1 Coordinate compression: introdução

2 Um problema do USACO: Silver

Problema 1. *(http://www.usaco.org/index.php?page=viewproblem2&cpid=1063) Farmer John's largest pasture can be regarded as a large 2D grid of square "cells" (picture a huge chess board). Currently, there are N cows occupying some of these cells ($1 \leq N \leq 2500$). Farmer John wants to build a fence that will enclose a rectangular region of cells; the rectangle must be oriented so its sides are parallel with the x and y axes, and it could be as small as a single cell. Please help him count the number of distinct subsets of cows that he can enclose in such a region. Note that the empty subset should be counted as one of these.*

INPUT FORMAT *(input arrives from the terminal / stdin): The first line contains a single integer N. Each of the next N lines Each of the next N lines contains two space-separated integers, indicating the (x, y)coordinates of a cow's cell. All x coordinates are distinct from each-other, and all y coordinates are distinct from each-other. All x and y values lie in the range $0 \ldots 10^9$.*

OUTPUT FORMAT *(print output to the terminal / stdout): The number of subsets of cows that FJ can fence off. It can be shown that this quantity fits within a signed 64-bit integer (e.g., a "long long" in C/C++).*

Solução. Inicialmente, criamos elementos que serão necessários e realizamos a compressão de coordenadas:

Definições iniciais :

Utilizamos **vetores indexados a partir de 1** nessa porção teórica da solução.

Como $x, y \leq 10^9$, o que é muito grande para qualquer algoritmo linear ou superior, utilizamos **compressão de coordenadas**, da seguinte maneira:

- Criamos uma lista $x_c[]$ das componentes no eixo x de cada coordenada, em ordem crescente. É feito em $\Theta(nlogn)$, pois as componentes do eixo x são distintas para qualquer par de coordenadas.
- Definimos $y_c[]$ de forma análoga para as componentes do eixo y.

Lema 2.1. *A compressão de coordenadas não afeta o resultado: a quantidade de subconjuntos que podem ser cobertos por retângulos na configuração "comprimida" é a mesma que na configuração original.*

Dessa forma, definimos um mapa/função **monótona** $f_x : \{x_1, \ldots, x_n\} \to \{1, \ldots, n\}$ que "ignora" todas as coordenadas não utilizadas no eixo x (com f_y análoga). Assim, definimos a matriz A de ordem $n \times n$ da seguinte forma:

$$A[i][j] = \begin{cases} 1 \text{ se existe ponto em } (x_i, y_j) \\ 0 \text{ do contrário} \end{cases}$$

para todo $1 \leq i, j \leq n$. Isto é, A é a nova configuração **comprimida** de disposição dos nossos pontos.

Portanto, defina recusivamente P_s como a seguinte matriz

$$P_s[i][j] = A[i][j] + P_s[i-1][j] + P_s[i][j-1] - P_s[i-1][j-1]$$

definindo $P[i][j] = 0$ quando $i = 0$ ou $j = 0$.

Lema 2.2. *$P[i][j]$ é a quantidade de pontos (x, y) com $1 \leq f_x(x) \leq i$ e $1 \leq f_y(y) \leq j$.*

Isto é, é a quantidade de pontos na caixa $[1, i] \times [1, j]$ na nova configuração comprimida.

Resolvendo o problema : Trabalhamos a partir de agora sempre na configuração comprimida, em que existe exatamente um ponto em cada linha e exatamente um ponto em cada coluna.

Seja P o conjunto de pontos marcados na grade (isto é, a configuração das n vacas no pasto).

Com essas definições, podemos partir para a resolução do problema. Para todo subconjunto $A \subset P$ que pode ser coberto por um retângulo, sem que elementos de $P \setminus A$ sejam cobertos também, temos 3 casos:

0 $|A| < 2$ (trivial)

1 O retângulo possui duas extremidades em pontos de P

2 $|A| \geq 2$, porém o retângulo possui menos de duas extremidades em pontos de P.

Note que é impossível existir um retângulo com 3 ou 4 extremidades correspondendo a pontos marcados, do contrário existiriam pontos marcados com mesma coordenada x ou y, o que não ocorre segundo o enunciado.

Além disso, o terceiro caso pode ser obtido expandindo um retângulo que satisfaz o segundo caso para cima ou para baixo.

