Apunte para Programación Competitiva

🤓☝️

Suplentes Sin Gloria

2023

Contents

1 Introducción, la mistica es primordial

¡Oh, mística enigmática, joya oculta del ser,
Elevas nuestras almas, nos haces renacer.
En la oscuridad profunda, eres luz y guía,
En cada paso incierto, en cada nuevo día.

Tu esencia trasciende lo que los ojos ven,
En lo más profundo de nosotros, floreces también.
Eres el susurro suave del viento en la noche,
La chispa de inspiración en cada pensamiento derroche.

Eres la llave dorada a lo etéreo y divino,
El eco de antiguos sabios, el secreto clandestino.
En el silencio de la meditación te encontramos,
En la contemplación profunda, en la que nos sumamos.

Eres el lazo que une lo terrenal y lo celestial,
El puente que nos lleva a un plano espiritual.
Nos conectas con la esencia de la existencia,
Despiertas en nosotros una profunda consciencia.

En la búsqueda de significado y propósito en la vida,
Eres el faro que nos guía, la respuesta más querida.
En la maraña de la incertidumbre y el caos,
Eres la brújula interna que nos muestra el camino audaz.

Mística, eres la chispa de creatividad sin fin,
La musa que nos inspira a crear, a dejar una huella en el confín.
Eres el fuego sagrado que arde en nuestro interior,
Nos impulsas a explorar, a descubrir nuestro mejor labor.

En la quietud de la noche, en la profundidad de la mente,
En la búsqueda de lo trascendental, en cada instante.
La mística es el alma que vive en cada uno de nosotros,
Una llama eterna que enciende sueños poderosos.

Así, en tu honor, cantemos con gratitud y devoción,
Oh, mística eterna, fuente de inspiración.
En tu abrazo encontramos significado y razón,
Eres la esencia de la vida, nuestra eterna canción.

La mística prevalecerá. 😎

2 Optimización y comentarios pre-competitivos

2.1 Librerías

Comenzaremos importando algunas librerías, aquí va un breve apunte de librerías importantes a considerar en la programación competitiva.

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <set>
#include <map>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
#include <string>

using namespace std;
```

Sin embargo, existe una opción mas fácil que incluye todas estas librerías, esta es **#include <bits/stdc++.h>**, sin embargo dejamos las anteriores para el hipotético caso en donde la ultima librería, no funcione de forma correcta (no sea portable). Obviamente en ambos casos usamos **using namespace std;**, para facilitar la velocidad y simpleza del código.

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

2.2 Optimización I/O

Llamamos $I/0$ a las operaciones de lectura y escritura de datos (*input/output*). Se puede mejorar el rendimiento de las operaciones de entrada y salida, en este sentido, se van a realizar comandos tales que evitan la sincronización automática entre las bibliotecas de C++ y las bibliotecas de C para el manejo de entrada y salida, mejorando así su rendimiento. Para esto hacemos:

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

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    return 0;
}
```

Aunque en la mayoría de casos los comandos **cin**, **cout** son lo sufientemente rápidos como para lograr no exceder el tiempo limite, existen casos donde podría pasar lo contrario. Es por esto que existen comandos mas rápidos para la lectura y escritura de datos, veamos entonces los comandos **scanf** y **printf**. Para esto primero debemos conocer unos especificadores de formato:

- $\%d$: INT
- $\%lld$: LONG LONG
- $\%f$: FLOAT
- $\%c$: CHAR
- $\%s$: STRING

Entonces, para leer e imprimir datos utilizamos los comando de la siguiente manera:

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

int main() {
    int num;
    scanf("%d", &num1);
    printf("%d", num1);

    long long num2;
    scanf("%lld", &num2);
    printf("%lld", num2);
    printf("%lld\n", num2); //Para imprimir y saltear linea

}
```

La idea es que dependiendo del tipo de datos que se quiera leer o escribir se cambia el especificador de formato correspondiente.

2.3 Optimizaciones extras

Algunas optimizaciones extras podrían ser:

```
typedef long long tint;
#define forn(n) for(tint i = 0; i < n; i++)
#define forn_back(n) for(tint i = n-1; i >=0; i--)
```

De esta manera, concluimos con un template básico para cualquier programa:

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

using namespace std;
typedef long long tint;
```

```
#define forn(n) for(tint i = 0; i < n; i++)
#define forn_back(n) for(tint i = n-1; i >=0; i--)

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    return 0;
}
```

3 Algoritmos varios

3.1 FFT (Fast Fourier Transform)

Complejidad : $O(n\ log(n))$

```
using cd = complex<double>;
const double PI = acos(-1);
using u64 = uint64_t;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1;
    base %= mod;
    while (e) {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }
    return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
}

void fft(vector<cd> & a, bool invert) {
```

```
    int n = a.size();
    if (n == 1)
        return;

    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[2*i];
        a1[i] = a[2*i+1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * PI / n * (invert ? -1 : 1);
    cd w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n/2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n/2] /= 2;
        }
        w *= wn;
    }
}

vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
```

