Capybara dreaming

Contents

1 First things first

1.1 Includes

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

#define ll long long
#define pb push_back
#define D(x) cout << #x " = " << (x) << endl

typedef vector<int> vi;
typedef vector<vi> vvi;

typedef pair<int, int> ii;
typedef vector<ii> vii;
```

2 Matemática

2.1 Algoritmo de euclides extendido

```
int xmdc(int a, int b, int &x, int &y)
{
    if (b == 0)
    {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1, mdc = xmdc(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return mdc;
}
```

2.2 Máximo divisor comum

```
int mdc(int a, int b)
{
    int remainder;
    while (b != 0)
    {
        remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}
```

2.3 Mínimo múltiplo comum

```
int mmc(int a, int b)
{
    int temp = mdc(a, b);

    return temp ? (a / temp * b) : 0;
}
```

2.4 Algoritmo de Pollard Rho

```
int pollard(int number)
{
    x_fixed = 2, cycle_size = 2, x = 2, factor = 1;

    while (factor == 1)
    {
        for (int count = 1; count <= cycle_size && factor <= 1; count++)
        {
            x = (x * x + 1) % number;
            factor = mdc(x - x_fixed, number);
        }

        cycle_size *= 2;
        x_fixed = x;
    }
    return factor;
}
```

2.5 Transformada rápida de Fourrier

```
// Resolve:
// - De quantas maneiras conseguimos atingir Y com X tentativas
// - Dado X tentativas, conseguimos atingir Y?
// Complexidade:
// X * Ymax * Ymax(log Ymax)
```

```cpp
// TEOREMA DA CONVOLUcÃO:
// Podemos fazer a convolucão de 2 polinomios utilizando a FFT
// Reduzindo a complexidade de n^2 para n log n
// Definimos a convolucão como h[i] = sum(a[j] * b[j-i]) para todo j de 0 a i.
// Exemplo: h[5] = a[5] * b[0] + a[4] * b[1] + a[3] * b[2]...
// Segundo o teorema da convolucão
// h(f . g) = transformada inversa de (transformada (f) * transformada (g))
// onde . é o operador de convolucão.
// e * é o operador de multiplicacão termo a termo.

#include <bits/stdc++.h>

using namespace std;

// primeira potência de 2 maior que o limite de H
const int MAX_DIST = 262144 * 2;

typedef complex<double> cpx;
const double pi = acos(-1.0);

int p[MAX_DIST];
int maxDist;

// in:     vector de entrada
// out:    vector de saida
// n:      Tamanho do input/output {DEVE SER DA ORDEM DE 2}
// type:   1 = Transformada, -1 = Transformada inversa
void FFT(vector<cpx> &v, vector<cpx> &ans, int n, int type)
{
	int i, sz, o;
	p[0] = 0;
	for (i = 1; i < n; i++)
		p[i] = (p[i >> 1] >> 1) | ((i & 1) ? (n >> 1) : 0);
	for (i = 0; i < n; i++)
		ans[i] = v[p[i]];
	for (sz = 1; sz < n; sz <<= 1)
	{
		const cpx wn(cos(type * pi / sz), sin(type * pi / sz));
		for (o = 0; o < n; o += (sz << 1))
		{
			cpx w = 1;
			for (i = 0; i < sz; i++)
			{
				const cpx u = ans[o + i], t = w * ans[o + sz + i];
				ans[o + i] = u + t;
				ans[o + i + sz] = u - t;
				w *= wn;
			}
		}
	}

	if (type == -1)
		for (i = 0; i < n; i++)
			ans[i] /= n;
}

// Exemplo:
// Há um robo que pode disparar bolas em N distâncias diferentes.
// Queremos saber se ele alcanca uma distância M com 1 ou 2 tacadas.
// Resolucão:
// Podemos definir um vetor distances[MAX_DIST],
// onde a distances[i] = 1 se ele pode tacar até a distancia i
// e distances[i] = 0 caso contrario
// Para ver se o robo acerta com 1 tacada, é trivial.
// Para ver se o robo acerta com 2 tacadas, podemos fazer a convolucão de distances com distances.
// Ex: Acertar a Pode[10] é igual a: Pode[10] || Pode[9] * Pode[1] || Pode[8] * Pode[2]...
// Ou seja, H = FFTi(FFT(distances) ** 2);

// Complexidade:
// 2 * 200k * log(200k) = 8m

int main()
{
	ios::sync_with_stdio(false);
	cin.tie(0);

	int N, d;
	vector<cpx> distances, fftOut;

	while (cin >> N)
	{
		maxDist = 0;

		distances = vector<cpx>(MAX_DIST);
```

```
                fftOut = vector<cpx>(MAX_DIST);

                // Distancia 0 é uma posicão de "possível"
                distances[0] = cpx(1, 0);

                for (int i = 0; i < N; i++)
                {
                        cin >> d;
                        if (d > maxDist)
                                maxDist = d;
                        distances[d] = cpx(1, 0);
                }

                int shiftAmount;

                for (shiftAmount = 0; (maxDist >> shiftAmount) != 0; shiftAmount++)
                        ;

                maxDist = 1 << (shiftAmount + 1);

                // fftOut <= transformada de distances
                FFT(distances, fftOut, maxDist, 1);

                // Multiplicacão termo a termo de f e g, no caso, f = g = fftOut
                // fftOut *= fftOut
                for (int i = 0; i < maxDist; i++)
                        fftOut[i] = fftOut[i] * fftOut[i];

                // transformada inversa da multiplcacão termo a termo.
                FFT(fftOut, distances, maxDist, -1);

                cin >> N;
                int total = 0;

                for (int i = 0; i < N; i++)
                {
                        cin >> d;

                        // Entra a distancia d
                        // e verifica se a parte real da distância[d] é positiva
                        // distância[d] guarda de quantas maneiras conseguimos atingir D
                        if (distances[d].real() > 0.01)
                                total++;
                }

                cout << total << endl;
        }
}
```

2.6 Matrizes

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

#define ll long long

typedef vector<ll> vl;
typedef vector<vl> vvl;
const int mod = 1000000;

// Retorna a matriz I_n
vvl matrixUnit(int n) {
    vvl res(n, vl(n));
    for (int i = 0; i < n; i++)
        res[i][i] = 1;
    return res;
}

// Retorna a+b
vvl matrixAdd(const vvl &a, const vvl &b) {
    int n = a.size();
    int m = a[0].size();
    vvl res(n, vl(m));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            res[i][j] = (a[i][j] + b[i][j]) % mod;
    return res;
}

// Retorna a*b
vvl matrixMul(const vvl &a, const vvl &b) {
    int n = a.size();
    int m = a[0].size();
    int k = b[0].size();
    vvl res(n, vl(k));
```

```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < k; j++)
            for (int p = 0; p < m; p++)
               res[i][j] = (res[i][j] + ((a[i][p] % mod) * (b[p][j] % mod) % mod)) % mod;
    return res;
}

// Retorna a matriz a^p
vvl matrixPow(const vvl &a, long long p) {
    if (p == 0)
        return matrixUnit(a.size());
    if (p & 1)
        return matrixMul(a, matrixPow(a, p - 1));
    return matrixPow(matrixMul(a, a), p / 2);
}

// Retorna sum^p_i=0 (a^i)
vvl matrixPowSum(const vvl &a, long long p) {
    long long n = a.size();
    if (p == 0)
        return vvl(n, vl(n));
    if (p % 2 == 0)
        return matrixMul(matrixPowSum(a, p / 2), matrixAdd(matrixUnit(n),  matrixPow(a, p / 2)));
    return matrixAdd(a, matrixMul(matrixPowSum(a, p - 1), a));
}

int main() {
        long long n, l, k, i;

        while(scanf("%lld %lld %lld", &n, &l, &k) > 0) {
                vvl matriz = vvl(2, vl(2));

                matriz[0][0] = l;
                matriz[0][1] = k;
                matriz[1][0] = 1;
                matriz[1][1] = 0;

                matriz = matrixPow(matriz, n / 5);

                printf("%06lld\n", matriz[0][0]);
        }
}
```

2.7 Quantidade de fatores primos de um número

```
#define pb push_back

typedef vector<int> vi;

int main()
{
    long long i, j, n, qtd = 0, resp = 0;
    bool crivo[LIM] = {0};
    vi primos;

    for (i = 2; i < LIM; i++)
        if (!crivo[i])
        {
            primos.pb(i);

            for (j = i + i; j < LIM; j += i)
                crivo[j] = 1;
        }

    scanf("%lld", &n);

    for (auto it : primos)
        if (n % it == 0)
        {
            qtd++;

            n /= it;

            while (n % it == 0)
                n /= it;
        }
        else if (it > n)
            break;

    if (n != 1)
        qtd++;

    printf("%d\n", qtd);
}
```

2.8 Fatoração em números primos

```
vector<int> primeFactors(int n)
{
    vector<int> v;
    int sqrtn = sqrt(n);

    while (n % 2 == 0)
    {
        v.push_back(2);
        n = n / 2;
    }

    for (int i = 3; i <= sqrtn; i = i + 2)
    {
        while (n % i == 0)
        {
            v.push_back(i);
            n = n / i;
        }
    }

    if (n > 2)
        v.push_back(n);

    return v;
}
```

