Notebook UNosnovatos

Contents

1 C++

1.1 C++ plantilla

```
#include <bits/stdc++.h>
using namespace std;
#define sz(arr) ((int) arr.size())
typedef long long ll;
typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
typedef vector<long long> vl;
const int INF = 1e9;
const ll INFL = 1e18;
const int MOD = 1e9+7;
const double EPS  = 1e-9;
int dirx[4] = {0,-1,1,0};
int diry[4] = {-1,0,0,1};
int dr[] = {1, 1, 0, -1, -1, -1, 0, 1};
int dc[] = {0, 1, 1, 1, 0, -1, -1, -1};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    // freopen("file.in", "r", stdin);
    // freopen("file.out", "w", stdout);

    return 0;
}
```

2 Estructuras de Datos

2.1 Disjoint Set Union

```
struct dsu{
    vi p, size;
    int num_sets;
    int maxSize;

    dsu(int n){
        p.assign(n, 0);
        size.assign(n, 1);
        num_sets = n;
        for (int i = 0; i<n; i++) p[i] = i;
    }

    int find_set(int i) {return (p[i] == i) ? i : (p[i] =
         find_set(p[i]));}
```

```
    bool is_same_set(int i, int j) {return find_set(i) ==
         find_set(j);}

    void unionSet(int i, int j){
            if (!is_same_set(i, j)){
                int a = find_set(i), b = find_set(j);
                if (size[a] < size[b])
                    swap(a, b);
                p[b] = a;
                size[a] += size[b];
                maxSize = max(size[a], maxSize);
                num_sets--;
            }
        }
};
```

2.2 Fenwick Tree

```
#define LSOne(S) ((S) & -(S))

struct fenwick_tree{
    vl ft; int n;
    fenwick_tree(int n): n (n){ft.assign(n+1, 0);}
    ll rsq(int j){
        ll sum = 0;
        for(;j;j -= LSOne(j)) sum += ft[j];
        return sum;
    }
    ll rsq(int i, int j) {return rsq(j) - (i == 1 ? 0 :
       rsq(i-1));}
    void upd(int i, ll v){
        for (; i <= n; i += LSOne(i)) ft[i] += v;
    }
};
```

2.3 Segment Tree

```
int nullValue = 0;

struct nodeST{
    nodeST *left, *right;
    int l, r; ll value, lazy, lazy1;

    nodeST(vi &v, int l, int r) : l(l), r(r){
        int m = (l+r)>>1;
        lazy = 0;
        lazy1 = 0;
        if (l!=r){
            left = new nodeST(v, l, m);
            right = new nodeST(v, m+1, r);
            value = opt(left->value, right->value);
        }
        else{
            value = v[l];
        }
    }

    ll opt(ll leftValue, ll rightValue){
        return leftValue + rightValue;
    }

    void propagate(){
        if(lazy1){
            value = lazy1 * (r-l+1);
            if (l != r){
                left->lazy1 = lazy1, right->lazy1 = lazy1
                    ;
                left->lazy = 0, right->lazy = 0;
            }
            lazy1 = 0;
            lazy = 0;
        }
        else{
            value += lazy * (r-l+1);
            if (l != r){
                if(left->lazy1) left->lazy1 += lazy;
                else left->lazy += lazy;
                if(right->lazy1) right->lazy1 += lazy;
                else right->lazy += lazy;
            }
            lazy = 0;
        }
    }

    ll get(int i, int j){
        propagate();
        if (l>=i && r<=j) return value;
        if (l>j  || r<i) return nullValue;

        return opt(left->get(i, j), right->get(i, j));
    }

    void upd(int i, int j, int nv){
        propagate();
        if (l>j  || r<i) return;
        if (l>=i && r<=j){
            lazy += nv;
            propagate();
            // value = nv;
            return;
        }

        left->upd(i, j, nv);
        right->upd(i, j, nv);

        value = opt(left->value, right->value);
    }

    void upd(int k, int nv){
        if (l>k  || r<k) return;
```

```
        if (l>=k && r<=k){
            value = nv;
            return;
        }

        left->upd(k, nv);
        right->upd(k, nv);

        value = opt(left->value, right->value);
    }

    void upd1(int i, int j, int nv){
        propagate();
        if (l>j  || r<i) return;
        if (l>=i && r<=j){
            lazy = 0;
            lazy1 = nv;
            propagate();
            return;
        }

        left->upd1(i, j, nv);
        right->upd1(i, j, nv);

        value = opt(left->value, right->value);
    }
};
```

