Citric Vindicators - Team notebook

Contents

1 Template

1.1 Template C++

```cpp
#include <bits/stdc++.h>
// Pura gente del coach moy
using namespace std;
#define ENDL '\n'
#define all(x)  x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()
#define sz(x)   (int) x.size()
#define FOR(x, b)     for(int x = 0; x < b; x++)
#define FORE(x, a, b) for(int x = a; x <= b; x++)
#define FORR(x, a, b) for(int x = a; x >= b; x--)
#define deb(x)      cerr << #x << " = " << x << '\n';
#define deb2(x, y)  cerr << #x << " = " << x << ", " << #y << " = " << y << '\n';
#define _ ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
const ll MOD = 1e9+7, INF = 1e18;

ll gcd(ll a, ll b){ return (b ? gcd(b, a % b) : a); }
ll lcm(ll a, ll b){ if(!a || !b) return 0; return a * b / gcd(a, b); }

void solve(){

}

int main(){_
    int tc;
    cin>>tc;
    while(tc--)
        solve();
    return 0;
}
```

2 Math

2.1 Mod Pow-Inverse

```cpp
// Retorna a % m, asegurando siempre una respuesta positiva
ll mod(ll a, ll m) { return (a % m + m) % m; }

ll modPow(ll b, ll p, ll m){                // O(log n)
    b %= m;
    ll ans = 1;
    while(p){
        if(p & 1) ans = mod(ans * b, m);
        b = mod(b * b, m);
        p >>= 1;
```

```
    }
    return ans;
}

int extEuclid(int a, int b, int &x, int &y) {
    int xx = y = 0;
    int yy = x = 1;
    while (b) {
        int q = a/b;
        tie(a, b) = tuple(b, a%b);
        tie(x, xx) = tuple(xx, x-q*xx);
        tie(y, yy) = tuple(yy, y-q*yy);
    }
    return a;                                    // Retorna gcd(a, b)
}

int modInverse(int b, int m) {                   // Retorna b^(-1) (mod m)
    int x, y;
    int d = extEuclid(b, m, x, y);               // Para obtener b*x + m*y == d
    if (d != 1) return -1;                       // Para indicar fallo
    // b*x + m*y == 1, ahora se aplica (mod m) para obtener b*x == 1 (mod m)
    return mod(x, m);
}

// Solo cuando m es primo
int modInverse(int b, int m){ return modPow(b, m - 2, m) % m; }
```

