Geometria Computacional

Envoltório convexo: problemas resolvidos

Prof. Edson Alves

2019

Faculdade UnB Gama

Sumário

1. URI 1464 – Camadas de Cebola

2. UVA 10652 – Board Wrapping

URI 1464 – Camadas de Cebola

Problema

Dr. Kabal, um reconhecido biólogo, recentemente descobriu um líquido que é capaz de curar as mais avançadas doenças. O líquido é extraído de uma cebola muito rara que pode ser encontrada em um país chamado Cebolândia. Mas nem todas cebolas de Cebolândia são apropriadas para se levar ao laboratório para processamento. Somente cebolas com um numero ímpar de camadas contém o líquido milagroso. Isto é uma descoberta ímpar!

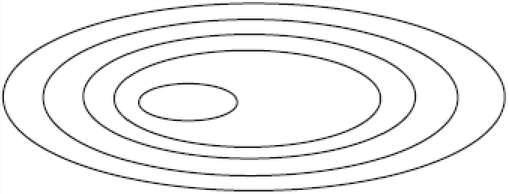


Figura 1: Cebola de Cebolândia

Problema

Dr. Kabal contratou muitos assistentes de pesquisa para coletar e analisar cebolas para ele. Como ele não quer compartilhar sua descoberta com o mundo ainda, ele não disse para os assistentes procurarem por cebolas com um numero ímpar de camadas. Ao invés disso, a cada assistente foi dada a tarefa de coletar cebolas, e selecionar pontos de cada uma das beiradas da camada mais externa, isso dá uma aproximação da estrutura de camadas da cebola que pode ser reconstruída depois. Dr. Kabal disse aos assistentes que o próximo passo seria a "análise complicada"desses pontos. De fato, tudo que eles farão é usar os pontos para contar o número de camadas em cada uma das cebolas, e selecionar aquelas com um número ímpar de camadas.

Problema

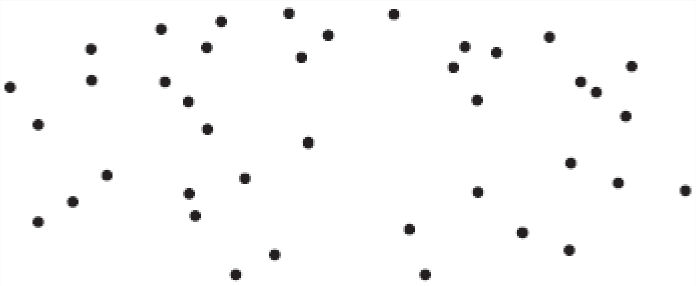


Figura 2: Pontos coletados por um assistente

Problema

É claro que a aproximação obtida por Dr. Kabal, dos pontos coletados, pode ter uma aparência diferente da cebola original. Por exemplo, somente alguns pontos da cebola mostrada na figura 1 podem ser extraídos no processo, dando origem a um conjunto de pontos como mostrado na figura 2. Com estes pontos Dr. Kabal tentará aproximar as camadas originais da cebola, obtendo algo como mostrado na figura 3. O procedimento de aproximação seguido pelo Dr. Kabal (cujo resultado é mostrado na figura 3) é simplesmente recursivamente encontrar polígonos convexos aninhados tais que no fim todo ponto pertencerá a um dos polígonos. Os assistentes foram informados para selecionar pontos de tal forma que o número de camadas na aproximação, se feita desta forma recursiva, seja o mesmo que na cebola original, o que é bom para o Dr. Kabal. Os assistentes também estão cientes de que eles precisam de pelo menos três pontos para aproximar uma camada, mesmo as internas.

Problema

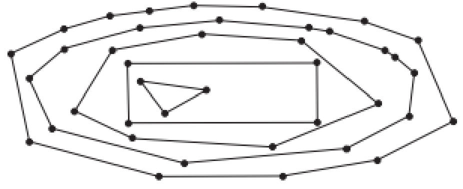


Figura 3: Aproximação do Dr. Kabal

Sua tarefa é escrever um programa que, dado o conjunto de pontos coletado pelo assistente (como mostrado na figura 2), determine se a respectiva cebola pode ser levada para o laboratório.

Entrada e saída

Entrada

A entrada contém vários casos de teste. Cada caso de teste consiste de um inteiro $3 \leq N \leq 2000$ em uma linha simples, indicando o número de pontos coletados pelo assistente. A seguir, haverão N linhas, cada uma contendo dois inteiros $-2000 \leq X, Y \leq 2000$ correspondendo às coordenadas de cada ponto. A entrada terminará com $N = 0$, que não deve ser processado.