3 Grafos

3.1 DFS

```
//O(V+E)
int vertices, aristas;

vector<int> dfs_num(vertices+1, -1); //Vector del estado
    de cada vertice (visitado o no visitado)

const int NO_VISITADO = -1;
const int VISITADO = 1;

vector<vector<int>> adj(vertices + 1); //Lista adjunta
    del grafo

// Complejidad O(V + E)
void dfs(int v){
    dfs_num[v] = VISITADO;
    //Se recorren los vecinos
    for (int i = 0; i < (int) adj[v].size(); i++){

        if (dfs_num[adj[v][i]] == NO_VISITADO){
            dfs(adj[v][i]);
        }

    }
}
```

3.2 BFS

```
//BFS, complejidad O(V + E)
queue<int> q; q.push(adj[1][0]);    //Origen
vi d(n+1, INT_MAX); d[adj[1][0]] = 0;   //La distancia
    del vertice a el mismo es cero
while(!q.empty()){
    int nodo = q.front(); q.pop();
    for (int i = 0; i<(int)adj[nodo].size(); i++){
        if (d[adj[nodo][i]] == INT_MAX){   //Si el vecino
            no visitado y alcanzable
            d[adj[nodo][i]] = d[nodo] + 1;    //Hacer d[
                adj[u][i]] != INT_MAX para etiquetarlo
            q.push(adj[nodo][i]);          //Anadiendo a
                la cola para siguiente iteracion
        }
    }
}
}
```

3.3 Puntos de articulacion y puentes

```
//Puntos de articulacion: son vertices que desconectan el
    grafo
//Puentes: son aristas que desconectan el grafo
//Usar para grafos dirigidos
//O(V+E)
vi dfs_num, dfs_low, dfs_parent, articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;
vector<vii> adj;
void articulationPointAndBridge(int u) {
    dfs_num[u] = dfsNumberCounter++;
    dfs_low[u] = dfs_num[u]; // dfs_low[u]<=dfs_num[u]
    for (auto &[v, w] : adj[u]) {
        if (dfs_num[v] == -1) { // una arista de arbol
            dfs_parent[v] = u;
            if (u == dfsRoot) ++rootChildren; // vaso
                especial, raiz
            articulationPointAndBridge(v);
            if (dfs_low[v] >= dfs_num[u]) // para puntos
                de articulacion
                articulation_vertex[u] = 1;
            if (dfs_low[v] > dfs_num[u]) // para puentes
                printf(" (%d, %d) is a bridge\n", u, v);
            dfs_low[u] = min(dfs_low[u], dfs_low[v]); //
        }
        else if (v != dfs_parent[u]) // si es ciclo no
            trivial
            dfs_low[u] = min(dfs_low[u], dfs_num[v]); //
                entonces actualizar
    }
}
```

```
int main(){
    dfs_num.assign(V, -1); dfs_low.assign(V, 0);
    dfs_parent.assign(V, -1); articulation_vertex.assign(
        V, 0);
    dfsNumberCounter = 0;
    adj.resize(V);

    printf("Bridges:\n");
    for (int u = 0; u < V; ++u)
        if (dfs_num[u] == -1) {
            dfsRoot = u; rootChildren = 0;
            articulationPointAndBridge(u);
            articulation_vertex[dfsRoot] = (rootChildren
                > 1); // caso especial
        }
    printf("Articulation Points:\n");
    for (int u = 0; u < V; ++u)
        if (articulation_vertex[u])
            printf(" Vertex %d\n", u);
}
```

3.4 Orden Topologico

```
//Orden de un grafo estilo malla curricular de
    prerrequisitos
vector<vi> adj;
vi dfs_num;
vi ts;

void dfs(int v){
    dfs_num[v] = 1;
    for (int i = 0; i < (int) adj[v].size(); i++){

        if (dfs_num[adj[v][i]] != 1){
            dfs(adj[v][i]);
        }

    }
    ts.push_back(v);
}
//Imprimir el vector ts al reves: reverse(ts.begin(), ts.
    end());
```

