

Notebook UNosnovatos

Contents

1 C++	1
1.1 C++ plantilla	1
1.2 Librerias	2
1.3 Bitmask	2
2 Estructuras de Datos	3
2.1 Disjoint Set Union	3
2.2 Fenwick Tree	3
2.3 Segment Tree	3
3 Programacion dinamica	4
3.1 LIS	4
3.2 Knapsack	4
3.3 Cambio de monedas	5
3.4 Algoritmo de Kadane 2D	5
4 Grafos	5
4.1 DFS	5
4.2 BFS	5
4.3 Puntos de articulacion y puentes	5
4.4 Orden Topologico	6
4.5 Algoritmo de Khan	6
4.6 Floodfill	7
4.7 Algoritmo Kosajaru	7
4.8 Dijkstra	7
4.9 Bellman Ford	8
4.10 Floyd Warshall	8
4.11 MST Kruskal	8
4.12 MST Prim	8
4.13 Shortest Path Faster Algorithm	9
4.14 Camino mas corto de longitud fija	9
5 Matematicas	9
5.1 Criba de Eratostenes	9
5.2 Descomposicion en primos (y mas cosas)	9
5.3 Prueba de primalidad	10
5.4 Criba Modificada	10
5.5 Funcion Totient de Euler	11
5.6 Exponenciacion binaria	11

5.7 Exponenciacion matricial	11
5.8 Fibonacci Matriz	11
5.9 GCD y LCM	11
5.10 Algoritmo Euclideo Extendido	11
5.11 Inverso modular	12
5.12 Coeficientes binomiales	12
6 Geometria	12
6.1 Puntos	12
6.2 Lineas	12
6.3 Vectores	13
6.4 Poligonos	13
6.5 Convex Hull	14
7 Teoría y miscelánea	15
7.1 Sumatorias	15
7.2 Teoría de Grafos	15
7.2.1 Teorema de Euler	15
7.2.2 Planaridad de Grafos	15
7.3 Teoría de Números	15
7.3.1 Ecuaciones Diofánticas Lineales	15
7.3.2 Pequeño Teorema de Fermat	15
7.3.3 Teorema de Euler	15
7.4 Teorema de Pick	15

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;
}

```

1.2 Librerías

```

// En caso de que no sirva #include <bits/stdc++.h>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <sstream>
#include <fstream>
#include <cassert>
#include <climits>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <deque>
#include <stack>
#include <list>
#include <map>
#include <set>
#include <bitset>
#include <iomanip>
#include <unordered_map>
////
#include <tuple>
#include <random>
#include <chrono>

```

1.3 Bitmask

```

// Todas son O(1) Representacion
int a = 5; // Representacion binaria: 0101
int b = 3; // Representacion binaria: 0011
// Operaciones Principales
int resultado_and = a & b; // 0001 (1 en decimal)
int resultado_or = a | b; // 0111 (7 en decimal)
int resultado_xor = a ^ b; // 0110 (6 en decimal)

int num = 42; // Representacion binaria: 00101010
bitset<8> bits(num); // Crear un objeto bitset a partir
del numero
cout << "Secuencia de bits: " << bits << "\n";

```

```

bits.count(); // Cantidad de bits activados
bits.set(3, true); // Establecer el cuarto bit en 1
bits.reset(6); // Establecer el septimo bit en 0

ll S, T;
// Operaciones con bits (/*) por 2 (redondea de forma
automatica)
S=34; // == 100010
S = S<<1; // == S*2 == 68 == 1000100
S = S>>2; // == S/4 == 17 == 10001
S = S>>1; // == S/2 == 8 == 1000

// Encender un bit
S = 34;
S = S | (1<<3); // S = 42 (101010)

// Limpiar o apagar un bit
// ~: Not operacion
S = 42;
S &= ~(1<<1); // S = 40 (101000)

// Comprobar si un bit esta encendido
S = 42;
T = S & (1<<3); // (!= 0): el tercer bit esta encendido

// Invertir el estado de un bit
S = 40;
S ^= (1<<2); // 44 (101100)

// LSB (Primero de la derecha)
S = 40;
T = ((S) & -(S)); // 8 (001000)
__builtin_ctz(T); // nos entrega el indice del LSB

// Encender todos los bits
ll n = 3; // el tamaño del set de bits
S = 0;
S = (1<<n) - 1; // 7 (111)

// Enumerar todos los posibles subsets de un bitmask
int mask = 18;
for (int subset = mask; subset; subset = (mask & (subset
-1))) {
    cout << subset << "\n";
}