3.2 Knapsack

Complejidad : $O(n * W)$, donde n es la cantidad de objetos y W la capacidad de la mochila

```
vector<vector<tint>> dp;
vector<pair<tint,tint>> items;

tint knapsack(tint capacity,tint n) {
    if (n == 0 || capacity == 0) {
        return 0;
    }

    if (dp[n][capacity] != -1) {
        return dp[n][capacity];
    }

    if (items[n-1].first <= capacity) {
        return dp[n][capacity] = max(
        items[n-1].second + knapsack(capacity-items[n-1].first, n-1),
        knapsack(capacity, n-1));
    } else {
        return dp[n][capacity] = knapsack(capacity, n-1);
    }
}

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

dp.assign(n + 1, vector<tint>(capacity + 1, -1));
tint maxValue = knapsack(capacity, n);
}
```

3.3 Coin Combinations

Complejidad : $O(n * K)$, donde n es la cantidad de monedas y K es el valor objetivo a sumar

```
vector<tint> coins;
vector<vector<tint>> dp;

tint getNumberOfWays(tint N, tint index) {
    if (N == 0) {
        return 1;
    }

    if (N < 0 || index >= coins.size()) {
        return 0;
```

```
    }

    if (dp[index][N] != -1) {
        return dp[index][N];
    }


    tint ways = getNumberOfWays(N - coins[index], index) + getNumberOfWays(N,
        index + 1);

    return dp[index][N] = ways;
}
int main(){
cin >> ....;

dp.assign(coins.size(), vector<tint>(N + 1, -1));
tint ways = getNumberOfWays(N, 0);
}
```

3.4 Longest Increasing Subsequence

Complejidad : $O(n\ log(n))$

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int l = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[l-1] < a[i] && a[i] < d[l])
            d[l] = a[i];
    }

    int ans = 0;
    for (int l = 0; l <= n; l++) {
        if (d[l] < INF)
            ans = l;
    }
    return ans;
}
```

3.5 Longest Common Subsequence

Complejidad : $O(n\ m)$, donde n y m son las longitudes de las secuencias.

```
int lcs(string X, string Y, int m, int n)
{

   // Initializing a matrix of size
   // (m+1)*(n+1)
   int L[m + 1][n + 1];

   // Following steps build L[m+1][n+1]
   // in bottom up fashion. Note that
   // L[i][j] contains length of LCS of
   // X[0..i-1] and Y[0..j-1]
   for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
         if (i == 0 || j == 0)
            L[i][j] = 0;

         else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;

         else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
      }
   }

   // L[m][n] contains length of LCS
   // for X[0..n-1] and Y[0..m-1]
   return L[m][n];
}
```

Si queremos recuperar la solución es:

```
int lcs(string X, string Y, int m, int n)
{

   // Initializing a matrix of size
   // (m+1)*(n+1)
   int L[m + 1][n + 1];
   pair<int,int> prev[m + 1][n + 1];

   // Following steps build L[m+1][n+1]
   // in bottom up fashion. Note that
   // L[i][j] contains length of LCS of
   // X[0..i-1] and Y[0..j-1]
```

```
   for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
         if (i == 0 || j == 0){
            L[i][j] = 0;
            prev[i][j] = {-1,-1};
         } else if (X[i - 1] == Y[j - 1]){
            L[i][j] = L[i - 1][j - 1] + 1;
            prev[i][j] = {i-1,j-1};
         } else {
            if(L[i-1][j] > L[i][j-1]){
               prev[i][j] = prev[i-1][j];
            }else{
               prev[i][j] = prev[i][j-1];
            }
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
         }
      }
   }

   // L[m][n] contains length of LCS
   // for X[0..n-1] and Y[0..m-1]
   string ans = "";
   pair<int,int> previous = prev[m][n];
   while(!(previous.first == -1 && previous.second == -1)){
      ans += X[previous.first];
      previous = prev[previous.first][previous.second];
   }

   reverse(ans.begin(),ans.end());
   cout << ans << endl;
   return L[m][n];
}
```

4 Geometría

4.1 Convex Hull

Complejidad : $O(n\ log(n))$

```
struct pt {
   double x, 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; // clockwise
    if (v > 0) return +1; // counter-clockwise
    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;
}
```

4.2 Conseguir Circuncentro

Complejidad : $O(1)$

```
#define pdd pair<double, double>
```

```
void lineFromPoints(pdd P, pdd Q, double &a,
                    double &b, double &c)
{
    a = Q.second - P.second;
    b = P.first - Q.first;
    c = a*(P.first)+ b*(P.second);
}

void perpendicularBisectorFromLine(pdd P, pdd Q,
               double &a, double &b, double &c)
{
    pdd mid_point = make_pair((P.first + Q.first)/2,
                        (P.second + Q.second)/2);

    c = -b*(mid_point.first) + a*(mid_point.second);

    double temp = a;
    a = -b;
    b = temp;
}


pdd lineLineIntersection(double a1, double b1, double c1,
                     double a2, double b2, double c2)
{
    double determinant = a1*b2 - a2*b1;
    if (determinant == 0)
    {
        // The lines are parallel. This is simplified
        // by returning a pair of FLT_MAX
        return make_pair(FLT_MAX, FLT_MAX);
    }

    else
    {
        double x = (b2*c1 - b1*c2)/determinant;
        double y = (a1*c2 - a2*c1)/determinant;
        return make_pair(x, y);
    }
}

void findCircumCenter(pdd P, pdd Q, pdd R)
{
    double a, b, c;
    lineFromPoints(P, Q, a, b, c);
```