Definição 2.1. *Seja* $a : \{1, \ldots, n\}^3 \to \{1, \ldots, n\}$ *definida da seguinte forma:* $a(x_0, x_1, y_0)$ *para* $x_0 \le x_1$ *é a quantidade de pontos* $(x, y) \in P$ *que satisfazem* $x_0 \le x \le x_1$ *e* $y > y_0$*. Isto é, a quantidade de pontos marcados acima do subretângulo* $[x_0, x_1] \times [0, y_0]$*.*

Defina analogamente $b : \{1, \ldots, n\}^3 \to \{1, \ldots, n\}$ *da seguinte forma:* $b(x_0, x_1, y_1)$ *para* $x_0 \le x_1$ *é a quantidade de pontos* ***abaixo*** *do subretângulo* $[x_0, x_1] \times [y_1, n]$*.*

Lema 2.3. *Suponha que* $p = (x_0, y_0), q = (x_1, y_1)$ *são dois pontos marcados na configuração comprimida e sejam* $x_\ell = min(x_0, x_1),\ x_r = \max(x_0, x_1),\ y_\ell = min(y_0, y_1),\ y_r = \max(y_0, y_1)$*.*

Então expandindo (ou não) o subretângulo ***apenas para cima e para baixo*** $[x_\ell, x_r] \times [y_\ell, y_r]$ *podemos cobrir um total de* $(a(x_\ell, x_r, y_r) + 1) \cdot (b(x_\ell, x_r, y_\ell) + 1)$ *subconjuntos.*

Além disso, cada um desses subconjuntos só pode ser coberto a partir de expansão dessa fonte (desse retângulo com extremidades em p, q*).*

Prova. Como cada ponto marcado possui coordenada y distinta de todos os outros, ao expandir para cima podemos cobrir os pontos acima do subretângulo um de cada vez. O mesmo vale para os pontos abaixo do subretângulo.

Logo podemos escolher em cobrir qualquer quantidade $0 \le \alpha \le a(x_\ell, x_r, y_r)$ de pontos acima do subretângulo original (expandindo α unidades para cima na configuração comprimida), e qualquer quantidade $0 \le \beta \le b(x_\ell, x_r, y_\ell)$ de pontos abaixo. O resultado segue, portanto, pelo princípio multiplicativo.

Defina agora $R(p, q)$ como o retângulo com extremidades em p, q. Quanto à unicidade, suponha que exista p_1, q_1 distintos de p, q e um subconjunto $A \subseteq P$ que pode ser coberto pela expansão vertical de ambos $R(p, q)$ e $R(p_1, q_1)$. Defina $x'_\ell, x'_r, y'_\ell, y'_r$ de forma análoga para p_1, q_1. Como são distintos, temos que algum dos pares $(x'_\ell, x_\ell), (x'_r, x_r), (y'_\ell, y_\ell), (y'_r, y_r)$ possui dois valores distintos. Por exemplo, se $x'_\ell > x_\ell$ e x_ℓ é a coordenada x de p, então $R(p_1, q_1)$ nunca pode ser expandido verticalmente de forma a cobrir p. Isso é uma contradição, pois ao expandir verticalmente $R(p, q)$, sempre continuamos cobrindo p. Todos os outros casos são análogos. □

Lema 2.4. *Não é necessário expandir para a esquerda ou para direita. Ou seja: dado um conjunto de pontos* $A \subseteq P$ *e dois pontos* p, q *marcados, tal que* A *pode ser coberto a partir da expansão horizontal de* $R(p, q)$*, existem* p', q' *marcados tal que* A *pode ser coberto da expansão vertical de* $R(p', q')$*.*

Prova. Tome $p' = (p'_x, p'_y)$ como o ponto mais à esquerda de A e $q' = q'_x, q'_y$ como o ponto mais à direita de A, e expanda $\max_{u \in A} u_y - \max(p'_y, q'_y)$ unidades para cima e $\min(p'_y, q'_y) - \min_{u \in A} u_y$ unidades para baixo. O conjunto coberto será exatamente A. □

Portanto, considerando que existem exatamente $n+1$ conjunto com cardinalidade menor que 2 (os unitários e o vazio), e que todos eles podem ser cobertos por retângulos trivialmente, a resposta final é:

$$\text{Ans} = n + 1 + \sum_{p,q \in P} (a(\min(p_x, q_x), \max(p_x, q_x), \max(p_y, q_y)) + 1) \cdot (b(\min(p_x, q_x), \max(p_x, q_x), \min(p_y, q_y)) + 1) \quad (1)$$