2.9 Modpow

```
int modPow(int a, int b, int m)
{
    int res = 1;
    for (; b > 0; b >>= 1)
    {
        if (b & 1)
            res = (long long)res * a % m;
        a = (long long)a * a % m;
    }
    return res;
}
```

2.10 Quantidade de coprimos.

```
#define pb push_back

typedef vector<int> vi;

bool crivo[100000] = {0};
vi primos;

int main() {
        int i, j, n, resp;

        for(i = 2; i < 100000; i++)
                if(!crivo[i]) {
                        primos.pb(i);
                        for(j = i + i; j < 100000; j+=i)
                                crivo[j] = 1;
                }

        while(scanf("%d", &n) > 0) {
                resp = n;

                for(auto &it : primos) {
                        if(it * it > n) {
                                if(n != 1)
                                        resp -= resp / n;

                                break;
                        }

                        if(n % it == 0) {
                                resp -= resp / it;

                                while(n % it == 0)
                                        n /= it;
                        }
```

```
            }
            printf("%d\n", resp / 2);
        }
}
```

2.11 Máximo e mínimo de funções

```
double gss(double a, double b, double (*f)(double), double e = 1e-6)
{
    double r = (sqrt(5) - 1) / 2; //=.618...=golden ratio-1
    double x1 = b - r * (b - a), x2 = a + r * (b - a);
    double f1 = f(x1), f2 = f(x2);
    while (b - a > e)
    {
        if (f1 < f2)
        { //change to > to find maximum
            b = x2;
            x2 = x1;
            f2 = f1;
            x1 = b - r * (b - a);
            f1 = f(x1);
        }
        else
        {
            a = x1;
            x1 = x2;
            f1 = f2;
            x2 = a + r * (b - a);
            f2 = f(x2);
        }
    }
    return (b + a) / 2;
}
```

2.12 Todos divisores de um número

```
vector<int> divisores(int n)
{
    vector<int> divis;
    int sqrtn = sqrt(n);

    while(sqrtn * sqrtn < n)
        sqrtn++;

    for (i = 1; i < sqrtn; i++)
        if (!(n % i))
            divis.push_back(i), divis.push_back(n / i);

    if(sqrtn * sqrtn == n)
        divis.push_back(sqrtn);

    return divis;
}
```

2.13 Crivo de Eratóstenes segmentado

```
char nprimo[100001] = {0}; // tamanho = sqrt(maximo)
std::vector<int> primos;

int main()
{
    int n, a, b, i, j;

    nprimo[1] = 1;
    nprimo[0] = 1;

    for (i = 2; i < 320; i++) // i [2, sqrt(sqrt(maximo))]
        if (!nprimo[i])
            for (j = i * i; j < 100001; j += i) // j [i^2, sqrt(maximo)]
                nprimo[j] = 1;

    for (i = 2; i < 100001; i++)
        if (!nprimo[i])
            primos.push_back(i);
```

```
    scanf("%d", &n);
    while (n--)
    {
        scanf("%d %d", &a, &b);
        if (a > 100000 && b > 100000)
        { // (a > sqrt(N) && b > sqrt(N))
            for (i = a; i <= b; i++)
            {
                for (j = 0; j < primos.size(); j++)
                    if (i % primos[j] == 0)
                        goto ab;
                printf("%d\n", i);
            ab:;
            }
        }
        else if (a < 100001 && b < 100001)
        { // (a < sqrt(N) && b < sqrt(N))
            for (i = a; i <= b; i++)
                if (!nprimo[i])
                    printf("%d\n", i);
        }
        else
        {
            for (i = 0; i < primos.size(); i++)
                if (primos[i] >= a)
                    break;
            for (; i < primos.size(); i++)
                printf("%d\n", primos[i]);
            for (; i <= b; i++)
            {
                for (j = 0; j < primos.size(); j++)
                    if (i % primos[j] == 0)
                        goto ac;
                printf("%d\n", i);
            ac:;
            }
        }
    }
}
```

3 Grafos

3.1 Grafos

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

#define ll long long
#define pb push_back

typedef vector<int> vi;

struct Vertice
{
    int id, pai;
    ll dist;

    Vertice(int id, ll dist = 1, int pai = -1) : id(id), dist(dist), pai(pai) {}

    bool operator<(Vertice a) const
    {
        return a.dist < dist;
    }
};

typedef vector<Vertice> vv;
typedef vector<vv> vvv;

struct Grafo
{
    vvv g;
    vi pais;
    int n;

    Grafo(int n) : n(n)
    {
        g = vvv(n, vv());
        pais = vi(n);
    }

    void operator=(Grafo const &a)
    {
        g = a.g;
        pais = a.pais;
        n = a.n;
    }

    void addAresta(int a, int b, ll d = 0)
    {
        g[a].pb(Vertice(b, d));
    }

    void removeAresta(int a, int b)
    {
        g[a].erase(remove_if(g[a].begin(), g[a].end(), [b](Vertice v) { return v.id == b; }));
    }

    ll valAresta(int a, int b)
    {
        for (auto it : g[a])
            if (it.id == b)
                return it.dist;

        return 0;
    }

    void modificaAresta(int a, int b, ll dif)
    {
        for (auto &it : g[a])
            if (it.id == b)
            {
                it.dist += dif;
                break;
            }

        g[a].erase(remove_if(g[a].begin(), g[a].end(), [b](Vertice v) { return v.dist == 0; }));
    }

    ll dijkstra(int s, int d)
    {
        priority_queue<Vertice> fila;
        bool visitados[n];

        fill(visitados, visitados+n, 0);
        fill(pais.begin(), pais.end(), -1);
```

```
    fila.push(Vertice(s, 0));

    auto top = fila.top();

    while (top.id != d)
    {
        if (!visitados[top.id])
        {
            for (auto &it : g[top.id])
                if (!visitados[it.id])
                    fila.push(Vertice(it.id, it.dist + top.dist, top.id));

            visitados[top.id] = 1;
            pais[top.id] = top.pai;
        }

        fila.pop();

        if (fila.empty())
            return -1;

        top = fila.top();
    }

    pais[top.id] = top.pai;

    return top.dist;
}

ll busca(int s, int d)
{
    queue<Vertice> fila;
    bool visitados[n];

    fill(visitados, visitados+n, 0);
    fill(pais.begin(), pais.end(), -1);

    fila.push(Vertice(s, 0));

    auto top = fila.front();

    while (top.id != d)
    {
        if (!visitados[top.id])
        {
            for (auto &it : g[top.id])
                if (!visitados[it.id])
                    fila.push(Vertice(it.id, it.dist + 1, top.id));

            visitados[top.id] = 1;
            pais[top.id] = top.pai;
        }

        fila.pop();

        if (fila.empty())
            return -1;

        top = fila.front();
    }

    pais[top.id] = top.pai;

    return top.dist;
}

ll fluxo_maximo(int s, int d)
{
    int u, v;
    ll flow = 0;

    Grafo g2 = *this;

    while (g2.busca(s, d) >= 0)
    {
        ll path = 1ll << 50;

        for (v = d; v != s; v = u)
        {
            u = g2.pais[v];
            path = min(path, valAresta(u, v));
        }

        for (v = d; v != s; v = u)
        {
            u = g2.pais[v];
            g2.modificaAresta(u, v, -path);
```

```
                g2.modificaAresta(v, u, path);
            }
            flow += path;
        }
        return flow;
    }
};
int main()
{
    Grafo g(20);
    g.addAresta(1, 2, 1);
    g.addAresta(1, 3, 5);
    g.addAresta(2, 1, 6);
    g.addAresta(3, 2, 10);
    g.removeAresta(1, 2);
    for (auto it : g.g[1])
        cout << it.id << endl; // 3
    cout << g.dijkstra(1, 2) << endl; // 15
    cout << g.fluxo_maximo(1, 2) << endl; // 5
}
```

3.2 Todas as pontes de um grafo

```
#include <bits/stdc++.h>
#define NIL -1
using namespace std;
// A class that represents an undirected graph
class Graph
{
    int V;          // No. of vertices
    list<int> *adj; // A dynamic array of adjacency lists
    void bridgeUtil(int v, bool visited[], int disc[], int low[],
                    int parent[]);
  public:
    Graph(int V);                 // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    void bridge();                // prints all bridges
};
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}
// A recursive function that finds and prints bridges using
// DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void Graph::bridgeUtil(int u, bool visited[], int disc[],
                       int low[], int parent[])
{
    // A static variable is used for simplicity, we can
    // avoid use of static variable by passing a pointer.
    static int time = 0;
    // Mark the current node as visited
    visited[u] = true;
    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    // Go through all vertices aadjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
```

```
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then recur for it
        if (!visited[v])
        {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = min(low[u], low[v]);

            // If the lowest vertex reachable from subtree
            // under v is  below u in DFS tree, then u-v
            // is a bridge
            if (low[v] > disc[u])
                cout << u << " " << v << endl;
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u] = min(low[u], disc[v]);
    }
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void Graph::bridge()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];