// otras funciones de c++
__builtin_popcount(32); // 100000 (base 2), only 1 bit is
on
__builtin_popcount(30); // 11110 (base 2), 4 bits are on
__builtin_popcountll((1ll<<62)-1ll); // 2^62-1 has 62 bits
on (near limit)
__builtin_ctz(32); // 100000 (base 2), 5 trailing zeroes
__builtin_ctz(30); // 11110 (base 2), 1 trailing zero
__builtin_ctzll(1ll<<62); // 2^62 has 62 trailing zeroes

```

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, lazyl;

    nodeST(vi &v, int l, int r) : l(l), r(r){
        int m = (l+r)>>1;
        lazy = 0;
        lazyl = 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(lazyl){
            value = lazyl * (r-l+1);
            if (l != r){
                left->lazyl = lazyl, right->lazyl = lazyl;
                left->lazy = 0, right->lazy = 0;
            }
            lazyl = 0;
            lazy = 0;
        }
        else{
            value += lazy * (r-l+1);
            if (l != r){
                if(left->lazyl) left->lazyl += lazy;
                else left->lazy += lazy;
                if(right->lazyl) right->lazyl += 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 Programacion dinamica

3.1 LIS

```

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

```

```

    int n; cin >> n;
    vl vals(n);
    for (int i = 0; i < n; i++) cin >> vals[i];

    vl copia(vals);
    sort(copia.begin(), copia.end());

    map<ll, ll> dicc;
    for (int i=0; i<n; i++) if (!dicc.count(copia[i])) dicc[
        copia[i]] = i;

    vl baseSt(n, 0);
    nodeSt st(baseSt, 0, n - 1);
    ll maxi = 0;
    for (ll pVal:vals) {
        ll op = st.get(0, dicc[pVal]-1)+1;
        maxi = max(maxi, op);
        st.act1(dicc[pVal], op);
    }
    cout<<maxi<<ln;
}

```

3.2 Knapsack

```

int main() {
    int n, w; cin >> n >> w;
    // w es la capacidad de la mochila
    // n es la cantidad de elementos
    vi pesos;
    vi valor;
    for (int i = 0; i < n; i++) {
        int p, v; cin >> p >> v;
        pesos.push_back(p);
        valor.push_back(v);
    }

    ll dp[n+1][w+1] = {0};

    for (int i = 0; i <= n; i++) dp[i][0] = 0;
    for (int i = 0; i <= w; i++) dp[0][i] = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= w; j++) {
            ll op1 = dp[i-1][j];
            ll op2;
            if (j < pesos[i-1]) op2 = 0;
            else op2 = valor[i-1] + dp[i-1][j - pesos[i-1]];
            dp[i][j] = max(op1, op2);
        }
    }

    ll res = dp[n][w];
    cout<<res;
}

```

3.3 Cambio de monedas

```
int main() {
    int inf = 99999999;
    int n, x; cin >> n >> x;
    // n: numero de monedas x: la cantidad buscada
    vi coins(n); // valor de cada moneda
    for (int i=0; i<n; i++) cin >> coins[i];
    vector<vi> dp(n+1, vi(x+1, 0));

    for (int i=0; i<=x; i++) dp[0][i] = inf;
    for (int i=1; i<=n; i++) {
        for (int j=1; j<=x; j++) {
            if (j < coins[i-1]) dp[i][j] = dp[i-1][j];
            else dp[i][j] = min(1+dp[i][j-coins[i-1]], dp[i-1][j]);
        }
    }

    int res = dp[n][x];
    cout << (res == inf ? -1 : res) << endl;
}
```

3.4 Algoritmo de Kadane 2D

```
int main() {
    ll fil, col; cin >> fil >> col;
    vector<vl> grid(fil, vl(col, 0));

    // Algoritmo de Kadane/DP para suma maxima de una matriz
    // 2D en  $O(n^3)$ 
    for (int i=0; i<fil; i++) {
        for (int e=0; e<col; e++) {
            ll num; cin >> num;
            if (e > 0) grid[i][e] = num + grid[i][e-1];
            else grid[i][e] = num;
        }
    }

    ll maxGlobal = -LONG_LONG_MAX;
    for (int l=0; l<col; l++) {
        for (int r=1; r<col; r++) {
            ll maxLoc = 0;
            for (int row=0; row<fil; row++) {
                if (l > 0) maxLoc += grid[row][r] - grid[row][l-1];
                else maxLoc += grid[row][r];
                if (maxLoc < 0) maxLoc = 0;
                maxGlobal = max(maxGlobal, maxLoc);
            }
        }
    }
}
```

4 Grafos

4.1 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]);
        }
    }
}
```

