) {
        return update(prev_root_idx, 1, value_range_size, p, c);
    }

    // 提供给外部调用的 query_kth
    int query_kth_public(int l_root_idx, int r_root_idx, int k) {
        return query_kth(l_root_idx, r_root_idx, 1, value_range_size, k);
    }

    // 暴露 roots 向量以便外部直接管理版本
    std::vector<int>& get_roots() {
        return roots;
    }
};

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);
    int N, M;
    std::cin >> N >> M;

    std::vector<int> initial_array(N);
    std::vector<int> all_values; // 收集所有值用于离散化
    for (int i = 0; i < N; ++i) {
        std::cin >> initial_array[i];
        all_values.push_back(initial_array[i]);
    }

    // 离散化
    sort(all_values.begin(), all_values.end());
    all_values.erase(unique(all_values.begin(), all_values.end()),
all_values.end());

    // 离散化后的值域大小
    int distinct_values_count = all_values.size();

    PersistentSegmentTree pst(distinct_values_count);

    // roots[0] 为空树的根
    pst.get_roots().push_back(pst.build_empty_tree_public());

    // 构建 N 个版本
    for (int i = 0; i < N; ++i) {
        // 找到 initial_array[i] 在 all_values 中的离散化下标 (1-indexed)
        int discretized_idx = lower_bound(all_values.begin(),
all_values.end(), initial_array[i]) - all_values.begin() + 1;
        // 从上一个版本更新，将 discretized_idx 位置的计数加 1
        pst.get_roots().push_back(pst.update_public(pst.get_roots().back(),
discretized_idx, 1));
```

```
    }

    for (int i = 0; i < M; ++i) {
        int l, r, k;
        std::cin >> l >> r >> k;
        // 查询版本 roots[r] 和 roots[l-1] 之间的第 k 小值
        int result_discretized_idx = pst.query_kth_public(pst.get_roots()[l
- 1], pst.get_roots()[r], k);
        // 将离散化后的结果转换回原始值并输出 (注意 all_values 是 0-indexed)
        std::cout << all_values[result_discretized_idx - 1] << "\n";
    }

    return 0;
}
```

C++ 离散傅里叶变换 (DFT) 实现

以下是一个完整的 C++ DFT 实现，包含正向和逆向变换，以及使用示例。

```
#include <iostream>
#include <vector>
#include <complex>
#include <cmath>
#include <numbers> // C++20 引入的数学常数

using namespace std;
using namespace std::complex_literals; // 允许使用 i 表示虚数单位

// 正向离散傅里叶变换 (DFT)
vector<complex<double>> dft(const vector<complex<double>>& input) {
    int N = input.size();
    vector<complex<double>> output(N);

    for (int k = 0; k < N; k++) {
        output[k] = 0;
        for (int n = 0; n < N; n++) {
            // 使用欧拉公式计算旋转因子
            double angle = -2.0 * numbers::pi * k * n / N;
            output[k] += input[n] * exp(1i * angle);
        }
    }

    return output;
}

// 逆向离散傅里叶变换 (IDFT)
vector<complex<double>> idft(const vector<complex<double>>& input) {
    int N = input.size();
    vector<complex<double>> output(N);
```

```
    for (int k = 0; k < N; k++) {
        output[k] = 0;
        for (int n = 0; n < N; n++) {
            // 使用欧拉公式计算旋转因子
            double angle = 2.0 * numbers::pi * k * n / N;
            output[k] += input[n] * exp(1i * angle);
        }
        output[k] /= N; // 归一化
    }

    return output;
}

// 打印复数向量
void printComplexVector(const string& name, const vector<complex<double>>&
vec) {
    cout << name << ":" << endl;
    for (size_t i = 0; i < vec.size(); i++) {
        cout << "  [" << i << "]: " << vec[i] << endl;
    }
    cout << endl;
}

int main() {
    // 示例1: 简单的余弦波
    const int N = 8;
    vector<complex<double>> signal;

    // 生成一个简单的信号: 2Hz 余弦波 + 直流分量
    for (int i = 0; i < N; i++) {
        double t = static_cast<double>(i) / N;
        double value = 1.0 + cos(2.0 * numbers::pi * 2.0 * t); // 直流分量 +
2Hz 余弦波
        signal.push_back(value);
    }

    cout << "DFT 示例演示" << endl;
    cout << "============" << endl << endl;

    printComplexVector("原始信号", signal);

    // 计算 DFT
    auto spectrum = dft(signal);
    printComplexVector("频域谱 (DFT)", spectrum);

    // 计算 IDFT (应该恢复原始信号)
    auto reconstructed = idft(spectrum);
    printComplexVector("重建信号 (IDFT)", reconstructed);

    // 计算重建误差
    double error = 0.0;
    for (int i = 0; i < N; i++) {
        error += abs(signal[i] - reconstructed[i]);
    }
```

```
    cout << "总重建误差: " << error << endl;

    return 0;
}
```

编译和运行

如果您使用支持 C++20 的编译器，可以使用以下命令编译：

```
g++ -std=c++20 -o dft_example dft_example.cpp
./dft_example
```

对于不支持 C++20 的编译器，您可以将 `numbers::pi` 替换为 `M_PI`（需要包含 `<cmath>`），并使用 `-std=c++11` 或更高标准：

```
// 替换 numbers::pi
double angle = -2.0 * M_PI * k * n / N;
```

```
g++ -std=c++11 -o dft_example dft_example.cpp
./dft_example
```

说明

1. **DFT 函数**：实现了离散傅里叶变换的正向计算，将时域信号转换为频域表示
2. **IDFT 函数**：实现了逆向离散傅里叶变换，将频域表示恢复为时域信号
3. **复数支持**：使用 C++ 的 `std::complex` 类型处理复数运算
4. **示例**：生成一个包含直流分量和 2Hz 余弦波的信号，演示 DFT 和 IDFT 的完整过程

这个实现是基础的 DFT 算法，时间复杂度为 $O(N^2)$。对于更大的数据集，可以考虑使用快速傅里叶变换 (FFT) 算法，其时间复杂度为 O(N log N)。

例子

Problem A. SCUPC

Input file: standard input

Output file: standard output

Time limit: 5 seconds

Memory limit: 1024 megabytes

这是全国大学生程序设计竞赛双流区域赛。

小胖有 3 个长度为 n 的 0/1 串，分别为 s1, s2, s3，小胖想让这三个字符串中的 1 的尽量重合，也就是想 要尽量多的位置 i，使得 si,1 + si,2 + si,3 ≥ 2。 小胖有超能力，可以在这三个字符串中选择一个，并对其做任意多次循环左移。 小胖想知道， nP−1 i=0 [si,1 + si,2 + si,3 ≥ 2] 的最大值。

Input

第一行输入一个整数 t，表示数据组数。 对于每组数据，第一行先输入一个整数 n，表示字符串长度。 接下来三行，每一行输入一个 0/1 串，第 i 个 0/1 串代表 si。 t ≤ 10e5 , Pn ≤ 10e5

Output

对于每组数据，输出一个整数，表示 nP−1 i=0 [si,1 + si,2 + si,3 ≥ 2] 的最大值。

Example

standard input

```
3
10
0000001000
0000000110
0000000001
10
0000001000
0000010000
0000001100
10
0000000111
0000000011
0001000000
```

standard output