2.2 Primes

```
// Implementacion de muchas funciones utiles respecto a primos
typedef long long ll;
typedef vector<ll> vll;

ll _sieve_size;
bitset<10000010> bs;
vll p;

void sieve(ll upperbound) { // Calcula la criba en O(N log(log N))
    _sieve_size = upperbound+1;
    bs.set();
    bs[0] = bs[1] = 0;
    for (ll i = 2; i < _sieve_size; ++i) if (bs[i]){
        // Tacha los multiplos de i a partir de i*i
        for (ll j = i*i; j < _sieve_size; j += i) bs[j] = 0;
        p.push_back(i);
    }
}

// Criba con complejidad O(n)
void linear_sieve(int N) {
    vector<int> lp(N + 1);
    vector<int> pr;

    for (int i = 2; i <= N; ++i) {
        if (lp[i] == 0) {
            lp[i] = i;
            pr.push_back(i);
        }
        for (int j = 0; i * pr[j] <= N; ++j) {
            lp[i * pr[j]] = pr[j];
            if (pr[j] == lp[i]) {
                break;
            }
        }
    }
}

bool isPrime(ll N) {    // Regresa si N es primo. Solo se garantiza su funcionamiento para N <= (
      ultimo primo en vll p)^2
    if (N < _sieve_size) return bs[N];
    for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
        if (N%p[i] == 0)
        return false;
    return true;
}

vll primeFactors(ll N) {    // Regresa un vector con los factores primos de N
    vll factors;
    for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
        while (N%p[i] == 0) {
            N /= p[i];
            factors.push_back(p[i]);
        }
    if (N != 1) factors.push_back(N);
    return factors;
}

int numPF(ll N) {   // Regresa el numero de factores primos de N
    int ans = 0;
    for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
        while (N%p[i] == 0) { N /= p[i]; ++ans; }
    return ans + (N != 1);
}

int numDiffPF(ll N) {   // Regresa el numero de factores primos diferentes de N
    int ans = 0;
    for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i) {
        if (N%p[i] == 0) ++ans;
        while (N%p[i] == 0) N /= p[i];
    }
    if (N != 1) ++ans;
    return ans;
}

ll sumPF(ll N) {    // Regresa la suma de los factores primos de N
    ll ans = 0;
    for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i)
        while (N%p[i] == 0) { N /= p[i]; ans += p[i]; }
    if (N != 1) ans += N;
    return ans;
}

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

ll sumDiv(ll N) {   // Regresa la suma de los divisores de N
    ll 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;
}

ll EulerPhi(ll N) { // Regresa cuantos numeros menores a N son coprimos con N
    ll 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];
        while (N%p[i] == 0) N /= p[i];
    }
    if (N != 1) ans -= ans/N;
    return ans;
}

// Calcula la funcion de Mobius, para todo entero menor o igual a n. O(N)
void preMobius(int N) {
    memset(check, false, sizeof(check));
    mu[1] = 1;
    int tot = 0;
    for(int i = 2; i < N; i++) {
        if (!check[i]) {  // i es primo
            prime[tot++] = i;
            mu[i] = -1;
        }
        for(int j = 0; j < tot; j++){
            if (i * prime[j] > N) break;
            check[i * prime[j]] = true;
            if (i % prime[j] == 0) {
                mu[i * prime[j]] = 0;
                break;
            } else {
                mu[i * prime[j]] = -mu[i];
            }
        }
    }
}
```

2.3 Bit operations

```
// NOTA - Si i > 30, usar 1LL
// Siendo S un numero y {i, j} indices 0-indexados:
#define isOn(S, j) (S & (1 << j))
```

```
#define setBit(S, j) (S |= (1 << j))
#define clearBit(S, j) (S &= ~(1 << j))
#define toggleBit(S, j) (S ^= (1 << j))
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1 << n) - 1)

#define modulo(S, N) ((S) & (N - 1)) // retorna S % N, siendo N una potencia de 2
#define isOdd(S) (s & 1)
#define isPowerOfTwo(S) (!(S & (S - 1)))
#define nearestPowerOfTwo(S) (1 << lround(log2(S)))
#define turnOffLastBit(S) ((S) & (S - 1))
#define turnOnLastZero(S) ((S) | (S + 1))
#define turnOffInRange(S, i, j) s &= (((~0) << (j + 1)) | ((1 << i) - 1));
#define turnOffLastConsecutiveBits(S) ((S) & (S + 1))
#define turnOnLastConsecutiveZeroes(S) ((S) | (S - 1))
/*
Si en un problema tenemos un conjunto de menos de 30 elementos y tenemos que probar cual es el "bueno"
Podemos usar una mascara de bits e intentar cada combinacion.
int limit = 1 << (n + 1);
for (int i = 1; i < limit; i++) {
  ....
}
*/
// Funciones integradas por el compilador GNU (GCC)
// IMPORTANTE ---> Si x cabe en un int quitar el ll de cada metodo :D
// Numero de bits encendidos de x
__builtin_popcountll(x);
// Indice del primer (de derecha a izquierda) bit encendido de x
// Por ejemplo __builtin_ffs(0b0001'0010'1100) = 3
__builtin_ffsll(x);
// Cuenta de ceros a la izquierda del primer bit encendido de x
// Utilizado para calcular piso(log2(x)) -> 63 - __builtin_clzll(x)
// Si x es int, utilizar 31 en lugar de 63
// Por ejemplo __builtin_clz(0b0001'0010'1100) = 23 (YA QUE X SE TOMA COMO ENTERO)
__builtin_clzll(x);
// Cuenta de ceros a la derecha del primer uno (de derecha a izquierda
// Por ejemplo __builtin_ctzll(0b0001'0010'1100) = 2
__builtin_ctzll(x);
```