3.5 Algoritmo de Khan

```
//ALgoritmo de orden topologico
//DAG: Grafo aciclico dirigido
int n, m;
vector<vi> adj;
vi grado;
vi orden;

void khan(){
    queue<int> q;

    for (int i = 1; i<=n; i++){
        if (!grado[i]) q.push(i);
    }
    int nodo;

    while(!q.empty()){
        nodo = q.front(); q.pop();
        orden.push_back(nodo);
        for (int v : adj[nodo]){
            grado[v]--;
            if (grado[v] == 0) q.push(v);
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m;
    adj.resize(n+1);
    grado.resize(n+1);

    for (int i = 0; i<m; i++){
        int x, y; cin >> x >> y;
        adj[x].push_back(y);
        grado[y]++;
    }

    khan();

    if (orden.size() == n){
        for (int i : orden) cout << i;
    }
    else{
        cout << "No DAG"; //No es un grafo aciclico
            dirigido (tiene un ciclo)
    }
}
```

3.6 Floodfill

```
//Relleno por difusion-etiquetado/coloreado de
    componentes conexos
//Recorrer matrices como grafos implicitos
//Pueden usar los vectores dirx y diry en lugar de dr y
    dc si se requiere
vector<string> grid;

int R, C, ans;

int floodfill(int r, int c, char c1, char c2){
    //Devuelve tamano de CC
    if (r < 0 || r >= R || c< 0 || c >= C) return 0;
        //fuera de la rejilla
```

```
        if (grid[r][c] != c1) return 0;    //No tiene color c1
        int ans = 1;                    //suma 1 a ans porque el vertice (r, c) tiene color c1
        grid[r][c] = c2;                //Colorea el vertice (r, c) a c2 para evitar ciclos
        for (int d = 0; d < 8; d++){
            ans += floodfill(r + dr[d], c + dc[d], c1, c2);
        }
        return ans;
    }

    int main() {
        ios::sync_with_stdio(false);
        cin.tie(0);
        cin >> R; cin >> C;
        cout << floodfill(0, 0, 'W', '.');
    }
```

3.7 Algoritmo Kosajaru

```
//Encontrar las componentes fuertemente conexas en un grafo dirigido
//Componente fuertemente conexa: es un grupo de nodos en el que hay
//un camino dirigido desde cualquier nodo hasta cualquier otro nodo dentro del grupo.
void Kosaraju(int u, int pass) {
    dfs_num[u] = 1;
    vii &neighbor = (pass == 1) ? AL[u] : AL_T[u];
    for (auto &[v, w] : neighbor)
        if (dfs_num[v] == UNVISITED)
            Kosaraju(v, pass);
    S.push_back(u);
}

int main(){
    S.clear();
    dfs_num.assign(N, UNVISITED);
    for (int u = 0; u < N; ++u)
        if (dfs_num[u] == UNVISITED)
            Kosaraju(u, 1);
    numSCC = 0;
    dfs_num.assign(N, UNVISITED);
    for (int i = N-1; i >= 0; --i)
        if (dfs_num[S[i]] == UNVISITED)
            ++numSCC, Kosaraju(S[i], 2);
    printf("There are %d SCCs\n", numSCC);
}
```

3.8 Dijkstra

```
//Camino mas cortos
//NO USAR CON PESOS NEGATIVOS, usar Bellman Ford o SPFA(mas rapido)
// O ((V+E)*log V)
vi dijkstra(vector<vii> &adj, int s, int V){
    vi dist(V+1, INT_MAX); dist[s] = 0;
    priority_queue<ii, vii, greater<ii> > pq; pq.push(ii(0, s));
    while(!pq.empty()){
        ii front = pq.top(); pq.pop();
        int d = front.first, u = front.second;
        if (d > dist[u]) continue;

        for (int j = 0; j < (int)adj[u].size(); j++){
            ii v = adj[u][j];
            if (dist[u] + v.second < dist[v.first]){
                dist[v.first] = dist[u] + v.second;
                pq.push(ii(dist[v.first], v.first));
            }
        }
    }
    return dist;
}
```

3.9 Bellman Ford

```
//Lo mismo que dijkstra pero con pesos negativos
//O(E*V)
void bellman_ford(){
    vi dist(V, INF); dist[s] = 0;
    for (int i = 0; i < V-1; ++i)
        for (int u = 0; u < V; ++u)
            if (dist[u] != INF)
            for (auto &[v, w] : adj[u])
                dist[v] = min(dist[v], dist[u]+w);
}
```