```
1
2
3
```

Note 对字符串 s 做一次循环左移：si → s(i−1+n) mod n。 例如字符串 ‘1234‘，一次循环左移后变成 ‘2341‘。

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

typedef complex<double> cd;
const double PI = acos(-1);
```

```
int next_power_of_two(int n) {
    int L = 1;
    while (L < n) L <<= 1;
    return L;
}

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;
        if (i < j)
            swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }
    if (invert) {
        for (cd & x : a)
            x /= n;
    }
}

vector<int> linear_correlation_fft(const vector<int> &x, const vector<int>
&y2, int n, int n2) {
    int len_conv = n + n2 - 1;
    int L = next_power_of_two(len_conv);
    vector<cd> x_pad(L, 0), y2_pad(L, 0);
    for (int i = 0; i < n; i++) {
        x_pad[i] = x[i];
    }
    for (int i = 0; i < n2; i++) {
        y2_pad[i] = y2[i];
    }
    fft(x_pad, false);
    fft(y2_pad, false);
    vector<cd> Z(L);
    for (int i = 0; i < L; i++) {
        Z[i] = conj(x_pad[i]) * y2_pad[i];
    }
    fft(Z, true);
```

```
    vector<int> result(n);
    for (int d = 0; d < n; d++) {
        result[d] = round(Z[d].real());
    }
    return result;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        vector<string> s(3);
        for (int i = 0; i < 3; i++) {
            cin >> s[i];
        }

        int ans = 0;
        for (int idx = 0; idx < 3; idx++) {
            int fixed1 = (idx+1)%3, fixed2 = (idx+2)%3;
            int rotate_idx = idx;
            int A = 0;
            vector<int> M(n, 0);
            for (int i = 0; i < n; i++) {
                if (s[fixed1][i]=='1' && s[fixed2][i]=='1') {
                    A++;
                } else if (s[fixed1][i]=='1' || s[fixed2][i]=='1') {
                    M[i] = 1;
                }
            }
            vector<int> T(n);
            for (int i = 0; i < n; i++) {
                T[i] = (s[rotate_idx][i] == '1') ? 1 : 0;
            }

            vector<int> F(n, 0);
            if (n <= 1000) {
                for (int d = 0; d < n; d++) {
                    for (int i = 0; i < n; i++) {
                        int j = (i + d) % n;
                        if (M[i] && T[j]) {
                            F[d]++;
                        }
                    }
                }
            } else {
                vector<int> y2 = T;
                y2.insert(y2.end(), T.begin(), T.end());
                vector<int> corr = linear_correlation_fft(M, y2, n, 2*n);
                for (int d = 0; d < n; d++) {
```

```
                    F[d] = corr[d];
                }
            }

            int max_val = 0;
            for (int d = 0; d < n; d++) {
                int total = A + F[d];
                if (total > max_val) {
                    max_val = total;
                }
            }
            if (max_val > ans) {
                ans = max_val;
            }
        }
        cout << ans << '\n';
    }

    return 0;
}
```

时域卷积等于频域点乘

计算几何

计算几何部分简介

计算几何是利用计算机建立数学模型来解决几何问题的学科。

注意： 在计算几何中，默认以下约定：

- 水平为 x 轴，向右为正。
- 竖直为 y 轴，向上为正。
- 使用弧度制。
- 逆时针为正。
- 除非题目要求无限精度（使用整型进行计算），否则 `double` 类型的精度通常足够。

1. 点

在平面直角坐标系下，点用坐标表示，例如点 (5,2)，点 (−1,0) 等。

我们通常记录其横纵坐标值。可以使用 pair 或结构体来记录。

```
struct point {
    double x, y;
};
```

在极坐标系下，点用极坐标表示，记录其极径和极角。

2. 向量

由于向量的坐标表示与点相同，因此可以像点一样存储向量（当然点不是向量）。

在极坐标系下，与点同理。

向量的运算

以下是 `point` 结构体中常用的向量运算重载：

```
struct point {
    double x, y;
    point(double _x, double _y) : x(_x), y(_y) {}
    // 判等：考虑浮点数精度
    bool operator==(const point &a) const {
        const double eps = 1e-9; // 浮点数比较的精度
        return (abs(x - a.x) <= eps && abs(y - a.y) <= eps);
    }

    // 小于：用于排序，先按 x 坐标排序，x 相等时按 y 坐标排序
    bool operator<(const point &a) const {
        const double eps = 1e-9;
        if (abs(x - a.x) <= eps) return y < a.y - eps;
        return x < a.x - eps;
    }

    // 大于：基于小于和等于的定义
    bool operator>(const point &a) const {
        return !(*this < a || *this == a);
    }

    // 向量加法
    point operator+(const point &a) const {
        return {x + a.x, y + a.y};
    }

    // 向量减法
    point operator-(const point &a) const {
        return {x - a.x, y - a.y};
    }

    // 向量取反
    point operator-() const {
        return {-x, -y};
    }

    // 向量与标量乘法
    point operator*(const double k) const {
        return {k * x, k * y};
    }
```

```
    // 向量与标量除法
    point operator/(const double k) const {
        return {x / k, y / k};
    }

    // 向量点积
    double operator*(const point &a) const { // 点积
        return x * a.x + y * a.y;
    }

    // 向量叉积
    double operator^(const point &a) const { // 叉积，注意优先级
        return x * a.y - y * a.x;
    }

    // 向量长度的平方
    double len2() const {
        return x * x + y * y;
    }

    // 向量长度
    double len() const {
        return sqrt(len2());
    }

    // 向量旋转（逆时针，给定角度）
    point rot(const long double rad) const {
        return {x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)};
    }

    // 向量旋转（逆时针，给定正弦余弦值）
    point rot(const long double cosr, const long double sinr) const {
        return {x * cosr - y * sinr, x * sinr + y * cosr};
    }

    // toleft 测试：一个点（向量）对另一个点（向量）的叉积正负性
    // 0: 共线, 1: 逆时针方向, -1: 顺时针方向
    int toleft(const point &a) const {
        const double eps = 1e-9;
        const auto t = (*this) ^ a;
        return (t > eps) - (t < -eps);
    }
};
```

叉积的性质： 表示两个向量表示的平行四边形的有向面积。如果结果为正，表示从第一个向量到第二个向量是逆时针方向；如果为负，表示顺时针方向；如果为零，表示两向量共线。

3. 直线与线段

直线

直线一般记录直线上一点和直线的方向向量。

```
struct line {
    point p, v; // p: 直线上一点, v: 方向向量

    // 直线交点
    // 假设两条直线不平行且不重合
    point inter(const line &a) const {
        // (a.v ^ (p - a.p)) / (v ^ a.v) 是参数 t，使得 p + t*v 为交点
        return p + v * ((a.v ^ (p - a.p)) / (v ^ a.v));
    }

    // 点到直线距离
    // abs(v ^ (a - p)) 是平行四边形面积，v.len() 是底边长
    long double dis(const point &a) const {
        return abs(v ^ (a - p)) / v.len();
    }

    // 点在直线上的投影
    // (v * (a - p)) / (v * v) 是参数 t，使得 p + t*v 为投影点
    point proj(const point &a) const {
        return p + v * ((v * (a - p)) / (v * v));
    }

    // 直线对另一点的 toleft 定义
    // 0: 点在直线上, 1: 点在直线左侧, -1: 点在直线右侧
    int toleft(const point &a) const {
        return v.toleft(a - p);
    }
};
```

线段

线段直接记录线段的两个端点。

```
struct segment {
    point a, b; // a, b: 线段的两个端点
    segment(point _a, point _b) : a(_a), b(_b) {}
    // 判断点是否在线段上
    // -1: 点在线段端点上 | 0: 点不在线段上 | 1: 点严格在线段上
    int is_on(const point &p) const {
        const double eps = 1e-9;
        if (p == a || p == b) return -1; // 在端点上
        // 判断共线且在两端点之间
        return (p - a).toleft(p - b) == 0 && (p - a) * (p - b) < -eps; //
严格在线段上
    }

    // 判断线段与直线是否相交
    // -1: 直线经过线段端点 | 0: 线段和直线不相交 | 1: 线段和直线严格相交
```

```
    int is_inter(const line &l) const {
        int t1 = l.toleft(a);
        int t2 = l.toleft(b);
        if (t1 == 0 || t2 == 0) return -1; // 直线经过线段端点
        if (t1 != t2) return 1; // 严格相交（端点在直线两侧）
        return 0; // 不相交
    }

    // 判断两条线段是否相交
    // -1: 在某一线的端点处相交 | 0: 两线段不相交 | 1: 两线段严格相交
    int is_inter(const segment &s) const {
        // 检查是否有端点在另一条线段上
        if (is_on(s.a) || is_on(s.b) || s.is_on(a) || s.is_on(b)) return
-1;

        // 将线段转换为直线进行叉积判断
        line l1 = {a, b - a};
        line l2 = {s.a, s.b - s.a};

        // 检查线段端点是否在另一条直线的不同侧
        // 如果 (l1.toleft(s.a) * l1.toleft(s.b) == -1) 表示 s 的两端点在 l1
的两侧
        // 如果 (l2.toleft(a) * l2.toleft(b) == -1) 表示 this 的两端点在 l2 的
两侧
        if (l1.toleft(s.a) * l1.toleft(s.b) == -1 && l2.toleft(a) *
l2.toleft(b) == -1) {
            return 1; // 严格相交
        }
        return 0; // 不相交
    }

    // 点到线段距离
    long double dis(const point &p) const {
        const double eps = 1e-9;
        // 如果投影在线段外部，则距离为到端点的最短距离
        if (((p - a) * (b - a) < -eps) || ((p - b) * (a - b) < -eps)) {
            return min(p.dis(a), p.dis(b)); // 到端点的距离
        }
        // 如果投影在线段内部，则距离为点到直线的距离
        line l = {a, b - a};
        return l.dis(p);
    }

    // 两线段间距离
    long double dis(const segment &s) const {
        if (is_inter(s)) return 0; // 如果相交，距离为0
        // 否则，距离是所有端点到另一条线段的距离的最小值
        return min({dis(s.a), dis(s.b), s.dis(a), s.dis(b)});
    }
};
```