```
    double e, f, g;
    lineFromPoints(Q, R, e, f, g);

    perpendicularBisectorFromLine(P, Q, a, b, c);
    perpendicularBisectorFromLine(Q, R, e, f, g);

    pdd circumcenter =
          lineLineIntersection(a, b, c, e, f, g);

    if (circumcenter.first == FLT_MAX &&
        circumcenter.second == FLT_MAX)
    {
        cout << "The two perpendicular bisectors "
                "found come parallel" << endl;
        cout << "Thus, the given points do not form "
                "a triangle and are collinear" << endl;
    }

    else
    {
        cout << "The circumcenter of the triangle PQR is: ";
        cout << "(" << circumcenter.first << ", "
             << circumcenter.second << ")" << endl;
    }
}
```

5 Teoría de Números

5.1 Criba de Erastotenes

Complejidad : $O(n\ ln(ln(n)))$

```
int n;
vector<bool> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i * i <= n; i++) {
    if (is_prime[i]) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

5.2 Test de Primalidad

Complejidad : $O(\sqrt[4]{n})$ (aprox)

```
bool MillerRabin(tint n) { // returns true if n is prime, else returns false.
    if (n < 2)
        return false;

    int r = 0;
    tint d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}
```

Nota: Si n es un int (no long long), alcanza con iterar por 2,3,5,7.

5.3 Factorización en Primos

Complejidad : $O(\sqrt[4]{n}\ log(n))$ (aprox)

```
long long gcd(long long a,long long b){
    if(b == 0) return a;
    return gcd(b,a%b);
}

long long mult(long long a, long long b, long long mod) {
    return (__int128)a * b % mod;
}

long long f(long long x, long long c, long long mod) {
    return (mult(x, x, mod) + c) % mod;
}

long long rho(long long n, long long x0=2, long long c=1) {
    long long x = x0;
    long long y = x0;
    long long g = 1;
```

```
    while (g == 1) {
        x = f(x, c, n);
        y = f(y, c, n);
        y = f(y, c, n);
        g = gcd(abs(x - y), n);
    }
    return g;
}

void factor(int n,map<long long,long long>& factors){
    if(n == 1) return;
    if(is_prime[n]){
        factors[n]++;
        return;
    }
    long long d = rho(n);
    factor(d,factors);
    factor(n / d,factors);
}
```

Nota 1: Es necesario hacer la criba de erastotenes previo a usar el algoritmo (y guardar en is prime).
Nota 2: si no tenemos int128 para mult, podemos usar

```
long long mult(long long a, long long b, long long mod) {
    long long result = 0;
    while (b) {
        if (b & 1)
            result = (result + a) % mod;
        a = (a + a) % mod;
        b >>= 1;
    }
    return result;
}
```

5.4 Inverso Modular

Complejidad : $O(1)$ (Not really pero muy rápido, menos de 50 operaciones para números menores a 10^9)

```
int inv(int a) {
  return a <= 1 ? a : m - (long long)(m/a) * inv(m % a) % m;
}
```

6 Algoritmos sobre Grafos

En todos los algoritmos sobre grafos, la complejidad puede estar expresada en función de V y E.

- $V \Rightarrow$ cantidad de nodos en el grafo
- $E \Rightarrow$ cantidad de aristas en el grafo

6.1 Recorrido de grafos

6.1.1 DFS (Depth First Search)

Complejidad : $O(|V| + |E|)$

```
vector<vector<tint>> aristas;
vector<bool> visitado;

void dfs(tint v) {
    visitado[v] = true;
    for (auto u : aristas[v]) {
        if (!visitado[u]) dfs(u);
    }
}
```

6.1.2 BFS (Breadth First Search)

Complejidad : $O(|V| + |E|)$

```
vector<vector<tint>> aristas;
vector<bool> visitado;
vector<tint> distancia;

void bfs(tint s) {
    visitado[s] = true;
    distancia[s] = 0;

    queue<tint> q;
    q.push(s);

    while (!q.empty()) {
        auto u = q.front(); q.pop();

        for (auto v : aristas[u]) {
            if (visitado[v]) continue;
            visitado[v] = true;
            distancia[v] = distancia[u] + 1;
            q.push(v);
        }
    }
}
```

6.2 Maximum Matching en Grafo Bipartito

Complejidad : $O(|V|\ |E|)$

```
int M,N;


bool bpm(vector<vector<bool>> &bpGraph, int u,
       vector<bool> &seen, vector<int> &matchR)
{
   for (int v = 0; v < N; v++)
   {
       if (bpGraph[u][v] && !seen[v])
       {
           seen[v] = true;

           if (matchR[v] < 0 || bpm(bpGraph, matchR[v],
                                seen, matchR))
           {
               matchR[v] = u;
               return true;
           }
       }
   }
   return false;
}

//bpGraph debe estar en formato grafo bipartito, es decir
//Las filas son la cantidad de elementos del lado izq y en las columnas tenemos
    si se conecta o no con uno del otro lado es decir
//bpGraph[i][j] = true sii el i-esimo del lado izq se conecta con el j-esimo del
    lado der
int maxBPM(vector<vector<bool>> &bpGraph)
{

   vector<int> matchR (N,-1);


   int result = 0;
   for (int u = 0; u < M; u++)
   {
       vector<bool> seen(N,false);

       if (bpm(bpGraph, u, seen, matchR))
           result++;
   }
```

```
    return result;
}
```