3.10 Floyd Warshall

```
//Camino minimo entre todos los pares de vertices
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int V; cin >> V;
    vector<vi> adjMat(V+1, vi(V+1));
    //Condicion previa: adjMat[i][j] contiene peso de la arista (i, j)
    //o INF si no existe esa arista
    for (int k = 0; k<V; k++)
        for (int i = 0; i<V; i++)
            for (int j = 0; j<V; j++)
```

```
                adjMat[i][j] = min(adjMat[i][j], adjMat[i
                    ][k] + adjMat[k][j]);
}
```

3.11 MST Kruskal

```
//Arbol de minima expansion
//O(E*log V)
int main() {
    int n, m;
    cin >> n >> m;
    vector<pair<int, ii>> adj; //Los pares son: {peso, {
        vertice, vecino}}

    for (int i = 0; i<m; i++){
        int x, y, w; cin >> x >> y >> w;
        adj.push_back(make_pair(w, ii(x, y)));
    }

    sort(adj.begin(), adj.end());

    int mst_costo = 0, tomados = 0;
    dsu UF(n);
    for (int i = 0; i<m && tomados < n-1; i++){
        pair<int, ii> front = adj[i];
        if (!UF.is_same_set(front.second.first, front.
            second.second)){
            tomados++;
            mst_costo += front.first;
            UF.unionSet(front.second.first, front.second.
                second);
        }
    }
    cout << mst_costo;
}
```

3.12 Shortest Path Faster Algorithm

```
//Algoritmo mas rapido de ruta minima
//O(V*E) peor caso, O(E) en promedio.
ll spfa(vector<vii>& adj, ll s, ll n) {
    vl d(n+1, INFL);
    vector<bool> inqueue(n, false);
    queue<ll> q;

    d[s] = 0;
    q.push(s);
    inqueue[s] = true;
    while (!q.empty()) {
        ll v = q.front();
        q.pop();
        inqueue[v] = false;
        for (auto edge : adj[v]) {
            ll to = edge.first;
            ll len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                if (!inqueue[to]) {
                    q.push(to);
                    inqueue[to] = true;
                }
            }
        }
    }
    return d[n];
}
```

4 Matematicas

4.1 Criba de Eratostenes

```
// O(N log log N)
ll _sieve_size;
bitset<10000010> bs;     //10^7 es el limite aprox
vl p;                    //Lista compacta de primos
void sieve(ll upperbound) {          //Rango = [0..limite]
    _sieve_size = upperbound+1;      //Para incluir al
        limite
    bs.set();                        //Todo unos
    bs[0] = bs[1] = 0;               //0 y 1 (no son
        primos)
    for (ll i = 2; i < _sieve_size; ++i) if (bs[i]) {
        for (ll j = i*i; j < _sieve_size; j += i) bs[j] =
            0;
        p.push_back(i);              //Anadir primo i a la
            lista
    }
}
```

4.2 Descomposicion en primos (y mas cosas)

```
ll _sieve_size;
bitset<10000010> bs;
vl p;
void sieve(ll upperbound) {
    _sieve_size = upperbound+1;
    bs.set();
    bs[0] = bs[1] = 0;
    for (ll i = 2; i < _sieve_size; ++i) if (bs[i]) {
        for (ll j = i*i; j < _sieve_size; j += i) bs[j] =
            0;
        p.push_back(i);
```

```
    }
}
// O( sqrt(N) / log(sqrt(N)) )
vl primeFactors(ll N) {
    vl factors;
    for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i)
        while (N%p[i] == 0) {        //Hallado un primo para N
            N /= p[i];               //Eliminarlo de N
            factors.push_back(p[i]);
        }
    if (N != 1) factors.push_back(N);//El N restante es primo
    return factors;
}
int main(){
    sieve(10000000);
}

//Variantes del algoritmo

//Contar el numero de divisores de N
int numDiv(ll N) {
    int ans = 1;    //Empezar con ans = 1
    for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
        int power = 0;  //Contar la potencia
        while (N%p[i] == 0) { N /= p[i]; ++power; }
        ans *= power+1; //Seguir la formula
    }
    return (N != 1) ? 2*ans : ans;  //Ultimo factor = N^1
}

//Suma de los divisores de N
//N = a^i * b^i * ... * c^k => N = (a^(i+1) - 1) / (a-1) + ...
ll sumDiv(ll N) {
    ll ans = 1;          // empezar con ans = 1
    for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
        ll multiplier = p[i], total = 1;
        while (N%p[i] == 0) {
            N /= p[i];
            total += multiplier;
            multiplier *= p[i];
        }                                // total para
        ans *= total;                    // este factor primo
    }
    if (N != 1) ans *= (N+1);    // N^2-1/N-1 = N+1
    return ans;
}
```