4. 圆

圆显然使用圆心和半径即可表示。

```
struct Circle{
    point p;
    double r;
    Circle(): p(0,0), r(0) {}
    Circle(point _p,double _r): p(_p),r(_r) {}

    [[nodiscard]] int relationLine(line v) const{
       double dst = v.dis(p);
       if(dcmp(dst - r) < 0) return 2;
       if(dcmp(dst - r) == 0) return 1;
       return 0;
    }

    int pointCrossLine(line v, point &p1, point &p2) const{
        int rel = relationLine(v);
        if(rel == 0) return 0;

        point a = v.proj(p);

        double d_to_line = v.dis(p);
        double half_chord_len_sq = r * r - d_to_line * d_to_line;

        if (dcmp(half_chord_len_sq) < 0) half_chord_len_sq = 0;
        double half_chord_len = std::sqrt(half_chord_len_sq);

        if(dcmp(half_chord_len) == 0){
            p1 = a;
            p2 = a;
            return 1;
        }

        if (dcmp(v.v.len()) == 0) {
            return 0;
        }
        point unit_dir = v.v / v.v.len();

        p1 = a + unit_dir * half_chord_len;
        p2 = a - unit_dir * half_chord_len;
        return 2;
    }
};
```

5. toleft 函数

`toleft` 函数是一个非常实用的工具，它通过叉积的正负性来判断一个点（或向量）相对于另一个点（或向量）的方向关系。

- **点（向量）对点（向量）的 `toleft`：**

- 定义为两点（向量）叉积的正负性。
- 叉积为 0：共线。
- 叉积为正：逆时针方向。
- 叉积为负：顺时针方向。

- **直线对另一点的 `toleft`：**
  - 当点在直线上时为 0。
  - 当点在直线左侧时为 1。
  - 当点在直线右侧时为 -1。

```
// 假设 point 结构体和 operator^ (叉积) 已经定义
// 假设 line 结构体已经定义，并且包含 point p 和 point v (方向向量)

// 定义一个小的浮点数精度误差
const double eps = 1e-9;

struct point {
    double x, y;

    // 向量叉积 (假设已定义)
    double operator^(const point &a) const {
        return x * a.y - y * a.x;
    }

    /**
     * @brief toleft 测试：判断当前点（向量）相对于另一个点（向量）的方向关系
     * 通过叉积的正负性来判断
     * @param a 另一个点（向量）
     * @return int 0: 共线, 1: 逆时针方向, -1: 顺时针方向
     */
    int toleft(const point &a) const {
        // 计算当前向量和向量 a 的叉积
        const auto t = (*this) ^ a;
        // 根据叉积的正负性返回结果
        // t > eps 返回 1 (逆时针)
        // t < -eps 返回 -1 (顺时针)
        // 否则返回 0 (共线)
        return (t > eps) - (t < -eps);
    }
};

struct line {
    point p, v; // p: 直线上一点, v: 方向向量

    /**
     * @brief 直线对另一点的 toleft 定义
     * 判断点相对于直线的方向
```

```
     * @param a 待判断的点
     * @return int 0: 点在直线上, 1: 点在直线左侧, -1: 点在直线右侧
     */
    int toleft(const point &a) const {
        // 将点 a 转换为相对于直线起始点 p 的向量 (a - p)
        // 然后判断直线方向向量 v 和 (a - p) 的叉积
        return v.toleft(a - p);
    }
};
```

有了 `toleft` 函数，可以方便地判断许多几何关系。

6. 凸包

定义

在平面上能包含所有给定点的最小凸多边形叫做凸包。

实际上可以理解为用一个橡皮筋包含住所有给定点的形态。

Andrew 算法求凸包

性质：

该算法的时间复杂度为 O(NlogN)，其中 N 为待求凸包点集的大小。复杂度的瓶颈在于对所有点坐标的关键键字排序。

过程：

1. 首先把所有点以横坐标为第一关键字，纵坐标为第二关键字排序。
2. 显然排序后最小的元素和最大的元素一定在凸包上。而且因为是凸多边形，我们如果从一个点出发逆时针走，轨迹总是“左拐”的，一旦出现右拐，就说明这一段不在凸包上。因此我们可以用一个单调栈来维护上下凸壳。
3. 因为从左向右看，上下凸壳所旋转的方向不同，为了让单调栈起作用，我们首先**升序枚举**求出下凸壳，然后**降序**求出上凸壳。
4. 求凸壳时，一旦发现即将进栈的点 (P) 和栈顶的两个点 (S_1,S_2)，其中 S_1 为栈顶，行进的方向向右旋转，即叉积 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$，则弹出栈顶，回到上一步，继续检测，直到 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} \ge 0$ 或者栈内仅剩一个元素为止。
5. 通常情况下不需要保留位于凸包边上的点，因此上面一段中 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$这个条件中的 $<$ 可以视情况改为 $\le$，同时后面一个条件应改为$>$ 。

以下是 Andrew 算法的 C++ 实现：

```
struct Convex {
    std::vector<point> p; // 存储凸包上的点
```

```
};

// Andrew 算法求凸包
Convex convexhull(std::vector<point> p) {
    std::vector<point> st; // 单调栈
    if (p.size() <= 2) return Convex{p}; // 点数小于等于2，直接返回

    // 1. 按 x 坐标升序，x 相同按 y 坐标升序排序
    std::sort(p.begin(), p.end());

    // 辅助函数：判断是否需要弹出栈顶元素
    // back1: 栈顶元素, back2: 栈顶前一个元素, u: 待加入元素
    // 如果 (back1 - back2) 到 (u - back1) 是顺时针或共线，则需要弹出 back1
    auto check = [](const std::vector<point> &st_vec, const point &u) {
        const double eps = 1e-9;
        point back1 = st_vec.back();
        point back2 = *std::prev(st_vec.end(), 2);
        // 如果 (u - back1) 相对于 (back1 - back2) 是顺时针或共线（叉积 <= 0）
        return ((back1 - back2) ^ (u - back1)) <= eps;
    };

    // 2. 构建下凸壳
    for (const point &u : p) {
        // 保持栈内点的"左拐"性质，如果出现"右拐"或共线，弹出栈顶
        while (st.size() > 1 && check(st, u)) {
            st.pop_back();
        }
        st.push_back(u);
    }

    // 记录下凸壳的最后一个点（即整个点集的最大 x 坐标点）
    // 这个点在构建上凸壳时会作为起始点，但最终不应重复
    size_t k = st.size();
    st.pop_back(); // 弹出最后一个点，因为它会是上凸壳的起始点，避免重复

    // 3. 构建上凸壳
    // 逆序遍历排序后的点，从 x 最大的点开始
    std::reverse(p.begin(), p.end());
    for (const point &u : p) {
        // 保持栈内点的"左拐"性质（对于上凸壳而言），如果出现"右拐"或共线，弹出栈顶
        // 注意：这里的 check 函数判断的是 (back1 - back2) 到 (u - back1) 的叉积
        // 对于上凸壳，我们希望是顺时针或共线，所以条件依然是 <= 0
        while (st.size() > k && check(st, u)) { // st.size() > k 确保不弹出下
凸壳的起始点
            st.pop_back();
        }
        st.push_back(u);
    }
    st.pop_back(); // 弹出最后一个点，避免重复（与第一个点相同）

    return Convex{st};
}
```

P2742 [USACO5.1] 圈奶牛Fencing the Cows /【模板】二维凸包 - 洛谷

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <cstdio> // For printf and scanf

// 定义浮点数比较的精度
const double eps = 1e-9;