6.3 Least Common Ancestor (LCA)

Complejidad Queries: $O(log(V))$
Complejidad Precomputo: $O(V\ log(V))$

```
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

    for (int u : adj[v]) {
        if (u != p)
            dfs(u, v);
    }

    tout[v] = ++timer;
}

bool is_ancestor(int u, int v)
{
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v)
{
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
```

```
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```

6.4 Camino mínimo

6.4.1 Dijkstra's algorithm $O(|E|\ log(|V|))$

Disclaimer: camino mínimo de 1 a todos, en grafos dirigidos o no dirigidos las aristas tienen que tener peso positivo.
Complejidad : $O(|E|\ log(|V|))$

```
tint n, m;
const tint INF = 1e18;
vector<vector<pair<tint, tint>>> adj;
vector<tint> dist;
vector<bool> processed;

void dijkstra(tint x){
    priority_queue<pair<tint,tint>> q;
    for (tint i = 0; i < n; i++) dist[i] = INF;
    dist[x] = 0; q.push({0,x});
    while (!q.empty()) {
        tint a = q.top().second; q.pop();
        if (processed[a]) continue;
        processed[a] = true;
        for (auto u : adj[a]) {
            tint b = u.first, w = u.second;
            if (dist[a]+w < dist[b]) {
                dist[b] = dist[a]+w;
                q.push({-dist[b],b});
            }
        }
    }
}
```

6.4.2 Dijkstra's algorithm $O(|V|^2 + |E|)$

Complejidad : $O(|V|^2 + |E|))$

```
tint n, m;
const tint INF = 1e18;
vector<vector<pair<tint, tint>>> adj;
vector<tint> dist;
vector<tint> p;

void dijkstra(tint s, vector<tint> & d, vector<tint> & p) {
    tint n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (tint i = 0; i < n; i++) {
        tint v = -1;
        for (tint j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

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

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}
```

6.4.3 Bellman–Ford algorithm

Disclaimer: camino mínimo de 1 a todos, en un grafo dirigido las aristas pueden tener peso negativo mientras no haya ciclos negativos.
Complejidad : $O(|V|\ |E|)$

```
tint INF = 1000000000;
```

```
vector<vector<pair<tint, tint>>> aristas;
vector<tint> dist;
tint n;

void bellmanFord(tint x) {
    vector<tint> dist(n, INF);
    dist[x] = 0;

    for (tint i = 1; i <= n - 1; i++) {
        for (tint u = 0; u < n; u++) {
            for (auto edge : aristas[u]) {
                tint v = edge.first;
                tint w = edge.second;
                if (dist[u] != INF && dist[u] + w < dist[v]) {
                    dist[v] = dist[u] + w;
                }
            }
        }
    }

    /* Checkeo de ciclos negativos
    for (tint u = 0; u < n; u++) {
        for (const pair<tint, tint>& edge : graph[u]) {
            tint v = edge.first;
            tint w = edge.second;
            if (dist[u] != INF && dist[u] + w < dist[v]) {
                cout << "Graph contains a negative cycle" << endl;
                return;
            }
        }
    }
    */
}
```

6.4.4 Floyd-Warshall algorithm

Disclaimer: camino mínimo de todos a todos, en un grafo dirigido o no dirigido las aristas pueden tener peso negativo mientras no haya ciclos negativos.
Complejidad : $O(|V|^3)$

```
const tint INF = 1e18;
vector<vector<tint>> aristas;
vector<vector<tint>> dist;

void floydWarshall() {
```

```
    // Calcular distancias minimas entre todos los pares de nodos
    for (tint k = 0; k < V; k++) {
        for (tint i = 0; i < V; i++) {
            for (tint j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF)
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }

}

int main(){
    aristas.assign(n, vector<tint>());
    dist.assign(n, vector<tint>(n, INF));
    for(tint i = 0; i < n; i++) {
        dist[i][i] = 0;
    }
}
```

6.4.5 Johnson's algorithm

Disclaimer: camino mínimo de todos a todos, en un grafo dirigido o no dirigido las aristas pueden tener peso negativo mientras no haya ciclos negativos. Este algoritmo es mejor que Floyd-Warshall para grafos esparsos.
Complejidad : $O(|V|^2\ log(|V|) + |V||E|)$

```
typedef long long tint;
tint INF = 1e18;

// Estructura para representar una arista
struct Edge {
    tint from;
    tint to;
    tint w;
};

// Funcion para el algoritmo de Bellman-Ford
void bellmanFord(const vector<Edge>& edges, tint n, vector<tint>& dist) {
    dist.assign(n, INF);
    dist[n - 1] = 0;

    for (tint i = 0; i < n - 1; ++i) {
        for (const Edge& e : edges) {
            if (dist[e.from] != INF && dist[e.from] + e.w < dist[e.to]) {
```