    // Initialize parent and visited arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find Bridges
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            bridgeUtil(i, visited, disc, low, parent);
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nBridges in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();

    cout << "\nBridges in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.bridge();

    cout << "\nBridges in third graph \n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.bridge();

    return 0;
```

```
}
```

3.3 Isomorfismo de árvores

```
#include <bits/stdc++.h>
#define pb push_back
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;
int main()
{
	int n, i, a, b, count, atual;
	while (scanf("%d", &n) > 0)
	{
		map<multiset<int>, int> mapa;
		vvi grafo_esq(n), grafo_dir(n);
		vi valores_esq(n), valores_dir(n), pais_esq(n, -1), pais_dir(n, -1);
		set<int> centros_esq, centros_dir;
		for (i = 0; i < n - 1; i++)
			scanf("%d %d", &a, &b), grafo_esq[a - 1].pb(b - 1), grafo_esq[b - 1].pb(a - 1),
				centros_esq.insert(i);
		centros_esq.insert(i);
		for (i = 0; i < n - 1; i++)
			scanf("%d %d", &a, &b), grafo_dir[a - 1].pb(b - 1), grafo_dir[b - 1].pb(a - 1),
				centros_dir.insert(i);
		centros_dir.insert(i);
		atual = count = 0;
		while (centros_esq.size() > 2)
		{
			vi a_remover;
			for (auto &linha : centros_esq)
			{
				int count = 0, pai;
				for (auto &it : grafo_esq[linha])
					if (centros_esq.count(it))
						count++, pai = it;
				if (count == 1)
				{
					pais_esq[linha] = pai;
					a_remover.pb(linha);
				}
			}
			for (auto &it : a_remover)
			{
				multiset<int> valores;
				for (auto &it2 : grafo_esq[it])
					if (pais_esq[it2] == it)
						valores.insert(valores_esq[it2]);
				if (mapa.count(valores))
					valores_esq[it] = mapa[valores];
				else
					valores_esq[it] = mapa[valores] = atual++;
				centros_esq.erase(it);
			}
		}
		for (auto &it : centros_esq)
		{
			multiset<int> valores;
			for (auto &it2 : grafo_esq[it])
				if (pais_esq[it2] == it)
					valores.insert(valores_esq[it2]);
			if (mapa.count(valores))
				valores_esq[it] = mapa[valores];
			else
				valores_esq[it] = mapa[valores] = atual++;
```

```
                }
                while (centros_dir.size() > 2)
                {
                        vi a_remover;
                        for (auto &linha : centros_dir)
                        {
                                int count = 0, pai;
                                for (auto &it : grafo_dir[linha])
                                        if (centros_dir.count(it))
                                                count++, pai = it;
                                if (count == 1)
                                {
                                        pais_dir[linha] = pai;
                                        a_remover.pb(linha);
                                }
                        }
                        for (auto &it : a_remover)
                        {
                                multiset<int> valores;
                                for (auto &it2 : grafo_dir[it])
                                        if (pais_dir[it2] == it)
                                                valores.insert(valores_dir[it2]);
                                if (mapa.count(valores))
                                        valores_dir[it] = mapa[valores];
                                else
                                        valores_dir[it] = mapa[valores] = atual++;
                                centros_dir.erase(it);
                        }
                }
                for (auto &it : centros_dir)
                {
                        multiset<int> valores;
                        for (auto &it2 : grafo_dir[it])
                                if (pais_dir[it2] == it)
                                        valores.insert(valores_dir[it2]);
                        if (mapa.count(valores))
                                valores_dir[it] = mapa[valores];
                        else
                                valores_dir[it] = mapa[valores] = atual++;
                }
                sort(valores_dir.begin(), valores_dir.end());
                sort(valores_esq.begin(), valores_esq.end());
                for(i = 0; i < valores_dir.size(); i++)
                        if(valores_esq[i] != valores_dir[i]) {
                                puts("N");
                                goto proximo;
                        }
                puts("S");
                proximo:;
        }
}
```

3.4 Matching máximo em grafo bipartido

```
const int MAXN1 = 50000, MAXN2 = 50000, MAXM = 150000;
int n1, n2, edges, last[MAXN1], prev[MAXM], head[MAXM], matching[MAXN2], dist[MAXN1], Q[MAXN1], used[MAXN1], vis
    [MAXN1];
void init(int _n1, int _n2)
{
    n1 = _n1;
    n2 = _n2;
    edges = 0;
    fill(last, last + n1, -1);
}
void addAresta(int u, int v)
{
    head[edges] = v;
```

```
    prev[edges] = last[u];
    last[u] = edges++;
}

void bfs()
{
    fill(dist, dist + n1, -1);
    int sizeQ = 0;
    for (int u = 0; u < n1; ++u)
    {
        if (!used[u])
        {
            Q[sizeQ++] = u;
            dist[u] = 0;
        }
    }
    for (int i = 0; i < sizeQ; i++)
    {
        int u1 = Q[i];
        for (int e = last[u1]; e >= 0; e = prev[e])
        {
            int u2 = matching[head[e]];
            if (u2 >= 0 && dist[u2] < 0)
            {
                dist[u2] = dist[u1] + 1;
                Q[sizeQ++] = u2;
            }
        }
    }
}

bool dfs(int u1)
{
    vis[u1] = true;
    for (int e = last[u1]; e >= 0; e = prev[e])
    {
        int v = head[e];
        int u2 = matching[v];
        if (u2 < 0 || !vis[u2] && dist[u2] == dist[u1] + 1 && dfs(u2))
        {
            matching[v] = u1;
            used[u1] = true;
            return true;
        }
    }
    return false;
}
int maxMatching()
{
    fill(used, used + n1, false);
    fill(matching, matching + n2, -1);
    for (int res = 0;;)
    {
        bfs();
        fill(vis, vis + n1, false);
        int f = 0;
        for (int u = 0; u < n1; ++u)
            if (!used[u] && dfs(u))
                ++f;
        if (!f)
            return res;
        res += f;
    }
}
```

3.5 Algoritmo húngaro

```
#define N 100
#define INF 100000000

int cost[N][N], n, max_match, lx[N], ly[N], xy[N], yx[N], slack[N], slackx[N], prev[N];
bool S[N], T[N];

void init_labels()
{
    memset(lx, 0, sizeof(lx));
    memset(ly, 0, sizeof(ly));
    for (int x = 0; x < n; x++)
        for (int y = 0; y < n; y++)
            lx[x] = max(lx[x], cost[x][y]);
```

```
}
void update_labels()
{
    int x, y, delta = INF;
    for (y = 0; y < n; y++)
        if (!T[y])
            delta = min(delta, slack[y]);
    for (x = 0; x < n; x++)
        if (S[x])
            lx[x] -= delta;
    for (y = 0; y < n; y++)
        if (T[y])
            ly[y] += delta;
    for (y = 0; y < n; y++)
        if (!T[y])
            slack[y] -= delta;
}
void add_to_tree(int x, int prevx)
{
    S[x] = true;
    prev[x] = prevx;
    for (int y = 0; y < n; y++)
        if (lx[x] + ly[y] - cost[x][y] < slack[y])
        {
            slack[y] = lx[x] + ly[y] - cost[x][y];
            slackx[y] = x;
        }
}
void augment()
{
    if (max_match == n)
        return;
    int x, y, root, q[N], wr = 0, rd = 0;
    memset(S, false, sizeof(S));
    memset(T, false, sizeof(T));
    memset(prev, -1, sizeof(prev));
    for (x = 0; x < n; x++)
        if (xy[x] == -1)
        {
            q[wr++] = root = x;
            prev[x] = -2;
            S[x] = true;
            break;
        }
    for (y = 0; y < n; y++)
    {
        slack[y] = lx[root] + ly[y] - cost[root][y];
        slackx[y] = root;
    }
    while (true)
    {
        while (rd < wr)
        {
            x = q[rd++];
            for (y = 0; y < n; y++)
                if (cost[x][y] == lx[x] + ly[y] && !T[y])
                {
                    if (yx[y] == -1)
                        break;
                    T[y] = true;
                    q[wr++] = yx[y];
                    add_to_tree(yx[y], x);
                }
            if (y < n)
                break;
        }
        if (y < n)
            break;
        update_labels();
        wr = rd = 0;
        for (y = 0; y < n; y++)
            if (!T[y] && slack[y] == 0)
```

```
            {
                if (yx[y] == -1)
                {
                    x = slackx[y];
                    break;
                }
                else
                {
                    T[y] = true;
                    if (!S[yx[y]])
                    {
                        q[wr++] = yx[y];
                        add_to_tree(yx[y], slackx[y]);
                    }
                }
            }
        if (y < n)
            break;
    }
    if (y < n)
    {
        max_match++;
        for (int cx = x, cy = y, ty; cx != -2; cx = prev[cx], cy = ty)
        {
            ty = xy[cx];
            yx[cy] = cx;
            xy[cx] = cy;
        }
        augment();
    }
}
int hungaro()
{
    int ret = 0;
    max_match = 0;
    memset(xy, -1, sizeof(xy));
    memset(yx, -1, sizeof(yx));
    init_labels();
    augment();
    for (int x = 0; x < n; x++)
        ret += cost[x][xy[x]];
    return ret;
}
```

4 Strings

4.1 Suffix array

```
//Usage:
// Fill txt with the characters of the txting.
// Call SuffixSort(n), where n is the length of the txting stored in txt.
// That's it!