4.2 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
        }
    }
}
```

4.3 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);
}

```

4.4 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());

```

4.5 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)
    }
}

```

4.6 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', '.');
}

```

4.7 Algoritmo Kosaraju

```

//Encontrar las componentes fuertemente conexas en un
//grafo dirigido

```

```

//Componente fuertemente conexas: 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);
}

```

4.8 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;
}

```

4.9 Bellman Ford

```

vi bellman_ford(vector<vii> &adj, int s, int n){
    vi dist(n, INF); dist[s] = 0;
    for (int i = 0; i<n-1; i++){
        bool modified = false;
        for (int u = 0; u<n; u++){
            if (dist[u] != INF)
                for (auto &[v, w] : adj[u]){
                    if (dist[v] <= dist[u] + w) continue;
                    dist[v] = dist[u] + w;
                    modified = true;
                }
            if (!modified) break;
        }
        bool negativeCicle = false;
        for (int u = 0; u<n; u++){
            if (dist[u] != INF)
                for (auto &[v, w] : adj[u]){
                    if (dist[v] > dist[u] + w) negativeCicle = true;
                }
        }
        return dist;
    }
}

```

4.10 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]);
}

```

4.11 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;
}

```

4.12 MST Prim

```

vector<vii> adj;
vi tomado;
priority_queue<ii> pq;
void process(int u){
    tomado[u] = 1;
    for (auto &[v, w] : adj[u]){
        if (!tomado[v]) pq.emplace(-w, -v);
    }
}
int prim(int v, int n){
    tomado.assign(n, 0);
    process(0);
    int mst_costo = 0, tomados = 0;
    while (!pq.empty()){
        auto [w, u] = pq.top(); pq.pop();
        w = -w; u = -u;
        if (tomado[u]) continue;
        mst_costo += w;
        process(u);
        tomados++;
        if (tomados == n-1) break;
    }
    return mst_costo;
}

```


4.13 Shortest Path Faster Algorithm

```
//Algoritmo mas rapido de ruta minima
//O(V*E) peor caso, O(E) en promedio.
bool spfa(vector<vii> &adj, vector<int> &d, int s, int n)
{
    d.assign(n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inqueue(n, false);
    queue<int> q;

    d[s] = 0;
    q.push(s);
    inqueue[s] = true;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        inqueue[v] = false;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

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

4.14 Camino mas corto de longitud fija

```
/*
Modificar operacion * de matrix de esta forma:
En la exponenciacion binaria inicializar matrix ans = b
*/
matrix operator * (const matrix &b){
    matrix ans(this->r, b.c, vector<vl>(this->r, vl(b.c,
        INFL)));

    for (int i = 0; i < this->r; i++) {
        for (int k = 0; k < b.r; k++) {
            for (int j = 0; j < b.c; j++) {
                ans.m[i][j] = min(ans.m[i][j], m[i][k] +
                    b.m[k][j]);
            }
        }
    }
}
```

```
    }
    return ans;
}

int main() {
    int n, m, k; cin >> n >> m >> k;
    vector<vl> adj(n, vl(n, INFL));

    for (int i = 0; i < m; i++) {
        ll a, b, c; cin >> a >> b >> c; a--; b--;
        adj[a][b] = min(adj[a][b], c);
    }

    matrix graph(n, n, adj);
    graph = pow(graph, k-1);

    cout << (graph.m[0][n-1] == INFL ? -1 : graph.m[0][n-1]) << "\n";

    return 0;
}
```

5 Matematicas

5.1 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
    }
}
```

5.2 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(100000000);
}

//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];
        }
        ans *= total; // total para
        // este
        factor primo
    }
}

```

```

}
if (N != 1) ans *= (N+1); // N^2-1/N-1 = N+1
return ans;
}

```

5.3 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
} //Nota: solo se garantiza para N <= (ultimo primo de p)^2 = 9.99 * 10^13

```

5.4 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);

```

5.5 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;
}

```

5.6 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;
}

```

5.7 Exponenciacion matricial

```

struct matrix {
    int r, c; vector<vl> m;
    matrix(int r, int c, const vector<vl> &m) : r(r), c(c), m(m) {}

    matrix operator * (const matrix &b) {
        matrix ans(this->r, b.c, vector<vl>(this->r, vl(b.c, 0)));
        for (int i = 0; i < this->r; i++) {
            for (int k = 0; k < b.c; k++) {
                if (m[i][k] == 0) continue;
                for (int j = 0; j < b.c; j++) {

```