```
            dist[e.to] = dist[e.from] + e.w;
        }
    }
}

// Funcion para el algoritmo de Dijkstra
void dijkstra(const vector<vector<Edge>>& adjList, tint source, vector<tint>&
    dist, vector<tint>& previo) {
    tint n = adjList.size();
    dist.assign(n, INF);
    previo.assign(n, -1);
    dist[source] = 0;

    priority_queue<pair<tint, tint>, vector<pair<tint, tint>>, greater<pair<tint,
        tint>>> pq;
    pq.push({0, source});

    while (!pq.empty()) {
        tint u = pq.top().second;
        tint d = pq.top().first;
        pq.pop();

        if (d > dist[u]) {
            continue;
        }

        for (const Edge& e : adjList[u]) {
            if (dist[u] + e.w < dist[e.to]) {
                dist[e.to] = dist[u] + e.w;
                previo[e.to] = u;
                pq.push({dist[e.to], e.to});
            }
        }
    }
}

// Funcion principal del algoritmo de Johnson
void johnsonAlgorithm(const vector<vector<Edge>>& adjList, tint n, vector<vector<
    tint>>& distancias, vector<vector<tint>>& previos) {
    vector<Edge> edges;
    for (tint i = 0; i < n; ++i) {
        for (const Edge& e : adjList[i]) {
            edges.push_back(e);
        }
```

```
    }

    vector<tint> minCaminos;
    bellmanFord(edges, n, minCaminos);

    for (Edge& e : edges) {
        e.w = e.w + minCaminos[e.from] - minCaminos[e.to];
    }

    for (tint i = 0; i < n; ++i) {
        vector<tint> dist;
        vector<tint> previo;
        dijkstra(adjList, i, dist, previo);
        previos[i] = previo;

        for (tint j = 0; j < n; ++j) {
            distancias[i][j] = dist[j] - minCaminos[i] + minCaminos[j];
        }
    }
}

int main() {

    tint n; cin >> n;
    vector<vector<Edge>> adjList(n);

    // Llenar adjList con las aristas del grafo

    vector<vector<tint>> distancias(n, vector<tint>(n));
    vector<vector<tint>> previos(n, vector<tint>(n));

    johnsonAlgorithm(adjList, n, distancias, previos);

    // Imprimir las matrices de distancias y previos

    return 0;
}
```

6.5 Árbol Generador Mínimo (MST)

6.5.1 Prim's algorithm $O(|V|^2)$

Complejidad : $O(V^2)$

```
typedef long long tint;
const tint INF = 1e18;
```

```
tint minKey(const vector<tint>& key, const vector<bool>& mstSet, tint n) {
    tint minVal = INF, minIndex = -1;
    for (tint v = 0; v < n; ++v) {
        if (!mstSet[v] && key[v] < minVal) {
            minVal = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void prim(const vector<vector<tint>>& graph, tint n, vector<tint>& parent) {
    vector<tint> key(n, INF);
    vector<bool> mstSet(n, false);

    key[0] = 0;
    parent[0] = -1;

    for (tint count = 0; count < n - 1; ++count) {
        tint u = minKey(key, mstSet, n);
        mstSet[u] = true;

        for (tint v = 0; v < n; ++v) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
}

int main() {
    tint n, m;
    cin >> n >> m;

    vector<vector<tint>> graph(n, vector<tint>(n, 0));

    for (tint i = 0; i < m; ++i) {
        tint from, to, weight;
        cin >> from >> to >> weight;
        graph[from][to] = graph[to][from] = weight;
    }

    vector<tint> parent(n);
```

```
    prim(graph, n, parent);


    return 0;
}
```

6.5.2 Kruskal's algorithm $O(|E|\ log(|V|))$

Esta implementación usa DSU, es eficiente para lograr la complejidad buscada. Complejidad : $O(E\ log(V))$

```
typedef long long tint;
tint INF = 1e18;

vector<tint> parent;
vector<tint> ranking;

void make_set(tint v) {
    parent[v] = v;
    ranking[v] = 0;
}

tint find_set(tint v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}

void union_sets(tint a, tint b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (ranking[a] < ranking[b])
            swap(a, b);
        parent[b] = a;
        if (ranking[a] == ranking[b])
            ranking[a]++;
    }
}

struct Edge {
    tint from, to, w;
    bool operator<(Edge const& other) {
        return w < other.w;
    }
};
```

```
tint n;
vector<Edge> edges;

tint cost = 0;
vector<Edge> result;
int main(){
    parent.resize(n);
    ranking.resize(n);
    for (tint i = 0; i < n; i++){
        make_set(i);
    }

    //Agregar aristas al grafo edges
    sort(edges.begin(), edges.end());

    for (auto e : edges) {
        if (find_set(e.from) != find_set(e.to)) {
            cost += e.w;
            result.push_back(e);
            union_sets(e.from, e.to);
        }
    }
}
```

6.6 Max flow/Min cut

6.6.1 Dinic's algorithm

Complejidad : $O(|V|^2|E|)$

```
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
```

```
    adj.resize(n);
    level.resize(n);
    ptr.resize(n);
}

void add_edge(int v, int u, long long cap) {
    edges.emplace_back(v, u, cap);
    edges.emplace_back(u, v, 0);
    adj[v].push_back(m);
    adj[u].push_back(m + 1);
    m += 2;
}

bool bfs() {
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int id : adj[v]) {
            if (edges[id].cap - edges[id].flow < 1)
                continue;
            if (level[edges[id].u] != -1)
                continue;
            level[edges[id].u] = level[v] + 1;
            q.push(edges[id].u);
        }
    }
    return level[t] != -1;
}

long long dfs(int v, long long pushed) {
    if (pushed == 0)
        return 0;
    if (v == t)
        return pushed;
    for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
        int id = adj[v][cid];
        int u = edges[id].u;
        if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
            continue;
        long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
        if (tr == 0)
            continue;
        edges[id].flow += tr;
        edges[id ^ 1].flow -= tr;
        return tr;
```