//Output:
// SA = The suffix array.
// Contains the n suffixes of txt sorted in lexicographical order.
// Each suffix is represented as a single integer (the SAition of txt where it starts).
// iSA = The inverse of the suffix array. iSA[i] = the index of the suffix txt[i..n)
//   in the SA array. (In other words, SA[i] = k <==> iSA[k] = i)
//   With this array, you can compare two suffixes in O(1): Suffix txt[i..n) is smaller
//   than txt[j..n) if and only if iSA[i] < iSA[j]

const int MAX = 100010;
char txt[MAX];          //input
int iSA[MAX], SA[MAX];  //output
int cnt[MAX], prox[MAX]; //internal
bool bh[MAX], b2h[MAX];

// Compares two suffixes according to their first characters
bool smaller_first_char(int a, int b)
{
    return txt[a] < txt[b];
}

void suffixSort(int n)
{
    for (int i = 0; i < n; ++i)
        SA[i] = i;

    sort(SA, SA + n, smaller_first_char);

    for (int i = 0; i < n; ++i)
    {
        bh[i] = i == 0 || txt[SA[i]] != txt[SA[i - 1]];
        b2h[i] = false;
    }

    for (int h = 1; h < n; h <<= 1)
    {
        int buckets = 0;

        for (int i = 0, j; i < n; i = j)
        {
            j = i + 1;

            while (j < n && !bh[j])
                j++;

            prox[i] = j;
            buckets++;
        }

        if (buckets == n)
            break;

        for (int i = 0; i < n; i = prox[i])
        {
            cnt[i] = 0;

            for (int j = i; j < prox[i]; ++j)
                iSA[SA[j]] = i;
        }

        cnt[iSA[n - h]]++;
        b2h[iSA[n - h]] = true;

        for (int i = 0; i < n; i = prox[i])
        {
            for (int j = i; j < prox[i]; ++j)
            {
                int s = SA[j] - h;

                if (s >= 0)
                {
                    int head = iSA[s];
                    iSA[s] = head + cnt[head]++;
                    b2h[iSA[s]] = true;
                }
```

```
            }
            for (int j = i; j < prox[i]; ++j)
            {
                int s = SA[j] - h;

                if (s >= 0 && b2h[iSA[s]])
                    for (int k = iSA[s] + 1; !bh[k] && b2h[k]; k++)
                        b2h[k] = false;
            }
        }
        for (int i = 0; i < n; ++i)
        {
            SA[iSA[i]] = i;
            bh[i] |= b2h[i];
        }
    }
    for (int i = 0; i < n; ++i)
    {
        iSA[SA[i]] = i;
    }
}
// End of suffix array algorithm

// Begin of the O(n) longest common prefix algorithm
int lcp[MAX];
// lcp[i] = length of the longest common prefix of suffix SA[i] and suffix SA[i-1]
// lcp[0] = 0
void getlcp(int n)
{
    for (int i = 0; i < n; ++i)
        iSA[SA[i]] = i;

    lcp[0] = 0;

    for (int i = 0, h = 0; i < n; ++i)
    {
        if (iSA[i] > 0)
        {
            int j = SA[iSA[i] - 1];

            while (i + h < n && j + h < n && txt[i + h] == txt[j + h])
                h++;

            lcp[iSA[i]] = h;

            if (h > 0)
                h--;
        }
    }
}
```

5 Geometria

5.1 Linha de eventos radial

```
//  - Radial sweep in Q2 quadrant in nlogn.
//  - Sorts events using cross product to avoid dealing with
//    numeric problems.
#include <bits/stdc++.h>
using namespace std;

struct Point {
	Point(int x = 0, int y = 0) : x(x), y(y) {}

	bool operator<(const Point& o) const {
		// Order points in a quadrant by angle with origin:
		// Uses anti-clockwise order by returning true when the
		// cross product between the points is positive.
		return (x*o.y - y*o.x) > 0;
	}

	/*
	bool operator<=(const Point& o) const {
	return (x*o.y - y*o.x) >= 0;
	}
	*/
	int x, y;
};

pair<int, int> solve(const vector<Point>& points) {
	map<Point, pair<int, int> > events;

	Point begin(0, 1);
	Point end(-1, 0);

	// Add events on the borders to guarantee that we consider them.
	events[begin];
	events[end];

	int superior = 0;  // Number of points in Q1 quadrant.
	int same = 0;      // Number of points in origin.
	int active = 0;    // Number of current points in Q2 and Q4 quadrant better
	                                // than origin.

	int best_pos = points.size();
	int worst_pos = 0;

	for (const auto& p : points) {
		if (p.x < 0 && p.y < 0) {}
		else if (p.x > 0 && p.y > 0) superior++;
		else if (p.x == 0 && p.y == 0) same++;
		else if (p.x <= 0 && p.y >= 0) {
			// assert(begin <= Point(p.x, p.y));
			//assert(Point(p.x, p.y) <= end);
			events[Point(p.x, p.y)].first++;
		}
		else if (p.x >= 0 && p.y <= 0) {
			//assert(begin <= Point(-p.x, -p.y));
			//assert(Point(-p.x, -p.y) <= end);
			active++;
			events[Point(-p.x, -p.y)].second++;
		}
		else assert(false);
	}

	for (const auto& e : events) {
		int tie_best_pos = superior + active - e.second.second;
		int tie_worst_pos = superior + active + e.second.first + same;
		active += e.second.first - e.second.second;

		best_pos = min(best_pos, tie_best_pos);
		worst_pos = max(worst_pos, tie_worst_pos);
	}

	return make_pair(best_pos + 1, worst_pos + 1);
}

// Reads the set of points and centers them around Maria's product.
vector<Point> read() {
	int n, cx, cy;
	cin >> n >> cx >> cy;
	vector<Point> points(n - 1);
	for (Point& p : points) {
```

```
            cin >> p.x >> p.y;
            p.x -= cx;
            p.y -= cy;
        }
        return points;
}

int main() {
        auto input = read();
        auto solution = solve(input);

        for (auto& i : input)
                swap(i.x, i.y);

        assert(solution == solve(input));

        cout << solution.first << " " << solution.second << endl;
        return 0;
}
```

5.2 KD-Tree para pares mais próximos em O(log(n))

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

using namespace std;

typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};

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

// sorts points on x-coordinate
bool on_x(const point &a, const point &b)
{
    return a.x < b.x;
}

// sorts points on y-coordinate
bool on_y(const point &a, const point &b)
{
    return a.y < b.y;
}

// squared distance between points
ntype pdist2(const point &a, const point &b)
{
    ntype dx = a.x-b.x, dy = a.y-b.y;
    return dx*dx + dy*dy;
}

// bounding box for a set of points
struct bbox
{
    ntype x0, x1, y0, y1;

    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

    // computes bounding box from a bunch of points
    void compute(const vector<point> &v) {
        for (int i = 0; i < v.size(); ++i) {
            x0 = min(x0, v[i].x);   x1 = max(x1, v[i].x);
            y0 = min(y0, v[i].y);   y1 = max(y1, v[i].y);
        }
    }

    // squared distance between a point and this bbox, 0 if inside
    ntype distance(const point &p) {
        if (p.x < x0) {
            if (p.y < y0)       return pdist2(point(x0, y0), p);
            else if (p.y > y1)  return pdist2(point(x0, y1), p);
            else                return pdist2(point(x0, p.y), p);
        }
        else if (p.x > x1) {
            if (p.y < y0)       return pdist2(point(x1, y0), p);
```

```
            else if (p.y > y1)  return pdist2(point(x1, y1), p);
            else                return pdist2(point(x1, p.y), p);
        }
        else {
            if (p.y < y0)       return pdist2(point(p.x, y0), p);
            else if (p.y > y1)  return pdist2(point(p.x, y1), p);
            else                return 0;
        }
    }
};

// stores a single node of the kd-tree, either internal or leaf
struct kdnode
{
    bool leaf;      // true if this is a leaf node (has one point)
    point pt;       // the single point of this is a leaf
    bbox bound;     // bounding box for set of points in children

    kdnode *first, *second; // two children of this kd-node

    kdnode() : leaf(false), first(0), second(0) {}
    ~kdnode() { if (first) delete first; if (second) delete second; }

    // intersect a point with this node (returns squared distance)
    ntype intersect(const point &p) {
        return bound.distance(p);
    }

    // recursively builds a kd-tree from a given cloud of points
    void construct(vector<point> &vp)
    {
        // compute bounding box for points at this node
        bound.compute(vp);

        // if we're down to one point, then we're a leaf node
        if (vp.size() == 1) {
            leaf = true;
            pt = vp[0];
        }
        else {
            // split on x if the bbox is wider than high (not best heuristic...)
            if (bound.x1-bound.x0 >= bound.y1-bound.y0)
                sort(vp.begin(), vp.end(), on_x);
            // otherwise split on y-coordinate
            else
                sort(vp.begin(), vp.end(), on_y);

            // divide by taking half the array for each child
            // (not best performance if many duplicates in the middle)
            int half = vp.size()/2;
            vector<point> vl(vp.begin(), vp.begin()+half);
            vector<point> vr(vp.begin()+half, vp.end());
            first = new kdnode();   first->construct(vl);
            second = new kdnode();  second->construct(vr);
        }
    }
};

// simple kd-tree class to hold the tree and handle queries
struct kdtree
{
    kdnode *root;

    // constructs a kd-tree from a points (copied here, as it sorts them)
    kdtree(const vector<point> &vp) {
        vector<point> v(vp.begin(), vp.end());
        root = new kdnode();
        root->construct(v);
    }
    ~kdtree() { delete root; }

    // recursive search method returns squared distance to nearest point
    ntype search(kdnode *node, const point &p)
    {
        if (node->leaf) {
            // commented special case tells a point not to find itself
//            if (p == node->pt) return sentry;
//            else
                return pdist2(p, node->pt);
        }

        ntype bfirst = node->first->intersect(p);
        ntype bsecond = node->second->intersect(p);
```

```
        // choose the side with the closest bounding box to search first
        // (note that the other side is also searched if needed)
        if (bfirst < bsecond) {
            ntype best = search(node->first, p);
            if (bsecond < best)
                best = min(best, search(node->second, p));
            return best;
        }
        else {
            ntype best = search(node->second, p);
            if (bfirst < best)
                best = min(best, search(node->first, p));
            return best;
        }
    }

    // squared distance to the nearest
    ntype nearest(const point &p) {
        return search(root, p);
    }
};

int main()
{
    int n;

    while(scanf("%d", &n) && n)
    {
        vector<point> p(n);

        for(auto &it : p)
            scanf("%d %d", &it.x, &it.y);

        p.resize(unique(p.begin(), p.end()) - p.begin());

        kdtree tree(p);

        cout << tree.nearest(point(1000, 1000)) << endl;
    }

        return 0;
}
```