2.4 Binomial coefficients

```
//Binomial Coefficient n choose k
//DP top down manner (memset initialization requiered)
ll comb[MAX][MAX];

ll nCk(ll n, ll k){
    if(k < 0 || k > n){
        return 0;
    }

    if(n == k || k == 0){
        return 1;
    }

    if(comb[n][k] != -1){
        return comb[n][k];
    }

    return comb[n][k] = (nCk(n - 1, k - 1) + nCk(n - 1, k)) % MOD;
}
```

2.5 Catalan numbers

```
# Solution for small range ---> k <= 510. if k is greater, use Java's BigInteger class. if we need to
      only store catalan[i] % m, use c++
catalan = [0 for i in range(510)]
def precalculate():
    catalan[0] = 1
    for i in range(509):
        catalan[i + 1] = ((2*(2*i+1) * catalan[i])/(i+2))
precalculate()
print(int(catalan[505]))
```

2.6 Matrix multiplication and exponentation

```
/*
    Descripcion: estructura de matriz con algunas operaciones basicas
    se suele utilizar para la multiplicacion y/o exponenciacion de matrices
    Aplicaciones:
    Calcular el n-esimo fibonacci en tiempo logaritmico, esto es
    posible ya que para la matriz M = {{1, 1},{1, 0}}, se cumple
    que M^n = {{F[n + 1], F[n]}, {F[n], F[n - 2]}}
    Dado un grafo, su matriz de adyacencia M, y otra matriz P tal que P = M^k,
    se puede demostrar que P[i][j] contiene la cantidad de caminos de longitud k
    que inician en el i-esimo nodo y terminan en el j-esimo.
    Tiempo: O(n^3 * log p) para la exponenciacion y O(n^3) para la multiplicacion
*/

typedef long long ll;

template<typename T>
struct Matrix {
    using VVT = vector<vector<T>>;

    VVT M;
    int n, m;

    Matrix(VVT aux) : M(aux), n(M.size()), m(M[0].size()) {}

    // O(n^3)
    Matrix operator * (Matrix& other) const {
        int k = other.M[0].size();
        VVT C(n, vector<T>(k, 0));
        for(int i=0; i<n; i++)
            for(int j=0; j<k; j++)
                for(int l=0; l<m; l++)
                    C[i][j] = (C[i][j] % MOD + (M[i][l] % MOD * other.M[l][j] % MOD) % MOD) % MOD;
        return Matrix(C);
    }

    // O(n^3 * log p)
    Matrix operator ^ (ll p) const {
        assert(p >= 0);
        Matrix ret(VVT(n, vector<T>(n))), B(*this);
        for(int i=0; i<n; i++)
            ret.M[i][i] = 1;

        while (p) {
            if (p & 1)
                ret = ret * B;
            p >>= 1;
            B = B * B;
        }
        return ret;
    }
};

// Ejemplo de uso calculando el n-esimo fibonacci
// Para una mayor velocidad realizarlo con 4 variables
Matrix<ll> fibMat({{1, 1}, {1, 0}});
ll fibonacci(ll n){ return (n <= 2) ? (n != 0) : (fibMat^n).M[1][0]; }
```