```
        }
        return 0;
    }

    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
```

6.7 Otras ideas sobre grafos

6.7.1 Finding bridges

Complejidad : $O(V + E)$

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to); //la arista (v,to) es puente
```

```
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

Nota: No funciona bien en multigrafos.

6.7.2 Finding articulation points

Complejidad : $O(V + E)$

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v); //v es punto de articulacion
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v); //v es punto de articulacion
}
```

```
void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

Nota: La función IS_CUTPOINT podría ser llamada múltiples veces para el mismo vértice.

6.7.3 Kosaraju - Strongly Connected Components

Complejidad : $O(V + E)$

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

using namespace std;

vector<vector<int>> adj; // Lista de adyacencia del grafo original
vector<vector<int>> revAdj; // Lista de adyacencia del grafo transpuesto
vector<bool> visited; // Vector de visitados
stack<int> orderStack; // Pila de orden topológico
vector<int> component; // Componente fuertemente conexa actual

// Función DFS para el primer recorrido del grafo
void dfs1(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs1(u);
        }
    }
    orderStack.push(v); // Apilar el nodo en la pila
}

// Función DFS para el segundo recorrido del grafo transpuesto
void dfs2(int v) {
    visited[v] = true;
    component.push_back(v); // Agregar el nodo a la componente actual
    for (int u : revAdj[v]) {
        if (!visited[u]) {
            dfs2(u);
        }
```

```
    }
}

// Función para encontrar las componentes fuertemente conexas del grafo (n es
    cantidad de vertices)
vector<vector<int>> findStronglyConnectedComponents(int n) {
    visited.assign(n, false);

    // Paso 1: Realizar el primer recorrido DFS y almacenar el orden topológico
    for (int v = 0; v < n; ++v) {
        if (!visited[v]) {
            dfs1(v);
        }
    }

    // Paso 2: Transponer el grafo
    revAdj.assign(n, vector<int>());
    for (int v = 0; v < n; ++v) {
        for (int u : adj[v]) {
            revAdj[u].push_back(v);
        }
    }

    visited.assign(n, false);
    vector<vector<int>> components; // Lista de componentes fuertemente conexas

    // Paso 3: Realizar el segundo recorrido DFS en el grafo transpuesto
    while (!orderStack.empty()) {
        int v = orderStack.top();
        orderStack.pop();
        if (!visited[v]) {
            dfs2(v);
            components.push_back(component);
            component.clear();
        }
    }

    return components;
}
```

Nota: Además nos da el orden topológico de las componentes conexas.

6.7.3.1 Condensation Graph Implementation

Complejidad : $O(V + E)$

```
vector<int> roots(n, 0);
```

```
vector<int> root_nodes;
vector<vector<int>> adj_scc(n);

for (auto v : order)
    if (!used[v]) {
        dfs2(v);

        int root = component.front();
        for (auto u : component) roots[u] = root;
        root_nodes.push_back(root);
        //components.push_back(component); Esto es lo que haciamos en el codigo
            anterior.

        component.clear();
    }


for (int v = 0; v < n; v++)
    for (auto u : adj[v]) {
        int root_v = roots[v],
            root_u = roots[u];

        if (root_u != root_v)
            adj_scc[root_v].push_back(root_u);
    }
```

6.7.4 Topological Sort

Complejidad : $O(V + E)$

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
```

```
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
    reverse(ans.begin(), ans.end());
}
```

Nota 1: El orden topológico implica que si hay una arista de $a \to b$, entonces a aparecerá antes que b en el orden.
Nota 2: El orden topológico puede ser no único.

7 Particularidades de C++

7.1 Comparator en maps

Template básico para un comparator en C++:

```
struct Comparador {
    bool operator()(const string& a, const string& b) const {
        return a.length() < b.length(); // Ordena por long de la clave
    }
};
```

7.2 Imprimir con n decimales

Para imprimir con n decimales en C++, lo hacemos importando la librería **iomanip** en caso de no haber importado la **bits/stdc++.h** desde un comienzo. Entonces en código base para printear un numero con n decimales, puede ser:

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

using namespace std;
int main() {
    long double pi = 3.14159265358979323846;
    int n = 10;
    cout << fixed << setprecision(n) << pi << endl;

    //Caso enteros:
    long long k = 5;
    cout << fixed << setprecision(n) << (long double) k << endl;
    return 0;
```

Si se quisiera imprimir un entero (por ej. un *long long*) con n cantidad de ceros atrás, transformar ese **long long** a **long double** y usar el método de arriba.