// 定义点/向量结构体
struct vec {
    double x, y;

    // 默认构造函数
    vec() : x(0), y(0) {}
    // 带参数构造函数
    vec(double _x, double _y) : x(_x), y(_y) {}

    // 运算符重载：用于排序
    // 先按 x 坐标升序，x 相同则按 y 坐标升序
    bool operator<(const vec& other) const {
        if (std::abs(x - other.x) > eps) { // x 坐标不相等
            return x < other.x;
        }
        return y < other.y - eps; // x 坐标相等，比较 y 坐标 (考虑精度，防止共线点被错误排序)
    }

    // 运算符重载：用于相等判断（考虑浮点数精度）
    bool operator==(const vec& other) const {
        return std::abs(x - other.x) < eps && std::abs(y - other.y) < eps;
    }
};

// 全局变量，用于存储输入点和凸包点
const int MAXN = 100005; // 最大点数
vec p[MAXN]; // 输入点数组 (1-indexed)
vec st[MAXN]; // 栈，用于存储凸包上的点 (1-indexed)
int n_global; // 输入点的数量，使用 n_global 避免与 Andrew 函数内部的 n 冲突

/**
 * @brief 计算两个向量的二维叉积
 * @param A 向量 A
 * @param B 向量 B
 * @return double 叉积结果。A -> B 左转为正，右转为负，共线为零。
 */
double Cross(vec A, vec B) {
    return A.x * B.y - A.y * B.x;
}

/**
```

```
 * @brief 判断点 p 相对于由 a 和 b 构成的向量 (a->b) 的方向
 * 实际上是计算向量 (b-a) 和 (p-a) 的叉积
 * @param a 次栈顶元素 (向量起点)
 * @param b 栈顶元素 (向量终点)
 * @param p 新增点
 * @return double 叉积结果。如果结果 > 0，p 在 (a->b) 的左侧；< 0 在右侧；= 0 共线。
 */
double Side(vec a, vec b, vec p) {
    // 向量 AB (b - a)
    vec AB = vec(b.x - a.x, b.y - a.y);
    // 向量 AP (p - a)
    vec AP = vec(p.x - a.x, p.y - a.y);
    // 计算 AB 和 AP 的叉积
    return Cross(AB, AP);
}

/**
 * @brief 计算两点之间的欧几里得距离
 * @param p1 点 1
 * @param p2 点 2
 * @return double 两点之间的距离
 */
double dist(vec p1, vec p2) {
    return std::sqrt(std::pow(p1.x - p2.x, 2) + std::pow(p1.y - p2.y, 2));
}

/**
 * @brief 使用 Andrew 算法计算点集的凸包
 * @param num_points 输入点的数量
 * @return int 凸包上的点的数量
 */
int Andrew(int num_points) {
    n_global = num_points; // 更新全局变量 n_global

    // 如果点数小于等于2，凸包就是这些点本身
    if (n_global <= 2) {
        // 对于 n=1 或 n=2 的情况，直接将点复制到栈中
        for (int i = 0; i < n_global; ++i) {
            st[i + 1] = p[i + 1]; // 将点复制到栈中 (1-indexed)
        }
        return n_global; // 凸包上的点数就是 n_global
    }

    // 1. 对所有点进行排序：先按 x 坐标升序，x 相同则按 y 坐标升序
    // p+1 到 p+n_global+1 是因为 p 数组是 1-indexed
    std::sort(p + 1, p + n_global + 1);

    int top = 0; // 栈顶指针，表示栈中元素的数量 (0-indexed for st array, but logic uses 1-indexed for st[top])

    // 2. 构建下凸包
    for (int i = 1; i <= n_global; ++i) {
        // 当栈中至少有两个点，并且新点 p[i] 使得 (st[top-1] -> st[top]) 到
```

```
(st[top] -> p[i]) 形成右转或共线时
        // (Side <= 0 表示右转或共线，需要弹出栈顶)
        // 使用 eps 进行浮点数比较，<= eps 意味着共线或右转
        while (top >= 2 && Side(st[top - 1], st[top], p[i]) <= eps) {
            top--;
        }
        st[++top] = p[i]; // 将新点压入栈
    }

    // 记录下凸包的最后一个点在栈中的位置 (不包括重复的最后一个点)
    // 因为上凸包会从 n_global-1 开始，避免重复计算最右边的点
    int lower_hull_size = top;

    // 3. 构建上凸包
    // 从倒数第二个点开始 (p[n_global-1]) 逆序遍历，直到第一个点 (p[1])
    // 注意：st[top] 此时是 p[n_global] (最右边的点)，它既是下凸包的终点，也是上凸包的起点
    // 我们需要从 p[n_global-1] 开始，将 p[n_global] 作为上凸包的第一个点，避免重复
    for (int i = n_global - 1; i >= 1; --i) {
        // 同样，当栈中至少有两个点，并且新点 p[i] 使得 (st[top-1] -> st[top]) 到 (st[top] -> p[i]) 形成右转或共线时
        // (Side <= 0 表示右转或共线，需要弹出栈顶)
        // 这里 top > lower_hull_size 确保不会弹出下凸包中的点，除了最右边的点
        while (top > lower_hull_size && Side(st[top - 1], st[top], p[i]) <= eps) { // 使用 eps 进行浮点数比较
            top--;
        }
        st[++top] = p[i]; // 将新点压入栈
    }

    // 最后一个点 (st[top]) 会是 p[1] 的重复，需要弹出
    // 确保凸包是闭合的，且没有重复的起始/结束点
    if (top > 1 && st[top] == st[1]) { // 如果栈顶和栈底相同（除了第一个点），则弹出栈顶
        top--;
    }

    return top; // 返回凸包上的点数
}


// --- 主函数 ---
int main() {
    // 优化输入输出
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);

    // 读取输入点的数量
    std::cin >> n_global;

    // 读取所有点
    for (int i = 1; i <= n_global; ++i) {
        std::cin >> p[i].x >> p[i].y;
    }
```

```
    // 计算凸包上的点数
    int convex_hull_size = Andrew(n_global);

    double perimeter = 0.0;
    // 如果凸包点数小于2，周长为0（单个点或两个点，无法形成围栏）
    if (convex_hull_size <= 1) {
        perimeter = 0.0;
    } else {
        // 计算凸包的周长
        for (int i = 1; i < convex_hull_size; ++i) {
            perimeter += dist(st[i], st[i+1]);
        }
        // 加上最后一个点到第一个点的距离，形成闭合围栏
        perimeter += dist(st[convex_hull_size], st[1]);
    }

    // 输出结果，四舍五入保留两位小数
    printf("%.2f\n", perimeter);