4.3 Prueba de primalidad

```
ll _sieve_size;
bitset<10000010> bs;
vl p;
void sieve(ll upperbound) {
    _sieve_size = upperbound+1;
    bs.set();
    bs[0] = bs[1] = 0;
    for (ll i = 2; i < _sieve_size; ++i) if (bs[i]) {
        for (ll j = i*i; j < _sieve_size; j += i) bs[j] = 0;
        p.push_back(i);
    }
}
bool isPrime(ll N) {
    if (N < _sieve_size) return bs[N]; // O(1) primos pequenos
    for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
        if (N%p[i] == 0)
            return false;
    return true;        // O ( sqrt(N) / log(sqrt(N)) ) para N > 10^7
}   //Nata: solo se garantiza para N <= (ultimo primo de p)^2 = 9.99 * 10^13
```

4.4 Criba Modificada

```
//Criba modificada
/*
Si hay que determinar el numero de factores primos para muchos (o un rango) de enteros.
La mejor solucion es el algoritmo de criba modificada O(N log log N)
*/
int numDiffPFarr[MAX_N+10] = {0}; // e.g., MAX_N = 10^7
for (int i = 2; i <= MAX_N; ++i)
    if (numDiffPFarr[i] == 0) // i is a prime number
        for (int j = i; j <= MAX_N; j += i)
            ++numDiffPFarr[j]; // j is a multiple of i

//Similar para EulerPhi
int EulerPhi[MAX_N+10];
for (int i = 1; i <= MAX_N; ++i) EulerPhi[i] = i;
for (int i = 2; i <= MAX_N; ++i)
    if (EulerPhi[i] == i) // i is a prime number
        for (int j = i; j <= MAX_N; j += i)
            EulerPhi[j] = (EulerPhi[j]/i) * (i-1);
```

4.5 Funcion Totient de Euler

```
//EulerPhi(N): contar el numero de enteros positivos < N
    que son primos relativos a N.
//El vector p es el que genera la criba de eratostenes
//Phi(N) = N * productoria(1 - (1/pi))
ll EulerPhi(ll N) {
    ll ans = N; // Empezar con ans = N
    for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <=
        N); ++i) {
        if (N%p[i] == 0) ans -= ans/p[i]; //contar
            factores
        while (N%p[i] == 0) N /= p[i];    //primos unicos
    }
    if (N != 1) ans -= ans/N; // ultimo factor
    return ans;
}
```

4.6 Exponenciacion binaria

```
ll binpow(ll b, ll n, ll m) {
    b %= m;
    ll res = 1;
    while (n > 0) {
        if (n & 1)
            res = res * b % m;
        b = b * b % m;
        n >>= 1;
    }
    return res % m;
}
```

4.7 Fibonacci Matriz

```
def mult(matriz1, matriz2):
    fila_1 = [matriz1[0][0] * matriz2[0][0] + matriz1
        [0][1] * matriz2[1][0], matriz1[0][0] * matriz2
        [0][1] + matriz1[0][1] * matriz2[1][1]]
    fila_2 = [matriz1[1][0] * matriz2[0][0] + matriz1
        [1][1] * matriz2[1][0], matriz1[1][0] * matriz2
        [0][1] + matriz1[1][1] * matriz2[1][1]]
    return [fila_1, fila_2]
def mult_vector(matriz, vector):
    a = matriz[0][0] * vector[0] + matriz[0][1] * vector
        [1]
    b = matriz[1][0] * vector[0] + matriz[1][1] * vector
        [1]
    return [a, b]
def modulos(matriz, n):
    matriz[0][0] %= n
    matriz[0][1] %= n
    matriz[1][0] %= n
    matriz[1][1] %= n
    return matriz
def exp_bin(b, n, m):
    res = [[1, 0], [0, 1]]
    while n > 0:
        if n % 2 == 1:
            res = mult(modulos(res, m), modulos(b, m))
        b = mult(modulos(b, m), modulos(b, m))
        n //= 2
    return modulos(res, m)
matriz = [[1, 1] , [1, 0]]
vector = [1, 0]
# a = list(map(int, input().split()))
m = exp_bin(matriz, int(input()), 1000000007)
v = mult_vector(m, vector)
print(v[1] % 1000000007)
```