2.7 Fractions

```
/**
 * Descripcion: estructura para manejar fracciones, es util cuando
 * necesitamos gran precision y solo usamos fracciones
 * Tiempo: O(1)
 */
struct Frac {
    int a, b;

    Frac() {}
    Frac(int _a, int _b) {
        assert(_b > 0);
        if ((_a < 0 && _b < 0) || (_a > 0 && _b < 0)) {
        _a = -_a;
        _b = -_b;
        }

        int GCD = gcd(abs(_a), abs(_b));

        a = _a / GCD;
        b = _b / GCD;
    }

    Frac operator*(Frac& other) const { return Frac(a * other.a, b * other.b); }
    Frac operator/(Frac& other) const {
        Frac o(other.b, other.a);
        return (*this) * o;
    }
    Frac operator+(Frac& other) const {
        int sup = a * other.b + b * other.a, inf = b * other.b;
        return Frac(sup, inf);
    }
    Frac operator-(Frac& other) const {
        int sup = a * other.b - b * other.a, inf = b * other.b;
```

```
        return Frac(sup, inf);
    }
    Frac operator*(int& x) const { return Frac(a * x, b); }
    Frac operator/(int& x) const {
        Frac o(1, x);
        return (*this) * o;
    }
    bool operator<(Frac& other) const {  // PROVISIONAL, IMPLEMENTARLA MEJOR SI HACEN FALTA LOWER
          BOUNDS
        if (a != other.a)
        return a < other.a;
        return b < other.b;
    }
    bool operator==(Frac& other) const {
        return a == other.a && b == other.b;
    }
    bool operator!=(Frac& other) const {
        return !(*this == other);
    }
};
```

2.8 Simpson's rule

```
/*
 * Descripcion: Calcula el valor de una integral definida
 * Tiempo: O(pasos)
 */

const int N = 1000 * 1000;  // numero de pasos (entre mas grande mas preciso)

double simpson_integration(double a, double b) {
    double h = (b - a) / N;
    double s = f(a) + f(b);
    for (int i = 1; i <= N - 1; ++i) {
        double x = a + h * i;
        s += f(x) * ((i & 1) ? 4 : 2);
    }
    s *= h / 3;
    return s;
}
```

2.9 Linear Diophantine

```
/*
    Problema: Dado a, b y n. Encuentra 'x' y 'y' que satisfagan la ecuacion ax + by = n.
    Imprimir cualquiera de las 'x' y 'y' que la satisfagan.
*/

ll euclid(ll a, ll b, ll &x, ll &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    ll d = euclid(b, a % b, y, x);
    return y -= a / b * x, d;
}

void solution(int a, int b, int n) {
    int x0, y0, g = euclid(a, b, x0, y0);
    if (n % g != 0) {
        cout << "No Solution Exists" << '\n';
        return;
    }
    x0 *= n / g;
    y0 *= n / g;
    // single valid answer
    cout << "x = " << x0 << ", y = " << y0 << '\n';

    // other valid answers can be obtained through...
    // x = x0 + k*(b/g)
    // y = y0 - k*(a/g)
    for (int k = -3; k <= 3; k++) {
        int x = x0 + k * (b / g);
        int y = y0 - k * (a / g);
        cout << "x = " << x << ", y = " << y << '\n';
    }
}
```