Porém, pelo Lema 2.2, podemos concluir que

$$a(x_0, x_1, y) = (P_s[x_1][n] - P_s[x_0][n]) - (P_s[x_1][y] - P_s[x_0][y]) \quad (2)$$

pois $P_s[x_1][n] - P_s[x_0][n]$ conta quantos pontos marcados há em $[x_0, x_1] \times [1, n]$ e $P_s[x_1][y] - P_s[x_0][y]$ conta quantos pontos marcados há em $[x_0, x_1] \times [1, y]$. Similarmente

$$b(x_0, x_1, y) = \begin{cases} P_s[x_1][y-1] - P_s[x_0][y-1] \text{ se } y > 1 \\ 0 \text{ se } y = 1 \end{cases} \quad (3)$$

Isso finaliza a corretude do algoritmo a ser demonstrada em código na seção seguinte.

Calculando a complexidade. Podemos calcular $x_c[]$ e $y_c[]$ em $O(n \log n)$ com ordenação.

Depois, para cada ponto marcado $p \in P$, podemos calcular $f_x(p_x)$ e $f_y(p_y)$ em $O(\log n)$ usando busca binária nos vetores $x_c[],\ y_c[]$, e fazemos a substituição do ponto $p = (p_x, p_y)$ pelo ponto **já comprimido** $p' = (f_x(p_x), f_y(p_y))$. A complexidade total dessa seção novamente é $O(n \log n)$. Agora, já temos a configuração comprimida.

Dessa forma, a matriz A (e consequentemente, a matriz P_s) podem ser calculadas facilmente em $O(n^2)$, uma vez que são matrizes $n \times n$ e cada entrada pode ser determinada em tempo constante.

Por fim, usando eq. (2) e eq. (3) para calcular a quantidade de pontos abaixo/acima de um subretângulo em tempo constante, eq. (1) pode ser calculada em $O(\binom{n}{2}) = O(n^2)$. Isso se deve ao fato de que ela apresenta operações de tempo constante sobre cada par dos n pontos.

Como $\log n = o(n)$, a complexidade final do algoritmo é $O(n^2)$. □

□

2.1 Código em C++

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

int bsearch(int val, int x[], int n)
{
    int lo = 0, hi = n - 1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if (val == x[mid])
        {
            return mid;
        }
        else if (val < x[mid])
        {
            hi = mid - 1;
        }
        else
        {
            lo = mid + 1;
        }
    }
    return -1;
}

void swap(int &a, int &b)
{
    int temp;
    temp = max(a, b);
    a = min(a, b);
    b = temp;
}

int main()
{
    int n;
    cin >> n;
    int xs[n], ys[n];
    pair<int, int> points[n];
    for (int i = 0; i < n; i++)
    {
        cin >> points[i].first >> points[i].second;
        xs[i] = points[i].first;
        ys[i] = points[i].second;
    }
    sort(xs, xs + n);
    sort(ys, ys + n);

    int ps[n][n];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            ps[i][j] = 0;
        }
    }
```

```
        // compressing
        for (int i = 0; i < n; i++)
        {
            int ix = bsearch(points[i].first, xs, n);
            int iy = bsearch(points[i].second, ys, n);
            points[i].first = ix;
            points[i].second = iy;
            ++ps[ix][iy];
        }
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (j > 0)
                    ps[i][j] += ps[i][j - 1];
                if (i > 0)
                    ps[i][j] += ps[i - 1][j];
                if (i > 0 && j > 0)
                    ps[i][j] -= ps[i - 1][j - 1];
            }
        }

        long long ans = 0;
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < i; j++)
            {
                int i0 = points[i].first;
                int j0 = points[i].second;
                int i1 = points[j].first;
                int j1 = points[j].second;
                swap(i0, i1);
                swap(j0, j1);

                int a = (long long)(ps[i1][n - 1] - ps[i0][n - 1]) - (long long)(ps[i1][j1] - p
                int b = 0;
                if (j0 > 0)
                {

                    b += (long long)(ps[i1][j0 - 1] - ps[i0][j0 - 1]);
                }
                ans += (a + 1) * (b + 1);
            }
        }
        cout << ans + n + 1;
    }
```