5.3 Geometria (reduzido)

```
typedef pair<double, double> Ponto;

bool cw(Ponto a, Ponto b, Ponto c)
{
    return (b.first - a.first) * (c.second - a.second) - (b.second - a.second) * (c.first - a.first) < 0;
}

// Retorna o casco convexo do conjunto de pontos p
vector<Ponto> convexHull(vector<Ponto> p)
{
    int n = p.size();
    if (n <= 1)
        return p;
    int k = 0;
    sort(p.begin(), p.end());
    vector<Ponto> q(n * 2);
    for (int i = 0; i < n; q[k++] = p[i++])
        for (; k >= 2 && !cw(q[k - 2], q[k - 1], p[i]); --k)
            ;
    for (int i = n - 2, t = k; i >= 0; q[k++] = p[i--])
        for (; k > t && !cw(q[k - 2], q[k - 1], p[i]); --k)
            ;
    q.resize(k - 1 - (q[0] == q[1]));
    return q;
}

//O dobro da área definida pelo triangulo de pontos pontos a, b e c (sem sinal).
double uArea2(Ponto a, Ponto b, Ponto c)
{
    return abs((b.first - a.first) * (c.second - a.second) - (b.second - a.second) * (c.first - a.first));
}

//O dobro da área definida pelo triangulo de pontos pontos a, b e c (com sinal).
double area2(Ponto a, Ponto b, Ponto c)
{
    return (b.first - a.first) * (c.second - a.second) - (b.second - a.second) * (c.first - a.first);
}
```

```cpp
//Distância entre os pontos a e b
double dist(Ponto a, Ponto b)
{
    return hypot(a.first - b.first, a.second - b.second);
}
//Intersecão de semi-retas (p1 -> p2), (p3 -> p4)
bool segIntercept(Ponto p1, Ponto p2, Ponto p3, Ponto p4)
{
    return cw(p1, p2, p3) != cw(p1, p2, p4) && cw(p3, p4, p1) != cw(p3, p4, p2);
}
//Retorna a área do polígono p
double polygonArea(vector<Ponto> p)
{
    double s = 0.0;
    for (int i = 0; i < p.size(); i++)
        s += area2(Ponto(0, 0), p[i], p[(i + 1) % p.size()]);
    return fabs(s / 2.0);
}
//Retorna a área do polígono p definido pelos pontos p[i, f]
double polygonArea2(vector<Ponto> p, int i, int f)
{
    double s = 0.0;
    Ponto primeiro = p[i];
    for (; i != f; i++)
        s += area2(Ponto(0, 0), p[i], p[(i + 1)]);
    s += area2(Ponto(0, 0), p[i], primeiro);
    return fabs(s / 2.0);
}
//Retorna a menor largura do conjunto de pontos p
double raio(vector<Ponto> p)
{
    vector<Ponto> h = convexHull(p);
    int m = h.size();
    if (m == 1)
        return 0;
    if (m == 2)
        return 0;
    int k = 1;
    while (uArea2(h[m - 1], h[0], h[(k + 1) % m]) > uArea2(h[m - 1], h[0], h[k]))
        ++k;
    double res = 10000000;
    for (int i = 0, j = k; i <= k && j < m; i++)
    {
        res = min(res, dist(h[i], h[j]));
        while (j < m && uArea2(h[i], h[(i + 1) % m], h[(j + 1) % m]) > uArea2(h[i], h[(i + 1) % m], h[j]))
        {
            res = min(res, dist(h[i], h[(j + 1) % m]));
            ++j;
        }
    }
    return res;
}
//Retorna a maior largura do conjunto de pontos p
double diametro(vector<Ponto> p)
{
    vector<Ponto> h = convexHull(p);
    int m = h.size();
    if (m == 1)
        return 0;
    if (m == 2)
        return dist(h[0], h[1]);
    int k = 1;
    while (uArea2(h[m - 1], h[0], h[(k + 1) % m]) > uArea2(h[m - 1], h[0], h[k]))
        ++k;
    double res = 0;
    for (int i = 0, j = k; i <= k && j < m; i++)
    {
        res = max(res, dist(h[i], h[j]));
        while (j < m && uArea2(h[i], h[(i + 1) % m], h[(j + 1) % m]) > uArea2(h[i], h[(i + 1) % m], h[j]))
        {
            res = max(res, dist(h[i], h[(j + 1) % m]));
            ++j;
        }
    }
    return res;
}
```

5.4 Geometria (grande)

```
#include <bits/stdc++.h>
using namespace std;
const double EPS = 1e-10;
inline int cmp( double x, double y = 0, double tol = EPS ) {
    return (x <= y + tol ) ? ( x + tol < y ) ? -1 : 0 : 1;
}
struct Point {
    double x, y;
    Point( double x = 0, double y = 0 ) : x( x ), y( y ) {}
    Point operator+( Point q ) const {
        return Point( x + q.x, y + q.y );
    }
    Point operator-( Point q ) const {
        return Point( x - q.x, y - q.y );
    }
    Point operator*( double t ) const {
        return Point( x * t, y * t );
    }
    Point operator/( double t ) const {
        return Point( x / t, y / t );
    }
    double operator*( Point q )const {
        return x * q.x + y * q.y;
    }
    double operator^( Point q ) const {
        return x * q.y - y * q.x;
    }
    int cmp( Point q ) const {
        if ( int t = ::cmp( x, q.x ) )
                return t;
        return ::cmp( y, q.y );
    }
    bool operator==( Point q ) const {
        return cmp( q ) == 0;
    }
    bool operator!=( Point q ) const {
        return cmp( q ) != 0;
    }
    bool operator<( Point q ) const {
        return cmp( q ) < 0;
    }
    static Point pivot;
};
Point Point::pivot;
typedef vector<Point> Polygon;
inline double abs( Point& p ) {
    return hypot( p.x, p.y );
}
inline double arg( Point& p ) {
    return atan2( p.y, p.x );
}
//Verifica o sinal do produto vetorial entre os vetores (p-r) e (q - r)
inline int ccw( Point& p, Point& q, Point& r ) {
    return cmp( ( p - r ) ^ ( q - r ) );
}
//calcula o angulo orientado entre os vetores (p-q) e (r - q)
inline double angle( Point& p, Point &q, Point& r ) {
    Point u = p - q, w = r - q;
    return atan2( u ^ w, u * w );
}
//Decide se o ponto p esta sobre a reta que passa por p1p2.
```

```cpp
bool pointoSobreReta( Point& p1, Point &p, Point& p2 ) {
    return ccw( p1, p2, p ) == 0;
}

//Decide de p esta sobre o segmento p1p2
bool between( Point& p1, Point &p, Point& p2 ) {
    return ccw( p1, p2, p ) == 0 && cmp( ( p1 - p ) * ( p2 - p ) ) <= 0;
}

//Calcula a distancia do ponto p a reta que passa por p1p2
double retaDistance( Point& p1, Point& p2, Point &p ) {
    Point A = p1 - p, B = p2 - p1;
    return fabs( A ^ B ) / sqrt( B * B );
}

//Calcula a distancia do ponto p ao segmento de reta que passa por p1p2
double segDistance( Point& p1, Point& p2, Point &p ) {
    Point A = p1 - p, B = p1 - p2, C = p2 - p;
    double a = A * A, b = B * B, c = C * C;
    if ( cmp( a, b + c ) >= 0 ) return sqrt( c );
    if ( cmp( c, a + b ) >= 0 )return sqrt( a );
    return fabs( A ^ C ) / sqrt( b );

}
//Calcula a area orientada do poligono T.
double polygonArea( Polygon& T ) {
    double s = 0.0;
    int n = T.size( );
    for ( int i = 0; i < n; i++ )
    {
        s += T[i] ^ T[( i + 1 ) % n];
    }
    return s / 2.0; //Retorna a area com sinal
}
//Classifica o ponto p em relacao ao poligono T dependendo se ele está
//na fronteira (-1) no exterior (0) ou no interior (1).
int inpoly( Point& p, Polygon& T ) {
    //-1 sobre, 0 fora, 1 dentro
    double a = 0.0;
    int n = T.size( );
    for ( int i = 0; i < n; i++ )
    {
        if ( between( T[i], p, T[( i + 1 ) % n] ) ) return -1;
        a += angle( T[i], p, T[( i + 1 ) % n] );
    }
    return cmp( a ) != 0;
}
//Ordenacao radial.
bool radialSort( Point p, Point q ) {
    Point P = p - Point::pivot, Q = q - Point::pivot;
    double R = P ^ Q;
    if ( cmp( R ) ) return R > 0;
    return cmp( P * P, Q * Q ) < 0;
}

//Determina o convex hull de T. ATENCAO. A lista de pontos T e destruida.
Polygon convexHull( vector<Point>& T ) {
    int j = 0, k, n = T.size( );
    Polygon U( n );
    Point::pivot = *min_element( T.begin( ), T.end( ) );
    sort( T.begin( ), T.end( ), radialSort );

    for ( k = n - 2; k >= 0 && ccw( T[0], T[n - 1], T[k] ) == 0; k-- );
    reverse( ( k + 1 ) + T.begin( ), T.end( ) );

    for ( int i = 0; i < n; i++ )
    {
        // troque o >= por > para manter pontos colineares
        while (j > 1 && ccw( U[j - 1], U[j - 2], T[i] ) >= 0) j--;
        U[j++] = T[i];
    }
    U.resize( j );
    return U;
}
//Intersecão de semi-retas (p1 -> p2), (p3 -> p4)
bool segIntercept(Point p1, Point p2, Point p3, Point p4) {
        return ccw(p1, p2, p3) != ccw(p1, p2, p4) && ccw(p3, p4, p1) != ccw(p3, p4, p2);
}
```