2.10 FFT

```
/*
 * Descripcion: Este algoritmo permite multiplicar dos polinomios de longitud n
 * Tiempo: O(n log n)
 */

typedef long long ll;
typedef double ld;
typedef complex<ld> C;
typedef vector<ld> vd;
typedef vector<ll> vl;

const ld PI = acos(-1.0L);
const ld one = 1;

void fft(vector<C> &a) {
    int n = sz(a), L = 31 - __builtin_clz(n);
    static vector<complex<ld>> R(2, 1);
    static vector<C> rt(2, 1);  // (^ 10% faster if double)
    for (static int k = 2; k < n; k *= 2) {
        R.resize(n);
        rt.resize(n);
        auto x = polar(one, PI / k);
        for(int i = k; i < 2*k; i++)
            rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
    }
    vi rev(n);
    for(int i = 0; i < n; i++)
        rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
    for(int i = 0; i < n; i++)
        if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k) for(int j = 0; j < k; j++) {
            // C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)     /// include-line
            auto x = (ld *)&rt[j + k], y = (ld *)&a[i + j + k];             /// exclude-line
            C z(x[0] * y[0] - x[1] * y[1], x[0] * y[1] + x[1] * y[0]);      /// exclude-line
            a[i + j + k] = a[i + j] - z;
            a[i + j] += z;
        }
}

typedef vector<ll> vl;

vl conv(const vl &a, const vl &b) {
    if (a.empty() || b.empty()) return {};
    vl res(sz(a) + sz(b) - 1);
    int L = 32 - __builtin_clz(sz(res)), n = 1 << L;
    vector<C> in(n), out(n);
    copy(all(a), begin(in));
    for(int i = 0; i < sz(b); i++)
        in[i].imag(b[i]);
    fft(in);
    for (C &x : in) x *= x;
    for(int i = 0; i < n; i++)
        out[i] = in[-i & (n - 1)] - conj(in[i]);
    fft(out);
    for(int i = 0; i < sz(res); i++)
        res[i] = floor(imag(out[i]) / (4 * n) + 0.5);
    return res;
}

vl convMod(const vl &a, const vl &b, const int &M) {
    if (a.empty() || b.empty()) return {};
    vl res(sz(a) + sz(b) - 1);
    int B = 32 - __builtin_clz(sz(res)), n = 1 << B, cut = int(sqrt(M));
    vector<C> L(n), R(n), outs(n), outl(n);
    for(int i = 0; i < sz(a); i++)
        L[i] = C((int)a[i] / cut, (int)a[i] % cut);
    for(int i = 0; i < sz(b); i++)
        R[i] = C((int)b[i] / cut, (int)b[i] % cut);
    fft(L), fft(R);
    for(int i = 0; i < n; i++) {
        int j = -i & (n - 1);
        outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
        outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
    }
    fft(outl), fft(outs);
    for(int i = 0; i < sz(res); i++) {
        ll av = ll(real(outl[i]) + .5), cv = ll(imag(outs[i]) + .5);
        ll bv = ll(imag(outl[i]) + .5) + ll(real(outs[i]) + .5);
        res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
    }
    return res;
}
```

3 Data structure

3.1 Union find (DSU)

```
// Union-Find Disjoint Set usando las heuristicas de compresion de camino y union por rango
typedef vector<int> vi;

class UnionFind {
 private:
    vi p, rank, setSize;
    int numSets;
 public:
    UnionFind(int N) : p(N, 0), rank(N, 0), setSize(N, 1), numSets(N) {
        for (int i = 0; i < N; ++i) p[i] = i;
    }
    int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
    bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
    int numDisjointSets() { return numSets; }
    int sizeOfSet(int i) { return setSize[findSet(i)]; }
    void unionSet(int i, int j) {
        if (isSameSet(i, j)) return;
        int x = findSet(i), y = findSet(j);
        if (rank[x] > rank[y]) swap(x, y);          // Para mantener x mas pequenio que y
        p[x] = y;
        if (rank[x] == rank[y]) ++rank[y];          // Acelaracion por rango
        setSize[y] += setSize[x];
        --numSets;
    }
};
```

3.2 Union find (DSU) with rollback

```
// Union-Find Disjoint-set con la operacion de deshacer una uniones previas y regresar a un tiempo "t"
// Si no es necesaria esta operacion, eliminar st, time() y rollback()
// Time complexity O(log n)
typedef vector<int> vi;
typedef pair<int, int> ii;

struct RollbackUF {
    vi e;
    vector<ii> st;
    RollbackUF(int n) : e(n, -1) {}
    int size(int x) { return -e[find(x)]; }
    int find(int x) { return e[x] < 0 ? x : find(e[x]); }
    int time() { return (int) st.size(); }
    void rollback(int t) {
        for (int i = time(); i-- > t;){
            e[st[i].first] = st[i].second;
        }
        st.resize(t);
    }
    bool join(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (e[a] > e[b]) swap(a, b);
        st.push_back({a, e[a]});
        st.push_back({b, e[b]});
        e[a] += e[b];
        e[b] = a;
        return true;
    }
};

int main(){
    // Ejemplo de uso
    RollbackUF UF(5);       // Creacion del DSU
    UF.join(0, 1);          // Union de los elementos 0 y 1
    cout<<UF.size(0)<<ENDL; // Ahora el tamanio del set del elemento 0 es 2
    UF.rollback(0);         // Regresar al tiempo 0
    cout<<UF.size(0)<<ENDL; // Ahora el tamanio del set del elemento 0 es 1 de nuevo, porque se
          deshizo el cambio
    return 0;
}
```