Entrada e saída

Saída

Deverá haver uma linha de saída para cada caso de teste na entrada. Para cada caso de teste imprima a string

Take this onion to the lab!

se a cebola deve ser levada para o laboratório ou

Do not take this onion to the lab!

se a cebola não deve ser levada para o laboratório.

Exemplo de entradas e saídas

Exemplo de Entrada

```
7
0 0
0 8
1 6
3 1
6 6
8 0
8 8
11
2 6
3 2
6 6
0 0
0 11
1 1
1 9
7 1
7 9
8 10
8 0
0
```

Exemplo de Saída

```
Do not take this onion to the lab!
Take this onion to the lab!
```

Solução

- O problema consiste em computar o envoltório convexo do conjunto de pontos, excluir os pontos identificados, e reiniciar a rotina
- A resposta dependerá da paridade de número de envoltórios computados
- Para esta solução é preciso adaptar o algoritmo de Andrew
- Primeiramente, os pontos que tem orientação igual a zero devem entrar no envoltório
- Os pontos devem ser armazenados em um `set`, o que permite a exclusão com complexidade $O(N)$
- No pior caso cada envoltório terá apenas 3 pontos, de modo que a solução de cada teste terá complexidade $O(N^2 \log N)$

Solução AC com complexidade $O(TN^2 \log N)$

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

using namespace std;

struct Point
{
    int x, y;

    bool operator<(const Point& P) const
    {
        return x == P.x ? y < P.y : x < P.x;
    }
};

int D(const Point& P, const Point& Q, const Point& 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);
}

```

Solução AC com complexidade $O(TN^2 \log N)$

```cpp
vector<Point> monotone_chain(set<Point>& P)
{
    vector<Point> lower, upper;

    if (P.size() < 3)
        return lower;

    for (const auto& p : P)
    {
        auto size = lower.size();

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

        lower.push_back(p);
    }

```

Solução AC com complexidade $O(TN^2 \log N)$

```
    for (auto it = P.rbegin(); it != P.rend(); ++it)
    {
        auto p = *it;
        auto size = upper.size();

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

        upper.push_back(p);
    }

    lower.pop_back();
    upper.pop_back();

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

    for (const auto& p : lower)
```

Solução AC com complexidade $O(TN^2 \log N)$

```
        P.erase(p);

    return lower;
}

int main()
{
    ios::sync_with_stdio(false);

    int N;

    while (cin >> N, N)
    {
        set<Point> P;
        int x, y;

        for (int i = 0; i < N; ++i)
        {
            cin >> x >> y;
            P.insert(Point { x, y });
        }
```

Solução AC com complexidade $O(TN^2 \log N)$

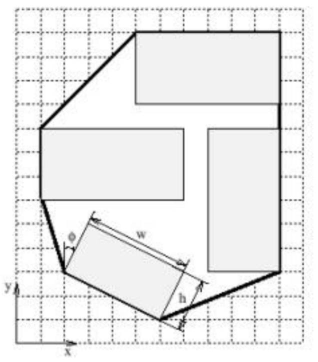
```
        int ans = 0;

        while (monotone_chain(P).size() > 2)
            ++ans;

        cout << (ans % 2 ? "Take this onion to the lab!\n" :
            "Do not take this onion to the lab!\n");
    }

    return 0;
}
```

UVA 10652 – Board Wrapping

Problema

The small sawmill in Mission, British Columbia, has developed a brand new way of packaging boards for drying. By fixating the boards in special moulds, the board can dry efficiently in a drying room.

Space is an issue though. The boards cannot be too close, because then the drying will be too slow. On the other hand, one wants to use the drying room efficiently.



Problema

Looking at it from a 2-D perspective, your task is to calculate the fraction between the space occupied by the boards to the total space occupied by the mould. Now, the mould is surrounded by an aluminium frame of negligible thickness, following the hull of the boards' corners tightly. The space occupied by the mould would thus be the interior of the frame.

Input

On the first line of input there is one integer, $N \leq 50$, giving the number of test cases (moulds) in the input. After this line, N test cases follow. Each test case starts with a line containing one integer n, $1 < n \leq 600$, which is the number of boards in the mould. Then n lines follow, each with five floating point numbers x, y, w, h, ϕ where $0 \leq x, y, w, h \leq 10000$ and $-90^o < \phi \leq 90^o$. The x and y are the coordinates of the center of the board and w and h are the width and height of the board, respectively. ϕ is the angle between the height axis of the board to the y-axis in degrees, positive clockwise. That is, if $\phi = 0$, the projection of the board on the x-axis would be w. Of course, the boards cannot intersect.

Output

For every test case, output one line containing the fraction of the space occupied by the boards to the total space in percent. Your output should have one decimal digit and be followed by a space and a percent sign ('%').

Note: The Sample Input and Sample Output corresponds to the given picture

Exemplo de entradas e saídas

Sample Input