5.5 Geometria (Marcelo)

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

#define D(x) cout << #x " = " << x << endl

//INT_MAX, UINT_MAX, LLONG_MAX, ULLONG_MAX, INFINITY

// Mudar para double dependendo do exercicio
typedef double Double;

static const Double EPS = 1e-10;

// Compara doubles
int cmp(Double x, Double y = 0.0, Double tol = EPS) {
    return (x <= y + tol) ? (x + tol <= y) ? -1 : 0 : 1;
}

struct Vec3 {
    Double x, y, z;

    Vec3() : x(0.0), y(0.0), z(0.0) {}
    Vec3(Double x, Double y, Double z) : x(x), y(y), z(z) {}
    Vec3(const Vec3& u, const Vec3& v) : x(v.x - u.x), y(v.y - u.y), z(v.z - u.z) {}
    Vec3(const Vec3& v) : x(v.x), y(v.y), z(v.z) {}

    void operator=(const Vec3& v) {
        x = v.x;
        y = v.y;
        z = v.z;
    }

    //----------------------------------------------------- OPERADORES VETORIAIS
    Vec3 operator+(const Vec3& v) const {
        return Vec3(x + v.x, y + v.y, z + v.z);
    }

    Vec3 operator-(const Vec3& v) const {
        return Vec3(x - v.x, y - v.y, z - v.z);
    }

    // Produto escalar
    double operator*(const Vec3& v) const {
        return x * v.x + y * v.y + z * v.z;
    }

    // Produto vetorial
    Vec3 operator^(const Vec3& v) const {
        return Vec3(y * v.z - z * v.y, z * v.x - x * v.z, x * v.y - y * v.x);
    }
    //-------------------------------------------------------------------------

    //------------------------------------------------- OPERADORES COM ESCALARES
    Vec3 operator+(const Double c) const {
        return Vec3(x + c, y + c, z + c);
    }

    Vec3 operator-(const Double c) const {
        return Vec3(x - c, y - c, z - c);
    }

    Vec3 operator*(const Double c) const {
        return Vec3(x * c, y * c, z * c);
    }

    Vec3 operator/(const Double c) const {
        assert(::cmp(c) != 0);
        return Vec3(x / c, y / c, z / c);
    }
    //------------------------------------------------------------------------

    //------------------------------------------------------------------ NORMAS
    // Retorna a norma
    Double norma() const {
        return sqrt(x * x + y * y + z * z);
    }

    // Retorna a norma ao quadrado
    Double norma2() const {
        return x * x + y * y + z * z;
    }

    // Retorna uma copia normalizada do vetor
    Vec3 normalizado() const {
```

```
        return *this/this->norma();
    }
    //------------------------------------------------------------------------

    //--------------------------------------------------------------- COMPARADORES
    //***************** CUIDADO - ADAPTAR AO PROBLEMA ************************
    bool operator==(const Vec3& v) const {
        return !cmp(x, v.x) && !cmp(y, v.y) && !cmp(z, v.z);
    }

    bool operator!=(const Vec3& v) const {
        return !(*this == v);
    }

    // Critehrios de comparacao
    bool operator<(const Vec3& v) const {
        return cmpMenorXYZ(v);
    }

    // Compara componentes na ordem x, y, z.
    // Retorna true assim que encontrar a primeira menor
    bool cmpMenorXYZ(const Vec3& v) const {
        int aux = cmp(x, v.x);

        if(aux < 0)
            return true;
        else if(aux == 0) {
            aux = cmp(y, v.y);

            if(aux < 0 || aux == 0 && cmp(z, v.z) < 0)
                return true;
        }

        return false;
    }
    //------------------------------------------------------------------------

    //--------------------------------------------------------------------- DEBUG
    friend ostream& operator<<(ostream& os, const Vec3& v) {
        return os << "(" << v.x << ", " << v.y << ", " << v.z << ") ";
    }
    //------------------------------------------------------------------------
};

//--------------------------------------------------------------- AUXILIARES
// Retorna se os pontos a, b, c estao em sentido horario
bool cw(Vec3 a, Vec3 b, Vec3 c) {
    return cmp((b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x)) < 0;
}

//O dobro da area definida pelo triangulo de pontos pontos a, b e c (com sinal).
Double area2(Vec3 a, Vec3 b, Vec3 c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

//Retorna a area do poligono p
Double areaPoligono(vector<Vec3>& p) {
    Double s = 0.0;
    for (unsigned int i = 0; i < p.size(); i++)
        s += area2(Vec3(), p[i], p[(i + 1) % p.size()]);

    return fabs(s / 2.0);
}

// Retorna a projecao de u em v
Vec3 projecao(const Vec3& u, const Vec3& v) {
    assert(v.x || v.y || v.z);
    return v * (u * v / v.norma2());
}

// (u ^ v) * w = [[ux, uy, uz], [vx, vy, vz], [wx, wy, wz]]
Double produtoMisto(const Vec3& u, const Vec3& v, const Vec3& w) {
    return (u ^ v) * w;
}
//--------------------------------------------------------------------------

//--------------------------------------------------------------- DISTÂNCIAS
// Retorna a distância do ponto p ao segmento ab
Double distPontoSegmento(const Vec3 &p, const Vec3 &a, const Vec3 &b) {
    Vec3 u = b - a, v = p - a;
    Double t = (u * v) / (u * u);
    if(cmp(t) == -1) t = 0.0;
    if(cmp(t, 1.0) == 1) t = 1.0;
    return (p - Vec3(a + u * t)).norma();
}
```