4.8 GCD y LCM

```
//O(log10 n) n == max(a, b)
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b);
    }
int lcm(int a, int b) { return a / gcd(a, b) * b; }
//gcd(a, b, c) = gcd(a, gcd(b, c))
```

4.9 Algoritmo Euclideo Extendido

```
ll extEuclid(ll a, ll b, ll &x, ll &y){
    ll xx = y = 0;
    ll yy = x = 1;
    while (b){
        ll q = a/b;
        ll t = b; b = a%b; a = t;
        t = xx; xx = x-q*xx; x = t;
        t = yy; yy = y-q*yy; y = t;
    }
    return a;   //Devuelve gcd(a, b)
}
```

4.10 Inverso modular

```
ll mod(ll a, ll m){
    return ((a%m) + m) % m;
}

ll modInverse(ll b, ll m){
    ll x, y;
    ll d = extEuclid(b, m, x, y);  //obtiene b*x + m*y ==
        d
    if (d != 1) return -1;          //indica error
    // b*x + m*y == 1, ahora aplicamos (mod m) para
        obtener b*x == 1 (mod m)
```

```
    return mod(x, m);
}
```

4.11 Coeficientes binomiales

```
const int MAX_N = 100010;

ll inv (ll a){
    return binpow(a, MOD-2, MOD);
}

ll fact[MAX_N];
ll C(int n, int k){
    if (n < k) return 0;
    return (((fact[n] * inv(fact[k])) % MOD) * inv(fact[n
        -k])) % MOD;
}

int main() {
    fact[0] = 1;
    for (int i = 1; i<MAX_N; i++){
        fact[i] = (fact[i-1]*i) % MOD;
    }
    cout << C(100000, 50000) << "\n";

    return 0;
}
```

5 Geometria

5.1 Puntos

```
// Punto entero
struct point{
    ll x,y;
    point(ll x,ll y): x(x),y(y){}
};

// Punto flotante
struct point{
    double x,y;
    point(double _x,double _y): x(_x),y(_y){}
    bool operator == (point other) const{
        return (fabs(x-other.x)<EPS) && (fabs(y-other.y)<
            EPS);
    };
};

// Distancia entre dos puntos
double dist(point p1, point p2){
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y
        -p2.y));
}

// Rotacion de un punto
point rotate(point p, double theta){
    // rotar por theta grados respecto al origen (0,0
    double rad = theta*(M_PI/180);
    return point(p.x*cos(rad)-p.y*sin(rad),p.x*sin(rad)+p
        .y*cos(rad));
}
```

5.2 Lineas

```
// Linea de flotantes de la forma ax+by+c=0
struct line{double a,b,c;};

// Creacion de linea con dos puntos
// b=1 para lineas no verticales y b =0 para verticales
void pointsToLine(point p1,point p2,line& l){
    if (fabs(p1.x-p2.x)<EPS){
        l.a=1.0; l.b=0.0; l.c=-p1.x;
    }else{
        l.a= -double(p1.y-p2.y)/(p1.x-p2.x);
        l.b= 1.0;
        l.c= -double(l.a*p1.x)-p1.y;
    }
}

// Comprobacion de lineas paralelas
bool areParallel(line l1,line l2){
    return (fabs(l1.a-l2.a)<EPS) && (fabs(l1.b-l2.b)<EPS)
        ;
}

// Comprobacion de lineas iguales
bool areSame(line l1,line l2){
    return areParallel(l1,l2) && (fabs(l1.c-l2.c)<EPS);
}

// Disntacia de un punto a una linea
double distPointToLineaEq(line l, point p){
    return fabs(l.a*p.x + l.b*p.y + l.c)/sqrt(l.a*l.a+l.b
        *l.b);
}

bool areIntersect(line l1, line l2, point& p){
    if (areParallel(l1,l2)) return false;
    // resolver sistema 2x2
    p.x = (l2.b*l1.c - l1.b*l2.c)/(l2.a*l1.b - l1.a*l2.b)
        ;

    // CS: comprobar linea vertical -> div por cero
    if (fabs(l1.b)>EPS) p.y = -(l1.a*p.x + l1.c);
    else p.y = -(l2.a*p.x + l2.c);

    return true;
}
```