    return 0;
}
```

7. 例题与参考

基础例题

- Vjudge Contest 727825
    - 前两道是比较基础的题目，建议练习。

凸包例题

- 洛谷 P2742 圈奶牛 Fencing the Cows /【模板】二维凸包

其它计算几何主题

- 半平面交
- 旋转卡壳
- 平面最近点对

其它例题

- 牛客 ACM Contest 27249 C: 判断直线和线段交点
- 牛客 ACM Contest 27249 E: 模拟

深入研究题单

以下题单供参考，可用于深入学习和练习：

- **二维基础：** 牛客 ACM Contest 27249
- **极角序：** 牛客 ACM Contest 27675
- **凸包：** 牛客 ACM Contest 28513
- **凸包进阶：** 牛客 ACM Contest 28770
- **扫描线：** 牛客 ACM Contest 36088
- **半平面交：** 牛客 ACM Contest 29764
- **圆：** 牛客 ACM Contest 40275

8. 模板汇总

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

using point_t=long double;  //全局数据类型，可修改为 long long 等

constexpr point_t eps=1e-8;
constexpr long double PI=3.14159265358979323841;

// 点与向量
template<typename T> struct point
{
    T x,y;

    bool operator==(const point &a) const {return (abs(x-a.x)<=eps &&
abs(y-a.y)<=eps);}
    bool operator<(const point &a) const {if (abs(x-a.x)<=eps) return
y<a.y-eps; return x<a.x-eps;}
    bool operator>(const point &a) const {return !(*this<a || *this==a);}
    point operator+(const point &a) const {return {x+a.x,y+a.y};}
    point operator-(const point &a) const {return {x-a.x,y-a.y};}
    point operator-() const {return {-x,-y};}
    point operator*(const T k) const {return {k*x,k*y};}
    point operator/(const T k) const {return {x/k,y/k};}
    T operator*(const point &a) const {return x*a.x+y*a.y;}  // 点积
    T operator^(const point &a) const {return x*a.y-y*a.x;}  // 叉积，注意优
先级
    int toleft(const point &a) const {const auto t=(*this)^a; return
(t>eps)-(t<-eps);}  // to-left 测试
    T len2() const {return (*this)*(*this);}  // 向量长度的平方
    T dis2(const point &a) const {return (a-(*this)).len2();}  // 两点距离的
平方

    // 涉及浮点数
    long double len() const {return sqrtl(len2());}  // 向量长度
    long double dis(const point &a) const {return sqrtl(dis2(a));}  // 两点
```

```
距离
    long double ang(const point &a) const {return acosl(max(-1.0l,min(1.0l,
((*this)*a)/(len()*a.len())));}  // 向量夹角
    point rot(const long double rad) const {return {x*cos(rad)-
y*sin(rad),x*sin(rad)+y*cos(rad)};}  // 逆时针旋转（给定角度）
    point rot(const long double cosr,const long double sinr) const {return
{x*cosr-y*sinr,x*sinr+y*cosr};}  // 逆时针旋转（给定角度的正弦与余弦）
};

using Point=point<point_t>;

// 极角排序
struct argcmp
{
    bool operator()(const Point &a,const Point &b) const
    {
        const auto quad=[](const Point &a)
        {
            if (a.y<-eps) return 1;
            if (a.y>eps) return 4;
            if (a.x<-eps) return 5;
            if (a.x>eps) return 3;
            return 2;
        };
        const int qa=quad(a),qb=quad(b);
        if (qa!=qb) return qa<qb;
        const auto t=a^b;
        // if (abs(t)<=eps) return a*a<b*b-eps;  // 不同长度的向量需要分开
        return t>eps;
    }
};

// 直线
template<typename T> struct line
{
    point<T> p,v;  // p 为直线上一点，v 为方向向量

    bool operator==(const line &a) const {return v.toleft(a.v)==0 &&
v.toleft(p-a.p)==0;}
    int toleft(const point<T> &a) const {return v.toleft(a-p);}  // to-left
测试
    bool operator<(const line &a) const  // 半平面交算法定义的排序
    {
        if (abs(v^a.v)<=eps && v*a.v>=-eps) return toleft(a.p)==-1;
        return argcmp()(v,a.v);
    }

    // 涉及浮点数
    point<T> inter(const line &a) const {return p+v*((a.v^(p-
a.p))/(v^a.v));}  // 直线交点
    long double dis(const point<T> &a) const {return abs(v^(a-p))/v.len();}
// 点到直线距离
    point<T> proj(const point<T> &a) const {return p+v*((v*(a-p))/(v*v));}
// 点在直线上的投影
```

```
};

using Line=line<point_t>;

//线段
template<typename T> struct segment
{
    point<T> a,b;

    bool operator<(const segment &s) const {return make_pair(a,b)
<make_pair(s.a,s.b);}

    // 判定性函数建议在整数域使用

    // 判断点是否在线段上
    // -1 点在线段端点 | 0 点不在线段上 | 1 点严格在线段上
    int is_on(const point<T> &p) const
    {
        if (p==a || p==b) return -1;
        return (p-a).toleft(p-b)==0 && (p-a)*(p-b)<-eps;
    }

    // 判断线段直线是否相交
    // -1 直线经过线段端点 | 0 线段和直线不相交 | 1 线段和直线严格相交
    int is_inter(const line<T> &l) const
    {
        if (l.toleft(a)==0 || l.toleft(b)==0) return -1;
        return l.toleft(a)!=l.toleft(b);
    }

    // 判断两线段是否相交
    // -1 在某一线段端点处相交 | 0 两线段不相交 | 1 两线段严格相交
    int is_inter(const segment<T> &s) const
    {
        if (is_on(s.a) || is_on(s.b) || s.is_on(a) || s.is_on(b)) return
-1;
        const line<T> l{a,b-a},ls{s.a,s.b-s.a};
        return l.toleft(s.a)*l.toleft(s.b)==-1 &&
ls.toleft(a)*ls.toleft(b)==-1;
    }

    // 点到线段距离
    long double dis(const point<T> &p) const
    {
        if ((p-a)*(b-a)<-eps || (p-b)*(a-b)<-eps) return
min(p.dis(a),p.dis(b));
        const line<T> l{a,b-a};
        return l.dis(p);
    }

    // 两线段间距离
    long double dis(const segment<T> &s) const
    {
        if (is_inter(s)) return 0;
```

```
        return min({dis(s.a),dis(s.b),s.dis(a),s.dis(b)});
    }
};

using Segment=segment<point_t>;

// 多边形
template<typename T> struct polygon
{
    vector<point<T>> p;  // 以逆时针顺序存储

    size_t nxt(const size_t i) const {return i==p.size()-1?0:i+1;}
    size_t pre(const size_t i) const {return i==0?p.size()-1:i-1;}

    // 回转数
    // 返回值第一项表示点是否在多边形边上
    // 对于狭义多边形，回转数为 0 表示点在多边形外，否则点在多边形内
    pair<bool,int> winding(const point<T> &a) const
    {
        int cnt=0;
        for (size_t i=0;i<p.size();i++)
        {
            const point<T> u=p[i],v=p[nxt(i)];
            if (abs((a-u)^(a-v))<=eps && (a-u)*(a-v)<=eps) return {true,0};
            if (abs(u.y-v.y)<=eps) continue;
            const Line uv={u,v-u};
            if (u.y<v.y-eps && uv.toleft(a)<=0) continue;
            if (u.y>v.y+eps && uv.toleft(a)>=0) continue;
            if (u.y<a.y-eps && v.y>=a.y-eps) cnt++;
            if (u.y>=a.y-eps && v.y<a.y-eps) cnt--;
        }
        return {false,cnt};
    }

    // 多边形面积的两倍
    // 可用于判断点的存储顺序是顺时针或逆时针
    T area() const
    {
        T sum=0;
        for (size_t i=0;i<p.size();i++) sum+=p[i]^p[nxt(i)];
        return sum;
    }

    // 多边形的周长
    long double circ() const
    {
        long double sum=0;
        for (size_t i=0;i<p.size();i++) sum+=p[i].dis(p[nxt(i)]);
        return sum;
    }
};

using Polygon=polygon<point_t>;
```

```
//凸多边形
template<typename T> struct convex: polygon<T>
{
    // 闵可夫斯基和
    convex operator+(const convex &c) const
    {
        const auto &p=this->p;
        vector<Segment>
e1(p.size()),e2(c.p.size()),edge(p.size()+c.p.size());
        vector<point<T>> res; res.reserve(p.size()+c.p.size());
        const auto cmp=[](const Segment &u,const Segment &v) {return
argcmp()(u.b-u.a,v.b-v.a);};
        for (size_t i=0;i<p.size();i++) e1[i]={p[i],p[this->nxt(i)]};
        for (size_t i=0;i<c.p.size();i++) e2[i]={c.p[i],c.p[c.nxt(i)]};
        rotate(e1.begin(),min_element(e1.begin(),e1.end(),cmp),e1.end());
        rotate(e2.begin(),min_element(e2.begin(),e2.end(),cmp),e2.end());
        merge(e1.begin(),e1.end(),e2.begin(),e2.end(),edge.begin(),cmp);
        const auto check=[](const vector<point<T>> &res,const point<T> &u)