```

                    ans.m[i][j] += mod(m[i][k], MOD) *
                        mod(b.m[k][j], MOD);
                    ans.m[i][j] = mod(ans.m[i][j], MOD);
                }
            }
        }
        return ans;
    }
};

matrix pow(matrix &b, ll p) {
    matrix ans(b.r, b.c, vector<vl>(b.r, vl(b.c, 0)));
    for (int i = 0; i < b.r; i++) ans.m[i][i] = 1;
    while (p) {
        if (p&1) {
            ans = ans*b;
        }
        b = b*b;
        p >>= 1;
    }
    return ans;
}

```

5.8 Fibonacci Matriz

```

/*
[1 1] p   [fib(p+1) fib(p)]
[1 0]     [fib(p)   fib(p-1)]
*/
vector<vl> matriz = {{1, 1}, {1, 0}};
matrix m(2, 2, matriz);

ll n; cin >> n;

cout << pow(m, n).m[0][1] << "\n";

```

5.9 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))

```

5.10 Algoritmo Euclideo Extendido

```

// O(log(min(a, b)))
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)
}

```

5.11 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);
}

// Otra forma
// O(log MOD)
ll inv (ll a){
    return binpow(a, MOD-2, MOD);
}

```

5.12 Coeficientes binomiales

```

const int MAX_N = 100010;    // MOD > MAX_N
// O (log MOD)
ll inv (ll a){
    return binpow(a, MOD-2, MOD);
}

ll fact[MAX_N];
// O(log MOD)
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;
}

```

6 Geometria

6.1 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));
}

```

6.2 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);
}

// Distancia de un punto a una linea
double distPointToLineEq(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;
}

```

6.3 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 negativo:
    // <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;
}

```

6.4 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 Convexo 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;
}

```

6.5 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;
    }
}

```

7 Teoría y miscelánea

7.1 Sumatorias

- $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=1}^n i^3 = \left(\frac{n(n+1)}{2}\right)^2$
- $\sum_{i=1}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$
- $\sum_{i=1}^n i^5 = \frac{(n(n+1))^2(2n^2+2n-1)}{12}$
- $\sum_{i=0}^n x^i = \frac{x^{n+1}-1}{x-1}$ para $x \neq 1$

7.2 Teoría de Grafos

7.2.1 Teorema de Euler

En un grafo conectado planar, se cumple que $V - E + F = 2$, donde V es el número de vértices, E es el número de aristas y F es el número de caras.

7.2.2 Planaridad de Grafos

Un grafo es planar si y solo si no contiene un subgrafo homeomorfo a K_5 (grafo completo con 5 vértices) ni a $K_{3,3}$ (grafo bipartito completo con 3 vértices en cada conjunto).

7.3 Teoría de Números

7.3.1 Ecuaciones Diofánticas Lineales

Una ecuación diofántica lineal es una ecuación en la que se buscan soluciones enteras x e y que satisfagan la relación lineal $ax + by = c$, donde a , b y c son constantes dadas.

Para encontrar soluciones enteras positivas en una ecuación diofántica lineal, podemos seguir el siguiente proceso:

1. Encontrar una solución particular: Encuentra una solución particular (x_0, y_0) de la ecuación. Esto puede hacerse utilizando el algoritmo de Euclides extendido.

2. Encontrar la solución general: Una vez que tengas una solución particular, puedes obtener la solución general utilizando la fórmula:

$$x = x_0 + \frac{b}{\text{mcd}(a, b)} \cdot t$$

$$y = y_0 - \frac{a}{\text{mcd}(a, b)} \cdot t$$

donde t es un parámetro entero.

3. Restringir a soluciones positivas: Si deseas soluciones positivas, asegúrate de que las soluciones generales satisfagan $x \geq 0$ y $y \geq 0$. Puedes ajustar el valor de t para cumplir con estas restricciones.

7.3.2 Pequeño Teorema de Fermat

Si p es un número primo y a es un entero no divisible por p , entonces $a^{p-1} \equiv 1 \pmod{p}$.

7.3.3 Teorema de Euler

Para cualquier número entero positivo n y un entero a coprimo con n , se cumple que $a^{\phi(n)} \equiv 1 \pmod{n}$, donde $\phi(n)$ es la función phi de Euler, que representa la cantidad de enteros positivos menores que n y coprimos con n .

7.4 Teorema de Pick

Sea un polígono simple cuyos vertices tienen coordenadas enteras. Si B es el número de puntos enteros en el borde, I el número de puntos enteros en el interior del polígono, entonces el área A del polígono se puede calcular con la fórmula:

$$A = I + \frac{B}{2} - 1$$