```
// Retorna a menor distância entre um ponto qualquer de a1b1 com um ponto qualquer de a2b2
Double distSegmentoSegmento(const Vec3 &a1, const Vec3 &b1, const Vec3 &a2, const Vec3 &b2) {
    Vec3 u = b1 - a1,  v = b2 - a2, w = a1 - a2;
    Double a = u * u, b = u * v, c = v * v, d = u * w, e = v * w, den = a * c - b * b, t1, t2;
    if(cmp(den) == 0) {
        t1 = 0;
        t2 = d / b;
    }
    else {
        t1 = (b * e - c * d) / den;
        t2 = (a * e - b * d) / den;
    }
    if(0 <= t1 && t1 <= 1 && 0 <= t2 && t2 <= 1) {
        Vec3 p = a1 + u * t1, q = a2 + v * t2;
        return (p - q).norma();
    }
    else {
        Double option1 = min(distPontoSegmento(a2, a1, b1), distPontoSegmento(b2, a1, b1));
        Double option2 = min(distPontoSegmento(a1, a2, b2), distPontoSegmento(b1, a2, b2));
        return min(option1, option2);
    }
}
// Retorna a menor distância entre o ponto p ao triângulo t1t2t3
Double distPontoTriangulo(const Vec3 &p, const Vec3 &t1, const Vec3 &t2, const Vec3 &t3) {
    Vec3 u = t2 - t1, v = t3 - t1, n = u ^ v;
    // Se falhar os pontos do triângulo sao colineares
    assert(cmp(n * n) != 0);
    Double s = -(n * (p - t1)) / (n * n);
    // q eh o ponto do plano do triângulo mais proximo de p
    Vec3 q = p + n * s;
    // Verificando se q esta dentro do triângulo
    Vec3 w = q - t1;
    Vec3 nv = n ^ v;
    Vec3 nu = n ^ u;
    Double a2 = (w * nv) / (u * nv);
    Double a3 = (w * nu) / (v * nu);
    Double a1 = 1 - a2 - a3;
    // Temos as coordenadas baricêntricas de q. q == a1*t1 + a2*t2 + a3*t3.
    if (0 <= a1 && a1 <= 1 && 0 <= a2 && a2 <= 1 && 0 <= a3 && a3 <= 1) {
                // O ponto esta dentro do triângulo ou em sua borda.
                // Basta retornar a distância de p a q
        return (p - q).norma();
    }
    else {
                // O ponto mais proximo esta no plano do triângulo.
        Double ans = distPontoSegmento(p, t1, t2);
        ans = min(ans, distPontoSegmento(p, t2, t3));
        ans = min(ans, distPontoSegmento(p, t3, t1));
        return ans;
    }
}
//------------------------------------------------------------------------------
//-------------------------------------------------------------- CASCO CONVEXO
// Retorna o casco convexo do conjunto de pontos p em sentido ANTI-HORaRIO
vector<Vec3> convexHull(vector<Vec3>& p) {
    int n = p.size();
    if (n <= 1)
        return p;
    int k = 0;
    // CUIDADO COM O OPERADOR <
    sort(p.begin(), p.end());
    vector<Vec3> q(n * 2);
    for (int i = 0; i < n; q[k++] = p[i++])
        for (; k >= 2 && cw(q[k - 2], q[k - 1], p[i]); --k)
            ;
    for (int i = n - 2, t = k; i >= 0; q[k++] = p[i--])
        for (; k > t && cw(q[k - 2], q[k - 1], p[i]); --k)
            ;
    q.resize(k - 1 - (q[0] == q[1]));
    return q;
```

```
}
//------------------------------------------------------------------------------

int main() {
        Vec3 u(1, 0, 0), v(2, 0, 0), p(1, 0, 1), q(0, 0, 0);
        D(distPontoTriangulo(q, u, v, p));
}
```

6 Estruturas de dados etc

6.1 Wavelet-tree

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

const int N = 10000;

struct KthSmallest
{
	struct Seg
	{
		int l, r, mid;

		void set(int _l, int _r)
		{
			l = _l;
			r = _r;
			mid = l + r >> 1;
		}
	} seg[N << 2];

	int b[25][N], left[25][N], sorted[N];

	void init(int *a, int n)
	{
		for (int i = 0; i < n; i++)
			b[0][i] = sorted[i] = a[i];

		sort(sorted, sorted + n);
		build(0, n, 0, 1);
	}

	void build(int l, int r, int d, int idx)
	{
		seg[idx].set(l, r);

		if (l + 1 == r)
			return;

		int mid = seg[idx].mid;
		int lsame = mid - l;

		for (int i = l; i < r; i++)
			if (b[d][i] < sorted[mid])
				lsame--;

		int lpos = l, rpos = mid, same = 0;

		for (int i = l; i < r; ++i)
		{
			left[d][i] = (i != l ? left[d][i - 1] : 0);

			if (b[d][i] < sorted[mid])
			{
				left[d][i]++;
				b[d + 1][lpos++] = b[d][i];
			}
			else if (b[d][i] > sorted[mid])
				b[d + 1][rpos++] = b[d][i];
			else
			{
				if (same < lsame)
				{
					same++;
					left[d][i]++;
					b[d + 1][lpos++] = b[d][i];
				}
				else
				{
					b[d + 1][rpos++] = b[d][i];
				}
			}
		}

		build(l, mid, d + 1, idx << 1);
		build(mid, r, d + 1, idx << 1 | 1);
	}

	//Quando ordernarmos [l, r), qual é o k-ésimo termo?
	int kth(int l, int r, int k, int d = 0, int idx = 1)
	{ // k : 1-origin!!!
		if (l + 1 == r)
			return b[d][l];
```

```
			int ltl = (l != seg[idx].l ? left[d][l - 1] : 0);
			int tl = left[d][r - 1] - ltl;

			if (tl >= k)
			{
				int newl = seg[idx].l + ltl;
				int newr = seg[idx].l + ltl + tl;

				return kth(newl, newr, k, d + 1, idx << 1);
			}
			else
			{
				int mid = seg[idx].mid;
				int tr = r - l - tl;
				int ltr = l - seg[idx].l - ltl;
				int newl = mid + ltr;
				int newr = mid + ltr + tr;

				return kth(newl, newr, k - tl, d + 1, idx << 1 | 1);
			}
		}

		//When sorting [l, r), what number will x come in?
		//If there are two or more x's, return the rank of the last one.
		//If there is no x, return the rank of the largest but less than x.
		//When there is no less than x, 0 is returned.
		int rank(int l, int r, int x, int d = 0, int idx = 1)
		{
			if (seg[idx].l + 1 == seg[idx].r)
				return l + 1 == r && sorted[l] <= x;

			int ltl = (l != seg[idx].l ? left[d][l - 1] : 0);
			int tl = left[d][r - 1] - ltl;
			int mid = seg[idx].mid;

			if (x < sorted[mid])
			{
				int newl = seg[idx].l + ltl;
				int newr = seg[idx].l + ltl + tl;

				return rank(newl, newr, x, d + 1, idx << 1);
			}
			else
			{
				int tr = r - l - tl;
				int ltr = l - seg[idx].l - ltl;
				int newl = mid + ltr;
				int newr = mid + ltr + tr;

				return tl + rank(newl, newr, x, d + 1, idx << 1 | 1);
			}
		}

		// Quantos x existem entre [l,r)
		int freq(int l, int r, int x)
		{
			return rank(l, r, x) - rank(l, r, x - 1);
		}
} kth;

int main()
{
	int a[8] = {6, 12, 5, 17, 10, 2, 7, 3};

	kth.init(a, 8);

	cout << kth.kth(2, 7, 3) << endl;  // 7
	cout << kth.rank(2, 7, 7) << endl; // 3
}
```

6.2 Seg-tree 2D

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

using namespace std;

#define Max 501
#define INF (1 << 30)
int P[Max][Max]; // container for 2D grid

/* 2D Segment Tree node */
struct Point
```

```
{
    int x, y, mx;
    Point() {}
    Point(int x, int y, int mx) : x(x), y(y), mx(mx) {}

    bool operator<(const Point &other) const
    {
        return mx < other.mx;
    }
};

struct Segtree2d
{
    // I didn't calculate the exact size needed in terms of 2D container size.
    // If anyone, please edit the answer.
    // It's just a safe size to store nodes for MAX * MAX 2D grids which won't cause stack overflow :)
    Point T[500000]; // TODO: calculate the accurate space needed

    int n, m;

    // initialize and construct segment tree
    void init(int n, int m)
    {
        this->n = n;
        this->m = m;
        build(1, 1, 1, n, m);
    }

    // build a 2D segment tree from data [ (a1, b1), (a2, b2) ]
    // Time: O(n logn)
    Point build(int node, int a1, int b1, int a2, int b2)
    {
        // out of range
        if (a1 > a2 or b1 > b2)
            return def();

        // if it is only a single index, assign value to node
        if (a1 == a2 and b1 == b2)
            return T[node] = Point(a1, b1, P[a1][b1]);

        // split the tree into four segments
        T[node] = def();
        T[node] = maxNode(T[node], build(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 + b2) / 2));
        T[node] = maxNode(T[node], build(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2, (b1 + b2) / 2));
        T[node] = maxNode(T[node], build(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 + a2) / 2, b2));
        T[node] = maxNode(T[node], build(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2) / 2 + 1, a2, b2));
        return T[node];
    }

    // helper function for query(int, int, int, int);
    Point query(int node, int a1, int b1, int a2, int b2, int x1, int y1, int x2, int y2)
    {
        // if we out of range, return dummy
        if (x1 > a2 or y1 > b2 or x2 < a1 or y2 < b1 or a1 > a2 or b1 > b2)
            return def();

        // if it is within range, return the node
        if (x1 <= a1 and y1 <= b1 and a2 <= x2 and b2 <= y2)
            return T[node];

        // split into four segments
        Point mx = def();
        mx = maxNode(mx, query(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 + b2) / 2, x1, y1, x2, y2));
        mx = maxNode(mx, query(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2, (b1 + b2) / 2, x1, y1, x2, y2));
        mx = maxNode(mx, query(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 + a2) / 2, b2, x1, y1, x2, y2));
        mx = maxNode(mx, query(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2) / 2 + 1, a2, b2, x1, y1, x2, y2));

        // return the maximum value
        return mx;
    }

    // query from range [ (x1, y1), (x2, y2) ]
    // Time: O(logn)
    Point query(int x1, int y1, int x2, int y2)
    {
        return query(1, 1, 1, n, m, x1, y1, x2, y2);
    }

    // helper function for update(int, int, int);
    Point update(int node, int a1, int b1, int a2, int b2, int x, int y, int value)
    {
        if (a1 > a2 or b1 > b2)
            return def();
```