5.3 Vectores

```
// Creacion de un vector
struct vec{
    double x,y;
    vec(double x,double y): x(x),y(y){}
};

// Puntos a vector
vec toVec(point a,point b){
    return vec(b.x-a.x , b.y-a.y);
}

// Escalar un vector
vec scale(vec v, double s){
    // s no negatico:
    // <1 mas corto
    // 1 igual
    // >1 mas largo
    return vec(v.x*s,v.y*s);
}

// Trasladar p segun v
point traslate(point p, vec v){
    return point(p.x+v.x , p.y+v.y);
}

// Producto Punto
double dot(vec a,vec b){
    return (a.x*b.x + a.y*b.y);
}

// Cuadrado de la norma
double norm_sq(vec v){
    return v.x*v.x + v.y*v.y;
}

// Angulo formado por aob
double angle(point a, point o, point b){
    vec oa = toVec(o,a);
    vec ob = toVec(o,b);
    return acos(dot(oa,ob)/sqrt(norm_sq(oa)*norm_sq(ob)))
        ;
}

// Producto cruz
double cross(vec a, vec b){
    return (a.x*b.y)-(a.y*b.x);
}

// Lado respecto una linea pq
bool ccw(point p,point q,point r){
    // Devuelve verdadero si el punto r esta a la
        izquierda de la linea pq
    return cross(toVec(p,q),toVec(p,r))>0;
}

// Colinear
bool collinear(point p, point q, point r){
    return fabs(cross(toVec(p,q), toVec(p,r)))<EPS;
}
```

5.4 Poligonos

```
// Crear un poligono
// la idea es crearlo con algun orden ya sea horario o
    anti-horario
// y debe cerrarse
vector<point> Poligono;

// Perimetro de un poligono
double perimeter(const vector<point>& P){
    double result =0.0;
    for (int i =0;i<(int)P.size()-1;i++)result+= dist(P[i
        ],P[i+1]);
    return result;
}

// Area de un poligono
double area(const vector<point>& P){
    // la mitad del determinante
    double result = 0.0, x1,y1,x2,y2;
    for (int i =0;i<(int)P.size()-1;i++){
        x1 = P[i].x;
        x2 = P[i+1].x;
        y1 = P[i].y;
        y2 = P[i+1].y;
        result += (x1*y2 - x2*y1);
    }
    return fabs(result/2.0);
}

// Comprobacion de si es Convexto un poligono
bool isConvex(const vector<point>& P){
    int sz = (int)P.size();
    if (sz<=3) return false;
    bool isLeft = ccw(P[0],P[1],P[2]);
    for (int i =1;i<sz-1;i++)
        if (ccw(P[i],P[i+1],P[(i+2)==sz ? 1:i+2])!=isLeft
            )
            return false;
    return true;
}

// Comprobar si un punto esta dentro de un poligono
bool inPoligono(point pt, const vector<point>& P){
    // P puede ser concavo/convexo
    if ((int)P.size()==0) return false;
    double sum =0;
    for (int i =0;i<(int)P.size()-1;i++){
        if (ccw(pt,P[i],P[i+1]))
            sum += angle(P[i],pt,P[i+1]); // izquierda/
                anti-horario
```

```
        else sum -= angle(P[i],pt,P[i+1]);// derecha/horario
    }
    return fabs(fabs(sum)-2*M_PI)<EPS;
}
```

5.5 Convex Hull

```
struct pt{
    double x,y;
    pt(double x,double y): x(x),y(y){}
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // horario
    if (v > 0) return +1; // anti-horario
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    a = st;
}

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

    ll n; cin>>n;

    vector<pt> Puntos;
    for (int i =0;i<n;i++){
        double x,y;cin>>x>>y;
        pt punto(x,y);
        Puntos.push_back(punto);
    }
    convex_hull(Puntos,true);
    cout<<Puntos.size()<<ln;
    for (pt punto:Puntos){
        cout<<(ll)punto.x<<" "<<(ll)punto.y<<ln;
    }
}
```