7.3 Strings operations

7.3.1 Find & rfind

```
#include <iostream>
#include <string>

int main() {
   std::string str ("There␣are␣two␣needles␣in␣this␣haystack␣with␣needles.");
   std::string str2 ("needle");

   // different member versions of find in the same order as above:
   std::size_t found = str.find(str2);
   if (found!=std::string::npos)
      std::cout << "first␣'needle'␣found␣at:␣" << found << '\n';

   found=str.find("needles␣are␣small",found+1,6);
   if (found!=std::string::npos)
      std::cout << "second␣'needle'␣found␣at:␣" << found << '\n';

   found=str.find("haystack");
   if (found!=std::string::npos)
      std::cout << "'haystack'␣also␣found␣at:␣" << found << '\n';

   found=str.find('.');
   if (found!=std::string::npos)
      std::cout << "Period␣found␣at:␣" << found << '\n';

   // let's replace the first needle:
   str.replace(str.find(str2),str2.length(),"preposition");
   std::cout << str << '\n';

   //last occurrence of content in string
   std::string str ("The␣sixth␣sick␣sheik's␣sixth␣sheep's␣sick.");
   std::string key ("sixth");

   std::size_t found = str.rfind(key);
   if (found!=std::string::npos)
      str.replace (found,key.length(),"seventh");

   std::cout << str << '\n';

   return 0;
}
```

7.3.2 Substrings

```
#include <iostream>
#include <string>

int main (){
   std::string str="We␣think␣in␣generalities,␣but␣we␣live␣in␣details.";

   std::string str2 = str.substr (3,5); // "think"

   std::size_t pos = str.find("live"); // position of "live" in str

   std::string str3 = str.substr (pos); // get from "live" to the end

   std::cout << str2 << '␣' << str3 << '\n';

   return 0;
   }
```

7.3.3 Comparar

```
#include <iostream>
#include <string>

int main (){
   std::string str1 ("green␣apple");
   std::string str2 ("red␣apple");

   if (str1.compare(str2) != 0)
      std::cout << str1 << "␣is␣not␣" << str2 << '\n';

   if (str1.compare(6,5,"apple") == 0)
      std::cout << "still,␣" << str1 << "␣is␣an␣apple\n";

   if (str2.compare(str2.size()-5,5,"apple") == 0)
      std::cout << "and␣" << str2 << "␣is␣also␣an␣apple\n";

   if (str1.compare(6,5,str2,4,5) == 0)
      std::cout << "therefore,␣both␣are␣apples\n";

   return 0;
}
```

8 Estructuras de Datos

8.1 Segment Tree

Todo en esta sección tiene las siguientes complejidades:
Complejidad Queries: $O(log(n))$
Complejidad Precomputo: $O(n\ log(n))$

8.1.1 Segment Tree Default

Esta implementación está basada en calcular la suma del rango pero el resto de operaciones son similares. Para construir el Segment Tree hay que llamar a:

```
int n, t[4*MAXN];
void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = t[v*2] + t[v*2+1];
    }
}
```

Con los parametros v = 0, tl = 0, tr = n-1. t tiene que tener 4 veces el tamaño de n, y se puede cambiar int a[] por vector<int>& a.

```
int sum(int v, int tl, int tr, int l, int r) {
    if (l > r)
        return 0;
    if (l == tl && r == tr) {
        return t[v];
    }
    int tm = (tl + tr) / 2;
    return sum(v*2, tl, tm, l, min(r, tm))
          + sum(v*2+1, tm+1, tr, max(l, tm+1), r);
}

void update(int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr) {
        t[v] = new_val;
    } else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update(v*2, tl, tm, pos, new_val);
        else
            update(v*2+1, tm+1, tr, pos, new_val);
```

```
        t[v] = t[v*2] + t[v*2+1];
    }
}
```

Nota: tl y tr es el rango que "cubre el elemento", l y r los del query.

8.1.2 Lazy Segtree Cuando el Update es sumar un determinado x al rango

```
void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = 0;
    }
}

void update(int v, int tl, int tr, int l, int r, int add) {
    if (l > r)
        return;
    if (l == tl && r == tr) {
        t[v] += add;
    } else {
        int tm = (tl + tr) / 2;
        update(v*2, tl, tm, l, min(r, tm), add);
        update(v*2+1, tm+1, tr, max(l, tm+1), r, add);
    }
}

int get(int v, int tl, int tr, int pos) {
    if (tl == tr)
        return t[v];
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return t[v] + get(v*2, tl, tm, pos);
    else
        return t[v] + get(v*2+1, tm+1, tr, pos);
}
```

8.1.3 Lazy Segtree cuando el update es settear a un valor especifico

```
void push(int v) {
    if (marked[v]) {
```

```
        t[v*2] = t[v*2+1] = t[v];
        marked[v*2] = marked[v*2+1] = true;
        marked[v] = false;
    }
}

void update(int v, int tl, int tr, int l, int r, int new_val) {
    if (l > r)
        return;
    if (l == tl && tr == r) {
        t[v] = new_val;
        marked[v] = true;
    } else {
        push(v);
        int tm = (tl + tr) / 2;
        update(v*2, tl, tm, l, min(r, tm), new_val);
        update(v*2+1, tm+1, tr, max(l, tm+1), r, new_val);
    }
}

int get(int v, int tl, int tr, int pos) {
    if (tl == tr) {
        return t[v];
    }
    push(v);
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return get(v*2, tl, tm, pos);
    else
        return get(v*2+1, tm+1, tr, pos);
}
```

Nota: Hay que crear un array con marked para todos los nodos del segtree.