```
        if (x > a2 or y > b2 or x < a1 or y < b1)
            return T[node];

        if (x == a1 and y == b1 and x == a2 and y == b2)
            return T[node] = Point(x, y, value);

        Point mx = def();
        mx = maxNode(mx, update(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 + b2) / 2, x, y, value));
        mx = maxNode(mx, update(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2, (b1 + b2) / 2, x, y, value));
        mx = maxNode(mx, update(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 + a2) / 2, b2, x, y, value));
        mx = maxNode(mx, update(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2) / 2 + 1, a2, b2, x, y, value));
        return T[node] = mx;
    }

    // update the value of (x, y) index to 'value'
    // Time: O(logn)
    Point update(int x, int y, int value)
    {
        return update(1, 1, 1, n, m, x, y, value);
    }

    // utility functions; these functions are virtual because they will be overridden in child class
    virtual Point maxNode(Point a, Point b)
    {
        return max(a, b);
    }

    // dummy node
    virtual Point def()
    {
        return Point(0, 0, -INF);
    }
};

/* 2D Segment Tree for range minimum query; a override of Segtree2d class */
struct Segtree2dMin : Segtree2d
{
    // overload maxNode() function to return minimum value
    Point maxNode(Point a, Point b)
    {
        return min(a, b);
    }

    Point def()
    {
        return Point(0, 0, INF);
    }
};

// initialize class objects
Segtree2d Tmax;
Segtree2dMin Tmin;

/* Drier program */
int main(void)
{
    int n, m;
    // input
    scanf("%d %d", &n, &m);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            scanf("%d", &P[i][j]);

    // initialize
    Tmax.init(n, m);
    Tmin.init(n, m);

    // query
    int x1, y1, x2, y2;
    scanf("%d %d %d %d", &x1, &y1, &x2, &y2);

    Tmax.query(x1, y1, x2, y2).mx;
    Tmin.query(x1, y1, x2, y2).mx;

    // update
    int x, y, v;
    scanf("%d %d %d", &x, &y, &v);
    Tmax.update(x, y, v);
    Tmin.update(x, y, v);

    return 0;
}
```

6.3 Seg-tree

```
#include <algorithm>

using namespace std;

#define MAX 1000000 // O valor aqui tem que ser >= 2 * tamanho do maior n
#define INF 1 << 28

// Não necessáriamente é um int, pode ser uma segtree de struct etc;
int init[MAX], tree[MAX], lazy[MAX];

void build_tree(int node, int a, int b)
{
	if (a > b)
		return;

	// Se folha
	if (a == b)
	{
		tree[node] = init[a];
		lazy[node] = 0;
		return;
	}

	build_tree(node * 2, a, (a + b) / 2);
	build_tree(node * 2 + 1, 1 + (a + b) / 2, b);

	// Se nó
	tree[node] = tree[node * 2] + tree[node * 2 + 1];
	lazy[node] = 0;
}

void update_tree(int node, int a, int b, int i, int j, int value)
{
	// Se fora do intervalo - retorna
	if (a > b || a > j || b < i)
		return;

	if (lazy[node] != 0)
	{
		//Atualizacão atrasada.
		tree[node] += lazy[node];

		// Passa lazy para filhos
		if (a != b)
		{
			lazy[node * 2] += lazy[node];
			lazy[node * 2 + 1] += lazy[node];
		}

		//Reseta o nó
		lazy[node] = 0;
	}

	// Se o nó atual cobre todo o intervalo
	if (a >= i && b <= j)
	{
		tree[node] += value;

		if (a != b)
		{
			lazy[node * 2] += value;
			lazy[node * 2 + 1] += value;
		}

		return;
	}

	// Se tem um pedaco em cada filho.

	// Atualiza os filhos.
	update_tree(node * 2, a, (a + b) / 2, i, j, value);
	update_tree(1 + node * 2, 1 + (a + b) / 2, b, i, j, value);

	// Atualiza o pai.
	tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

int query_tree(int node, int a, int b, int i, int j)
{
	// Se fora do intervalo
	if (a > b || a > j || b < i)
	{
		//Aqui deverá ser retornado o elemento neutro para a operacão desejada
```

```
		return 0;
	}
	if (lazy[node] != 0)
	{
		//Atualizacão atrasada.
		tree[node] += lazy[node];

		//Se não folha, passa lazy pros filhos
		if (a != b)
		{
			lazy[node * 2] += lazy[node];
			lazy[node * 2 + 1] += lazy[node];
		}

		//Reseta o nó
		lazy[node] = 0;
	}
	// Se o nó cobre o intervalo
	if (a >= i && b <= j)
		return tree[node];

	// Se o intervalo está um pedaco em cada filho.
	int q1 = query_tree(node * 2, a, (a + b) / 2, i, j);
	int q2 = query_tree(1 + node * 2, 1 + (a + b) / 2, b, i, j);

	// Retorna a combinacão dos intervalos.
	return q1 + q2;
}
/*
Uso:

Assumindo que "n" é o numero de termos que o segmento tem
Inicialize "init" com os valores iniciais:

*	for(i = 0; i < n; scanf("%d", val), i++)
*	init[i] = val;

E mande construir a arvore:

*   build_tree(1, 0, n-1);

Para atualizar a arvore:

*   update_tree(1, 0, n-1, inicio, fim, val);
*   Onde inicio é a posicão inicial do segmento desejado e fim é a posicão final do mesmo
*	e val é o quanto você quer alterar os valores desse seguimento

Para fazer queries

*   query_tree(1, 0, n-1, inicio, fim);
*	Onde inicio é a posicão inicial do segmento desejado e fim é a posicão final do mesmo
*	o retorno terá o mesmo tipo que os dados guardados na arvore e será o resultado do segmento pesquisado

*/
```

6.4 Mergesort

```
typedef vector<int>::iterator vec_it;

void merge(vec_it left, vec_it left_end, vec_it right, vec_it right_end, vec_it numbers)
{
    while (left != left_end)
    {
        if (*left < *right || right == right_end)
        {
            *numbers = *left;
            ++left;
        }
        else
        {
            *numbers = *right;
            ++right;
        }

        ++numbers;
    }

    while (right != right_end)
    {
        *numbers = *right;
        ++right;
```

```
        ++numbers;
    }
}
void merge_sort(vector<int> &numbers)
{
    if (numbers.size() <= 1)
        return;
    vector<int>::size_type middle = numbers.size() / 2;
    vector<int> left(numbers.begin(), numbers.begin() + middle);
    vector<int> right(numbers.begin() + middle, numbers.end());
    merge_sort(left);
    merge_sort(right);
    merge(left.begin(), left.end(), right.begin(), right.end(), numbers.begin());
}
```

6.5 Algoritmo de MO (queries offline)

```
#define N 311111
#define A 1111111
#define BLOCK 555 // ~sqrt(N)
int cnt[A], a[N], ans[N], answer = 0;
struct node
{
        int L, R, i;
} q[N];
bool cmp(node x, node y)
{
        if (x.L / BLOCK != y.L / BLOCK)
        {
                // different blocks, so sort by block.
                return x.L / BLOCK < y.L / BLOCK;
        }
        // same block, so sort by R value
        return x.R < y.R;
}
void add(int position)
{
        cnt[a[position]]++;
        if (cnt[a[position]] == 1)
        {
                answer++; // Verifica se é resposta aqui!!!
        }
}
void remove(int position)
{
        cnt[a[position]]--;
        if (cnt[a[position]] == 0)
        {
                answer--; // Verifica se é resposta aqui!!!
        }
}
int main()
{
        int n;
        scanf("%d", &n);
        for (int i = 0; i < n; i++)
                scanf("%d", &a[i]);
        int m;
        scanf("%d", &m);
        for (int i = 0; i < m; i++)
        {
                scanf("%d%d", &q[i].L, &q[i].R);
                q[i].L--;
                q[i].R--;
                q[i].i = i;
        }
        sort(q, q + m, cmp);
        int currentL = 0, currentR = 0;
        for (int i = 0; i < m; i++)
        {
```

```
            int L = q[i].L, R = q[i].R;
            while (currentL < L)
            {
                    remove(currentL);
                    currentL++;
            }
            while (currentL > L)
            {
                    add(currentL - 1);
                    currentL--;
            }
            while (currentR <= R)
            {
                    add(currentR);
                    currentR++;
            }
            while (currentR > R + 1)
            {
                    remove(currentR - 1);
                    currentR--;
            }
            ans[q[i].i] = answer;
        }
        for (int i = 0; i < m; i++)
            printf("%d\n", ans[i]);
}
```

6.6 Union-find

```
// Tamanho máximo de n
const int maxn = 200000;
int Rank[maxn], p[maxn], n;

void init(int _n)
{
    n = _n;
    fill(Rank, Rank + n, 0);
    for (int i = 0; i < n; i++)
        p[i] = i;
}

int find(int x)
{
    return x == p[x] ? x : (p[x] = find(p[x]));
}

void unir(int a, int b)
{
    a = find(a);
    b = find(b);
    if (a == b)
        return;
    if (Rank[a] < Rank[b])
        swap(a, b);
    if (Rank[a] == Rank[b])
        ++Rank[a];
    p[b] = a;
}
```