```
1
4
4 7.5 6 3 0
8 11.5 6 3 0
9.5 6 6 3 90
4.5 3 4.4721 2.2361 26.565
```

Sample Output

```
64.3 %
```

Solução $O(TN \log N)$

- A solução consiste em três etapas
- A primeira é determinar o área total ocupada pelas placas
- Basta somar a área individual de cada placa, que é o produto da base pela altura
- Em seguida, é preciso determinar os vértices de cada placa
- Pode-se assumir que eles estão inicialmente com o centro no origem, fazer a rotação em sentido horário e, em seguida, transladar os pontos para a posição correta
- Determinados estes pontos, os limites do polígono corresponde ao envoltório convexo
- A área do polígono pode ser determinada através da expressão que computa esta área por meio dos vértices do polígono
- A resposta será a diferença entre ambas áreas, em porcentagem

Solução com complexidade $O(TN \log N)$

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

using namespace std;

const double EPS { 1e-5 };
const double PI { acos(-1.0) };

bool equals(double a, double b)
{
    return fabs(a - b) < EPS;
}

struct Point {
public:
    double x, y;

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

```

Solução com complexidade $O(TN \log N)$

```
    Point translate(double dx, double dy) const
    {
        return Point { x + dx, y + dy };
    }

    Point rotate(double angle) const
    {
        return Point { x*cos(angle) - y*sin(angle),
            x*sin(angle) + y*cos(angle) };
    }
};

double D(const Point& P, const Point& Q, const Point& 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);
}

struct Polygon {
    vector<Point> vs;
    int n;
```

Solução com complexidade $O(TN \log N)$

```
    Polygon(const vector<Point>& vertices) : vs(vertices), n(vs.size())
    {
        vs.push_back(vs[0]);
    }

    double area() const
    {
        double a = 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);
    }
};
```

Solução com complexidade $O(TN \log N)$

```
Point pivot(vector<Point>& 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];
}

void sort_by_angle(vector<Point>& P)
{
    auto P0 = pivot(P);

```

Solução com complexidade $O(TN \log N)$

```
    sort(P.begin() + 1, P.end(), [&](const Point& A, const Point& B)
        {
            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;
        }
    );
}

Polygon convex_hull(const vector<Point>& points)
{
    vector<Point> P(points);
    auto n = P.size();

    if (n <= 3)
        return Polygon(P);

```

Solução com complexidade $O(TN \log N)$

```
    sort_by_angle(P);

    vector<Point> s;
    s.push_back(P[n - 1]);
    s.push_back(P[0]);
    s.push_back(P[1]);

    size_t i = 2;

    while (i < n)
    {
        auto j = s.size() - 1;

        if (D(s[j - 1], s[j], P[i]) > 0)
            s.push_back(P[i++]);
        else
            s.pop_back();
    }

    s.pop_back();
```

Solução com complexidade $O(TN \log N)$

```cpp
    return Polygon(s);
}

int main()
{
    int T;
    cin >> T;

    while (T--)
    {
        int n;
        cin >> n;

        vector<Point> points;
        double boards_area = 0;

        while (n--)
        {
            double x, y, w, h, theta;
            cin >> x >> y >> w >> h >> theta;

```

Solução com complexidade $O(TN \log N)$

```
            theta /= 180.0;
            theta *= PI;

            double xv = w / 2, yv = h / 2;

            vector<Point> ps { Point { xv, yv }, Point { -xv, yv },
                Point { -xv, -yv }, Point { xv, -yv } };

            for (auto p : ps)
            {
                auto q = p.rotate(-theta);
                points.push_back(q.translate(x, y));
            }

            boards_area += w * h;
        }

        auto ch = convex_hull(points);
        auto total = ch.area();
        auto percent = 100.0 * boards_area / total;

```

Solução com complexidade $O(TN \log N)$

```c
        printf("%.1f %%\n", percent);
    }

    return 0;
}
```

Referências

1. URI 1464 – Camadas de Cebola
2. UVA 10652 – Board Wrapping