        {
            const auto back1=res.back(),back2=*prev(res.end(),2);
            return (back1-back2).toleft(u-back1)==0 && (back1-back2)*(u-
back1)>=-eps;
        };
        auto u=e1[0].a+e2[0].a;
        for (const auto &v:edge)
        {
            while (res.size()>1 && check(res,u)) res.pop_back();
            res.push_back(u);
            u=u+v.b-v.a;
        }
        if (res.size()>1 && check(res,res[0])) res.pop_back();
        return {res};
    }

    // 旋转卡壳
    // func 为更新答案的函数，可以根据题目调整位置
    template<typename F> void rotcaliper(const F &func) const
    {
        const auto &p=this->p;
        const auto area=[](const point<T> &u,const point<T> &v,const
point<T> &w){return (w-u)^(w-v);};
        for (size_t i=0,j=1;i<p.size();i++)
        {
            const auto nxti=this->nxt(i);
            func(p[i],p[nxti],p[j]);
            while (area(p[this-
>nxt(j)],p[i],p[nxti])>=area(p[j],p[i],p[nxti]))
            {
                j=this->nxt(j);
                func(p[i],p[nxti],p[j]);
            }
        }
    }
```

```
    // 凸多边形的直径的平方
    T diameter2() const
    {
        const auto &p=this->p;
        if (p.size()==1) return 0;
        if (p.size()==2) return p[0].dis2(p[1]);
        T ans=0;
        auto func=[&](const point<T> &u,const point<T> &v,const point<T>
&w){ans=max({ans,w.dis2(u),w.dis2(v)});};
        rotcaliper(func);
        return ans;
    }

    // 判断点是否在凸多边形内
    // 复杂度 O(logn)
    // -1 点在多边形边上 | 0 点在多边形外 | 1 点在多边形内
    int is_in(const point<T> &a) const
    {
        const auto &p=this->p;
        if (p.size()==1) return a==p[0]?-1:0;
        if (p.size()==2) return segment<T>{p[0],p[1]}.is_on(a)?-1:0;
        if (a==p[0]) return -1;
        if ((p[1]-p[0]).toleft(a-p[0])==-1 || (p.back()-p[0]).toleft(a-
p[0])==1) return 0;
        const auto cmp=[&](const Point &u,const Point &v){return (u-
p[0]).toleft(v-p[0])==1;};
        const size_t i=lower_bound(p.begin()+1,p.end(),a,cmp)-p.begin();
        if (i==1) return segment<T>{p[0],p[i]}.is_on(a)?-1:0;
        if (i==p.size()-1 && segment<T>{p[0],p[i]}.is_on(a)) return -1;
        if (segment<T>{p[i-1],p[i]}.is_on(a)) return -1;
        return (p[i]-p[i-1]).toleft(a-p[i-1])>0;
    }

    // 凸多边形关于某一方向的极点
    // 复杂度 O(logn)
    // 参考资料：https://codeforces.com/blog/entry/48868
    template<typename F> size_t extreme(const F &dir) const
    {
        const auto &p=this->p;
        const auto check=[&](const size_t i){return
dir(p[i]).toleft(p[this->nxt(i)]-p[i])>=0;};
        const auto dir0=dir(p[0]); const auto check0=check(0);
        if (!check0 && check(p.size()-1)) return 0;
        const auto cmp=[&](const Point &v)
        {
            const size_t vi=&v-p.data();
            if (vi==0) return 1;
            const auto checkv=check(vi);
            const auto t=dir0.toleft(v-p[0]);
            if (vi==1 && checkv==check0 && t==0) return 1;
            return checkv^(checkv==check0 && t<=0);
        };
        return partition_point(p.begin(),p.end(),cmp)-p.begin();
    }
```

```
    // 过凸多边形外一点求凸多边形的切线，返回切点下标
    // 复杂度 O(logn)
    // 必须保证点在多边形外
    pair<size_t,size_t> tangent(const point<T> &a) const
    {
        const size_t i=extreme([&](const point<T> &u){return u-a;});
        const size_t j=extreme([&](const point<T> &u){return a-u;});
        return {i,j};
    }

    // 求平行于给定直线的凸多边形的切线，返回切点下标
    // 复杂度 O(logn)
    pair<size_t,size_t> tangent(const line<T> &a) const
    {
        const size_t i=extreme([&](...){return a.v;});
        const size_t j=extreme([&](...){return -a.v;});
        return {i,j};
    }
};

using Convex=convex<point_t>;

// 圆
struct Circle
{
    Point c;
    long double r;

    bool operator==(const Circle &a) const {return c==a.c && abs(r-a.r)
<=eps;}
    long double circ() const {return 2*PI*r;}  // 周长
    long double area() const {return PI*r*r;}  // 面积

    // 点与圆的关系
    // -1 圆上 | 0 圆外 | 1 圆内
    int is_in(const Point &p) const {const long double d=p.dis(c); return
abs(d-r)<=eps?-1:d<r-eps;}

    // 直线与圆关系
    // 0 相离 | 1 相切 | 2 相交
    int relation(const Line &l) const
    {
        const long double d=l.dis(c);
        if (d>r+eps) return 0;
        if (abs(d-r)<=eps) return 1;
        return 2;
    }

    // 圆与圆关系
    // -1 相同 | 0 相离 | 1 外切 | 2 相交 | 3 内切 | 4 内含
    int relation(const Circle &a) const
    {
        if (*this==a) return -1;
```

```
        const long double d=c.dis(a.c);
        if (d>r+a.r+eps) return 0;
        if (abs(d-r-a.r)<=eps) return 1;
        if (abs(d-abs(r-a.r))<=eps) return 3;
        if (d<abs(r-a.r)-eps) return 4;
        return 2;
    }

    // 直线与圆的交点
    vector<Point> inter(const Line &l) const
    {
        const long double d=l.dis(c);
        const Point p=l.proj(c);
        const int t=relation(l);
        if (t==0) return vector<Point>();
        if (t==1) return vector<Point>{p};
        const long double k=sqrt(r*r-d*d);
        return vector<Point>{p-(l.v/l.v.len())*k,p+(l.v/l.v.len())*k};
    }

    // 圆与圆交点
    vector<Point> inter(const Circle &a) const
    {
        const long double d=c.dis(a.c);
        const int t=relation(a);
        if (t==-1 || t==0 || t==4) return vector<Point>();
        Point e=a.c-c; e=e/e.len()*r;
        if (t==1 || t==3)
        {
            if (r*r+d*d-a.r*a.r>=-eps) return vector<Point>{c+e};
            return vector<Point>{c-e};
        }
        const long double costh=(r*r+d*d-a.r*a.r)/(2*r*d),sinth=sqrt(1-
costh*costh);
        return vector<Point>{c+e.rot(costh,-sinth),c+e.rot(costh,sinth)};
    }

    // 圆与圆交面积
    long double inter_area(const Circle &a) const
    {
        const long double d=c.dis(a.c);
        const int t=relation(a);
        if (t==-1) return area();
        if (t<2) return 0;
        if (t>2) return min(area(),a.area());
        const long double costh1=(r*r+d*d-a.r*a.r)/(2*r*d),costh2=
(a.r*a.r+d*d-r*r)/(2*a.r*d);
        const long double sinth1=sqrt(1-costh1*costh1),sinth2=sqrt(1-
costh2*costh2);
        const long double th1=acos(costh1),th2=acos(costh2);
        return r*r*(th1-costh1*sinth1)+a.r*a.r*(th2-costh2*sinth2);
    }

    // 过圆外一点圆的切线
```

```
    vector<Line> tangent(const Point &a) const
    {
        const int t=is_in(a);
        if (t==1) return vector<Line>();
        if (t==-1)
        {
            const Point v={-(a-c).y,(a-c).x};
            return vector<Line>{{a,v}};
        }
        Point e=a-c; e=e/e.len()*r;
        const long double costh=r/c.dis(a),sinth=sqrt(1-costh*costh);
        const Point t1=c+e.rot(costh,-sinth),t2=c+e.rot(costh,sinth);
        return vector<Line>{{a,t1-a},{a,t2-a}};
    }

    // 两圆的公切线
    vector<Line> tangent(const Circle &a) const
    {
        const int t=relation(a);
        vector<Line> lines;
        if (t==-1 || t==4) return lines;
        if (t==1 || t==3)
        {
            const Point p=inter(a)[0],v={-(a.c-c).y,(a.c-c).x};