8.2 Trie

```
const int K = 26;

struct Vertex {
    int next[K];
    bool output = false;

    Vertex() {
        fill(begin(next), end(next), -1);
    }
```

```
};

vector<Vertex> trie(1);

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (trie[v].next[c] == -1) {
            trie[v].next[c] = trie.size();
            trie.emplace_back();
        }
        v = trie[v].next[c];
    }
    trie[v].output = true;
}
```

8.3 Policy Based Set

Para incluirlo:

```
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp>
#include <functional> // for less
using namespace __gnu_pbds;
```

Para crearlo para ints:

```
typedef tree<int, null_type, less<int>, rb_tree_tag,
           tree_order_statistics_node_update>
    ordered_set;
```

Y si queremos repetidos (o tuplas):

```
typedef tree<pair<int, int>, null_type,
           less<pair<int, int> >, rb_tree_tag,
           tree_order_statistics_node_update>
    ordered_multiset;
```

Tiene 3 operaciones: insert, find_by_order y order_of_key

```
int main()
{
    ordered_set p;
    p.insert(5);
    p.insert(2);
    p.insert(6);
    p.insert(4);
```

```
    // value at 3rd index in sorted array.
    cout << "The value at 3rd index ::"
         << *p.find_by_order(3) << endl;

    // index of number 6
    cout << "The index of number 6::" << p.order_of_key(6)
         << endl;

    // number 7 not in the set but it will show the
    // index number if it was there in sorted array.
    cout << "The index of number seven ::"
         << p.order_of_key(7) << endl;

    return 0;
}
```

8.4 Treap

8.4.1 Default Treap

```
struct Node {
  tint priority, key;
  tint subtreeSize = 1;
  NodePtr left, right;

  Node(tint k) {
    priority = rand();
    key = k;
    left = right = nullptr;
  }
};

inline tint size(NodePtr root) { return (root ? root->subtreeSize : 0); }

void push(NodePtr root) {
  if (root == nullptr) return;
  root->subtreeSize = 1 + size(root->left) + size(root->right);
}

void merge(NodePtr &root, NodePtr left, NodePtr right) {
  push(left), push(right);

  if (left == nullptr || right == nullptr)
    root = (left != nullptr ? left : right);
```

```
  else if (left->priority > right->priority)
    merge(left->right, left->right, right), root = left;
  else
    merge(right->left, left, right->left), root = right;

  push(root);
}

void split(NodePtr root, NodePtr &left, NodePtr &right, tint key) {
  if (root == nullptr) return void(left = right = nullptr);

  push(root);
  if (root->key <= key)
    split(root->right, root->right, right, key), left = root;
  else
    split(root->left, left, root->left, key), right = root;
  push(left), push(right);
}

void insert(NodePtr &root, tint key) {
  NodePtr left, right;
  split(root, left, right, key);

  merge(left, left, new Node(key));
  merge(root, left, right);
}

tint getRank(NodePtr &root, tint key) {
  NodePtr left, right;
  split(root, left, right, key);

  tint ans = size(left);
  merge(root, left, right);
  return ans;
}

tint select(NodePtr &root, tint index, tint offset = 0) {
  if (root == nullptr) return -1;

  tint currRank = offset + size(root->left) + 1;
  if (currRank == index) {
    return root->key;
  } else if (currRank < index) {
    return select(root->right, index, currRank);
  } else {
```

```
    return select(root->left, index, offset);
  }
}
```

8.4.2 Treap que permite obtener el máximo en segunda coordenada entre los que tienen primera coordenada menor

```
struct Node;
using NodePtr = Node *;
const tint INF = 1e18;

struct Node {
  tint priority, key;
  tint value, subtreeMax;
  NodePtr left, right;

  Node(tint k, tint v) {
    priority = rand();
    key = k;
    subtreeMax = value = v;
    left = right = nullptr;
  }
};

tint subtreeMax(NodePtr node) {
  if (node == nullptr) return -INF;
  return node->subtreeMax;
}

void push(NodePtr node) {
  if (node == nullptr) return;
  node->subtreeMax =
      max({node->value, subtreeMax(node->left), subtreeMax(node->right)});
}

void merge(NodePtr &root, NodePtr left, NodePtr right) {
  if (left == nullptr || right == nullptr) {
    root = (left != nullptr ? left : right);
  } else if (left->priority > right->priority) {
    merge(left->right, left->right, right);
    root = left;
  } else {
    merge(right->left, left, right->left);
    root = right;
  }
```

```
  push(root);
}

void split(NodePtr root, NodePtr &left, NodePtr &right, tint key) {
  if (root == nullptr) return void(left = right = nullptr);

  push(root);

  if (root->key <= key) {
    split(root->right, root->right, right, key);
    left = root;
  } else {
    split(root->left, left, root->left, key);
    right = root;
  }

  push(left), push(right);
}

void insert(NodePtr &root, tint x, tint y) {
  NodePtr left, right;
  split(root, left, right, x);
  merge(left, left, new Node(x, y));
  merge(root, left, right);
}

tint query(NodePtr &root, tint x) {
  NodePtr left, right;

  split(root, left, right, x);
  tint ans = subtreeMax(left);
  merge(root, left, right);

  return ans;
}
```

Nota: Gracias Tarche ;)