            lines.push_back({p,v});
        }
        const long double d=c.dis(a.c);
        const Point e=(a.c-c)/(a.c-c).len();
        if (t<=2)
        {
            const long double costh=(r-a.r)/d,sinth=sqrt(1-costh*costh);
            const Point d1=e.rot(costh,-sinth),d2=e.rot(costh,sinth);
            const Point u1=c+d1*r,u2=c+d2*r,v1=a.c+d1*a.r,v2=a.c+d2*a.r;
            lines.push_back({u1,v1-u1}); lines.push_back({u2,v2-u2});
        }
        if (t==0)
        {
            const long double costh=(r+a.r)/d,sinth=sqrt(1-costh*costh);
            const Point d1=e.rot(costh,-sinth),d2=e.rot(costh,sinth);
            const Point u1=c+d1*r,u2=c+d2*r,v1=a.c-d1*a.r,v2=a.c-d2*a.r;
            lines.push_back({u1,v1-u1}); lines.push_back({u2,v2-u2});
        }
        return lines;
    }

    // 圆的反演
    tuple<int,Circle,Line> inverse(const Line &l) const
    {
        const Circle null_c={{0.0,0.0},0.0};
        const Line null_l={{0.0,0.0},{0.0,0.0}};
        if (l.toleft(c)==0) return {2,null_c,l};
        const Point v=l.toleft(c)==1?Point{l.v.y,-l.v.x}:Point{-
l.v.y,l.v.x};
        const long double d=r*r/l.dis(c);
```

```
        const Point p=c+v/v.len()*d;
        return {1,{(c+p)/2,d/2},null_l};
    }

    tuple<int,Circle,Line> inverse(const Circle &a) const
    {
        const Circle null_c={{0.0,0.0},0.0};
        const Line null_l={{0.0,0.0},{0.0,0.0}};
        const Point v=a.c-c;
        if (a.is_in(c)==-1)
        {
            const long double d=r*r/(a.r+a.r);
            const Point p=c+v/v.len()*d;
            return {2,null_c,{p,{-v.y,v.x}}};
        }
        if (c==a.c) return {1,{c,r*r/a.r},null_l};
        const long double d1=r*r/(c.dis(a.c)-a.r),d2=r*r/(c.dis(a.c)+a.r);
        const Point p=c+v/v.len()*d1,q=c+v/v.len()*d2;
        return {1,{(p+q)/2,p.dis(q)/2},null_l};
    }
};

// 圆与多边形面积交
long double area_inter(const Circle &circ,const Polygon &poly)
{
    const auto cal=[](const Circle &circ,const Point &a,const Point &b)
    {
        if ((a-circ.c).toleft(b-circ.c)==0) return 0.01;
        const auto ina=circ.is_in(a),inb=circ.is_in(b);
        const Line ab={a,b-a};
        if (ina && inb) return ((a-circ.c)^(b-circ.c))/2;
        if (ina && !inb)
        {
            const auto t=circ.inter(ab);
            const Point p=t.size()==1?t[0]:t[1];
            const long double ans=((a-circ.c)^(p-circ.c))/2;
            const long double th=(p-circ.c).ang(b-circ.c);
            const long double d=circ.r*circ.r*th/2;
            if ((a-circ.c).toleft(b-circ.c)==1) return ans+d;
            return ans-d;
        }
        if (!ina && inb)
        {
            const Point p=circ.inter(ab)[0];
            const long double ans=((p-circ.c)^(b-circ.c))/2;
            const long double th=(a-circ.c).ang(p-circ.c);
            const long double d=circ.r*circ.r*th/2;
            if ((a-circ.c).toleft(b-circ.c)==1) return ans+d;
            return ans-d;
        }
        const auto p=circ.inter(ab);
        if (p.size()==2 && Segment{a,b}.dis(circ.c)<=circ.r+eps)
        {
            const long double ans=((p[0]-circ.c)^(p[1]-circ.c))/2;
```

```
            const long double th1=(a-circ.c).ang(p[0]-circ.c),th2=(b-
circ.c).ang(p[1]-circ.c);
            const long double
d1=circ.r*circ.r*th1/2,d2=circ.r*circ.r*th2/2;
            if ((a-circ.c).toleft(b-circ.c)==1) return ans+d1+d2;
            return ans-d1-d2;
        }
        const long double th=(a-circ.c).ang(b-circ.c);
        if ((a-circ.c).toleft(b-circ.c)==1) return circ.r*circ.r*th/2;
        return -circ.r*circ.r*th/2;
    };

    long double ans=0;
    for (size_t i=0;i<poly.p.size();i++)
    {
        const Point a=poly.p[i],b=poly.p[poly.nxt(i)];
        ans+=cal(circ,a,b);
    }
    return ans;
}

// 点集的凸包
// Andrew 算法，复杂度 O(nlogn)
Convex convexhull(vector<Point> p)
{
    vector<Point> st;
    if (p.empty()) return Convex{st};
    sort(p.begin(),p.end());
    const auto check=[](const vector<Point> &st,const Point &u)
    {
        const auto back1=st.back(),back2=*prev(st.end(),2);
        return (back1-back2).toleft(u-back1)<=0;
    };
    for (const Point &u:p)
    {
        while (st.size()>1 && check(st,u)) st.pop_back();
        st.push_back(u);
    }
    size_t k=st.size();
    p.pop_back(); reverse(p.begin(),p.end());
    for (const Point &u:p)
    {
        while (st.size()>k && check(st,u)) st.pop_back();
        st.push_back(u);
    }
    st.pop_back();
    return Convex{st};
}

// 半平面交
// 排序增量法，复杂度 O(nlogn)
// 输入与返回值都是用直线表示的半平面集合
vector<Line> halfinter(vector<Line> l, const point_t lim=1e9)
{
```

```
    const auto check=[](const Line &a,const Line &b,const Line &c){return
a.toleft(b.inter(c))<0;};
    // 无精度误差的方法，但注意取值范围会扩大到三次方
    /*const auto check=[](const Line &a,const Line &b,const Line &c)
    {
        const Point p=a.v*(b.v^c.v),q=b.p*(b.v^c.v)+b.v*(c.v^(b.p-c.p))-
a.p*(b.v^c.v);
        return p.toleft(q)<0;
    };*/
    l.push_back({{-lim,0},{0,-1}}); l.push_back({{0,-lim},{1,0}});
    l.push_back({{lim,0},{0,1}}); l.push_back({{0,lim},{-1,0}});
    sort(l.begin(),l.end());
    deque<Line> q;
    for (size_t i=0;i<l.size();i++)
    {
        if (i>0 && l[i-1].v.toleft(l[i].v)==0 && l[i-1].v*l[i].v>eps)
continue;
        while (q.size()>1 && check(l[i],q.back(),q[q.size()-2]))
q.pop_back();
        while (q.size()>1 && check(l[i],q[0],q[1])) q.pop_front();
        if (!q.empty() && q.back().v.toleft(l[i].v)<=0) return vector<Line>
();
        q.push_back(l[i]);
    }
    while (q.size()>1 && check(q[0],q.back(),q[q.size()-2])) q.pop_back();
    while (q.size()>1 && check(q.back(),q[0],q[1])) q.pop_front();
    return vector<Line>(q.begin(),q.end());
}

// 点集形成的最小最大三角形
// 极角序扫描线，复杂度 O(n^2logn)
// 最大三角形问题可以使用凸包与旋转卡壳做到 O(n^2)
pair<point_t,point_t> minmax_triangle(const vector<Point> &vec)
{
    if (vec.size()<=2) return {0,0};
    vector<pair<int,int>> evt;
    evt.reserve(vec.size()*vec.size());
    point_t maxans=0,minans=numeric_limits<point_t>::max();
    for (size_t i=0;i<vec.size();i++)
    {
        for (size_t j=0;j<vec.size();j++)
        {
            if (i==j) continue;
            if (vec[i]==vec[j]) minans=0;
            else evt.push_back({i,j});
        }
    }
    sort(evt.begin(),evt.end(),[&](const pair<int,int> &u,const
pair<int,int> &v)
    {
        const Point du=vec[u.second]-vec[u.first],dv=vec[v.second]-
vec[v.first];
        return argcmp()({du.y,-du.x},{dv.y,-dv.x});
    });
```

```
    vector<size_t> vx(vec.size()),pos(vec.size());
    for (size_t i=0;i<vec.size();i++) vx[i]=i;
    sort(vx.begin(),vx.end(),[&](int x,int y){return vec[x]<vec[y];});
    for (size_t i=0;i<vx.size();i++) pos[vx[i]]=i;
    for (auto [u,v]:evt)
    {
        const size_t i=pos[u],j=pos[v];
        const size_t l=min(i,j),r=max(i,j);
        const Point vecu=vec[u],vecv=vec[v];
        if (l>0) minans=min(minans,abs((vec[vx[l-1]]-vecu)^(vec[vx[l-1]]-
vecv)));
        if (r<vx.size()-1) minans=min(minans,abs((vec[vx[r+1]]-
vecu)^(vec[vx[r+1]]-vecv)));
        maxans=max({maxans,abs((vec[vx[0]]-vecu)^(vec[vx[0]]-
vecv)),abs((vec[vx.back()]-vecu)^(vec[vx.back()]-vecv))});
        if (i<j) swap(vx[i],vx[j]),pos[u]=j,pos[v]=i;
    }
    return {minans,maxans};
}

// 判断多条线段是否有交点
// 扫描线，复杂度 O(nlogn)
bool segs_inter(const vector<Segment> &segs)
{
    if (segs.empty()) return false;
    using seq_t=tuple<point_t,int,Segment>;
    const auto seqcmp=[](const seq_t &u, const seq_t &v)
    {
        const auto [u0,u1,u2]=u;
        const auto [v0,v1,v2]=v;
        if (abs(u0-v0)<=eps) return make_pair(u1,u2)<make_pair(v1,v2);
        return u0<v0-eps;
    };
    vector<seq_t> seq;
    for (auto seg:segs)
    {
        if (seg.a.x>seg.b.x+eps) swap(seg.a,seg.b);
        seq.push_back({seg.a.x,0,seg});
        seq.push_back({seg.b.x,1,seg});
    }
    sort(seq.begin(),seq.end(),seqcmp);
    point_t x_now;
    auto cmp=[&](const Segment &u, const Segment &v)
    {
        if (abs(u.a.x-u.b.x)<=eps || abs(v.a.x-v.b.x)<=eps) return
u.a.y<v.a.y-eps;
        return ((x_now-u.a.x)*(u.b.y-u.a.y)+u.a.y*(u.b.x-u.a.x))*(v.b.x-
v.a.x)<((x_now-v.a.x)*(v.b.y-v.a.y)+v.a.y*(v.b.x-v.a.x))*(u.b.x-u.a.x)-eps;
    };
    multiset<Segment,decltype(cmp)> s{cmp};
    for (const auto [x,o,seg]:seq)
    {
        x_now=x;
        const auto it=s.lower_bound(seg);
```

```
        if (o==0)
        {
            if (it!=s.end() && seg.is_inter(*it)) return true;
            if (it!=s.begin() && seg.is_inter(*prev(it))) return true;
            s.insert(seg);
        }
        else
        {
            if (next(it)!=s.end() && it!=s.begin() &&
(*prev(it)).is_inter(*next(it))) return true;
            s.erase(it);
        }
    }
    return false;
}

// 多边形面积并
// 轮廓积分，复杂度 O(n^2logn)，n为边数
// ans[i] 表示被至少覆盖了 i+1 次的区域的面积
vector<long double> area_union(const vector<Polygon> &polys)
{
    const size_t siz=polys.size();
    vector<vector<pair<Point,Point>>> segs(siz);
    const auto check=[](const Point &u,const Segment &e){return !((u<e.a &&
u<e.b) || (u>e.a && u>e.b));};

    auto cut_edge=[&](const Segment &e,const size_t i)
    {
        const Line le{e.a,e.b-e.a};
        vector<pair<Point,int>> evt;
        evt.push_back({e.a,0}); evt.push_back({e.b,0});
        for (size_t j=0;j<polys.size();j++)
        {
            if (i==j) continue;
            const auto &pj=polys[j];
            for (size_t k=0;k<pj.p.size();k++)
            {
                const Segment s={pj.p[k],pj.p[pj.nxt(k)]};
                if (le.toleft(s.a)==0 && le.toleft(s.b)==0)
                {
                    evt.push_back({s.a,0});
                    evt.push_back({s.b,0});
                }
                else if (s.is_inter(le))
                {
                    const Line ls{s.a,s.b-s.a};
                    const Point u=le.inter(ls);
                    if (le.toleft(s.a)<0 && le.toleft(s.b)>=0)
evt.push_back({u,-1});
                    else if (le.toleft(s.a)>=0 && le.toleft(s.b)<0)
evt.push_back({u,1});
                }
            }
        }
```

```
        sort(evt.begin(),evt.end());
        if (e.a>e.b) reverse(evt.begin(),evt.end());
        int sum=0;
        for (size_t i=0;i<evt.size();i++)
        {
            sum+=evt[i].second;
            const Point u=evt[i].first,v=evt[i+1].first;
            if (!(u==v) && check(u,e) && check(v,e))
segs[sum].push_back({u,v});
            if (v==e.b) break;
        }
    };

    for (size_t i=0;i<polys.size();i++)
    {
        const auto &pi=polys[i];
        for (size_t k=0;k<pi.p.size();k++)
        {
            const Segment ei={pi.p[k],pi.p[pi.nxt(k)]};
            cut_edge(ei,i);
        }
    }
    vector<long double> ans(siz);
    for (size_t i=0;i<siz;i++)
    {
        long double sum=0;
        sort(segs[i].begin(),segs[i].end());
        int cnt=0;
        for (size_t j=0;j<segs[i].size();j++)
        {
            if (j>0 && segs[i][j]==segs[i][j-1]) segs[i+
(++cnt)].push_back(segs[i][j]);
            else cnt=0,sum+=segs[i][j].first^segs[i][j].second;
        }
        ans[i]=sum/2;
    }
    return ans;
}

// 圆面积并
// 轮廓积分，复杂度 O(n^2logn)
// ans[i] 表示被至少覆盖了 i+1 次的区域的面积
vector<long double> area_union(const vector<Circle> &circs)
{
    const size_t siz=circs.size();
    using arc_t=tuple<Point,long double,long double,long double>;
    vector<vector<arc_t>> arcs(siz);
    const auto eq=[](const arc_t &u,const arc_t &v)
    {
        const auto [u1,u2,u3,u4]=u;
        const auto [v1,v2,v3,v4]=v;
        return u1==v1 && abs(u2-v2)<=eps && abs(u3-v3)<=eps && abs(u4-v4)
<=eps;
    };
```

```
    auto cut_circ=[&](const Circle &ci,const size_t i)
    {
        vector<pair<long double,int>> evt;
        evt.push_back({-PI,0}); evt.push_back({PI,0});
        int init=0;
        for (size_t j=0;j<circs.size();j++)
        {
            if (i==j) continue;
            const Circle &cj=circs[j];
            if (ci.r<cj.r-eps && ci.relation(cj)>=3) init++;
            const auto inters=ci.inter(cj);
            if (inters.size()==1) evt.push_back({atan2l((inters[0]-ci.c).y,
(inters[0]-ci.c).x),0});
            if (inters.size()==2)
            {
                const Point dl=inters[0]-ci.c,dr=inters[1]-ci.c;
                long double argl=atan2l(dl.y,dl.x),argr=atan2l(dr.y,dr.x);
                if (abs(argl+PI)<=eps) argl=PI;
                if (abs(argr+PI)<=eps) argr=PI;
                if (argl>argr+eps)
                {
                    evt.push_back({argl,1}); evt.push_back({PI,-1});
                    evt.push_back({-PI,1}); evt.push_back({argr,-1});
                }
                else
                {
                    evt.push_back({argl,1});
                    evt.push_back({argr,-1});
                }
            }
        }
        sort(evt.begin(),evt.end());
        int sum=init;
        for (size_t i=0;i<evt.size();i++)
        {
            sum+=evt[i].second;
            if (abs(evt[i].first-evt[i+1].first)>eps)
arcs[sum].push_back({ci.c,ci.r,evt[i].first,evt[i+1].first});
            if (abs(evt[i+1].first-PI)<=eps) break;
        }
    };

    const auto oint=[](const arc_t &arc)
    {
        const auto [cc,cr,l,r]=arc;
        if (abs(r-l-PI-PI)<=eps) return 2.0l*PI*cr*cr;
        return cr*cr*(r-l)+cc.x*cr*(sin(r)-sin(l))-cc.y*cr*(cos(r)-cos(l));
    };

    for (size_t i=0;i<circs.size();i++)
    {
        const auto &ci=circs[i];
        cut_circ(ci,i);
```

```
    }
    vector<long double> ans(siz);
    for (size_t i=0;i<siz;i++)
    {
        long double sum=0;
        sort(arcs[i].begin(),arcs[i].end());
        int cnt=0;
        for (size_t j=0;j<arcs[i].size();j++)
        {
            if (j>0 && eq(arcs[i][j],arcs[i][j-1])) arcs[i+
(++cnt)].push_back(arcs[i][j]);
            else cnt=0,sum+=oint(arcs[i][j]);
        }
        ans[i]=sum/2;
    }
    return ans;
}
```