3.3 Monotonic stack

```
// Time complexity O(n)
typedef vector<int> vi;
```

```
int main(){
    int n = 6, arr[] = {7, 1, 4, 3, 5, 2};

    stack<int> st;
    vi nextGreater(n, -1);  // Para cada posicion se guarda cual es el siguiente elemento mayor

    for(int i=0; i<n; i++){
        while(!st.empty() && arr[i] > arr[st.top()]){   // Mientras la pila no este vacia y el i-esimo
                elemento sea mayor al top
            nextGreater[st.top()] = arr[i];             // El siguiente mayor del elemento en el top
                es el elemento en la i-esima posicion
            st.pop();                                   // Se saca el elemento del top
        }
        st.push(i);                                     // Se inserta la i-esima posicion en la pila
    }
    /*  Notas:
        - Para obtener los mayores previos, se hace un for reverso
        - Para obtener los menores, solo se invierte la segunda condicion en el ciclo while
    */
}
```

3.4 Binary indexed tree

```
int n, bit[MAXN];  // Utilizar a partir del 1

int query(int index) {
    int sum = 0;
    while (index > 0) {
        sum += bit[index];
        index -= index & (-index);
    }
    return sum;
}

void add(int index, int val) {
    while (index <= n) {
        bit[index] += val;
        index += index & (-index);
    }
}
```

3.5 Fenwick tree

```
#define LSOne(S) ((S) & -(S))                     // La operacion clave (Bit menos significativo)
typedef long long ll;
typedef vector<ll> vll;
typedef vector<int> vi;

class FenwickTree {                                 // El indice 0 no se usa
 private:
    vll ft;                                         // Internamente el FT es un vector
 public:
    FenwickTree(int m) { ft.assign(m+1, 0); }       // Crea un FT vacio

    void build(const vll &f) {
        int m = (int)f.size()-1;                    // Nota: f[0] siempre es 0
        ft.assign(m+1, 0);
        for (int i = 1; i <= m; ++i) {              // O(m)
            ft[i] += f[i];                          // Agrega este valor
            if (i+LSOne(i) <= m)                    // i tiene padre
                ft[i+LSOne(i)] += ft[i];            // Se agrega al padre
        }
    }

    FenwickTree(const vll &f) { build(f); }         // Crea un FT basado en f

    FenwickTree(int m, const vi &s) {               // Crea un FT basado en s
        vll f(m+1, 0);
        for (int i = 0; i < (int)s.size(); ++i)     // Se hace la conversion primero
            ++f[s[i]];                              // En O(n)
        build(f);                                   // En O(m)
    }

    ll rsq(int j) {                                 // returns RSQ(1, j)
        ll sum = 0;
        for (; j; j -= LSOne(j))
            sum += ft[j];
        return sum;
    }

    ll rsq(int i, int j) { return rsq(j) - rsq(i-1); }  // inc/exclusion

    // Actualiza el valor del i-esimo elemento por v (v+ = inc / v- = dec)
```

```
    void update(int i, ll v) {
        for (; i < (int)ft.size(); i += LSOne(i))
            ft[i] += v;
    }

    int select(ll k) {  // O(log m)
        int p = 1;
        while (p*2 < (int)ft.size()) p *= 2;
        int i = 0;
        while (p) {
            if (k > ft[i+p]) {
                k -= ft[i+p];
                i += p;
            }
            p /= 2;
        }
        return i+1;
    }
};

class RUPQ {            // Variante RUPQ
 private:
    FenwickTree ft;     // Internamente usa un FT PURQ
 public:
    RUPQ(int m) : ft(FenwickTree(m)) {}
    void range_update(int ui, int uj, ll v) {
        ft.update(ui, v);                           // [ui, ui+1, .., m] +v
        ft.update(uj+1, -v);                        // [uj+1, uj+2, .., m] -v
    }                                               // [ui, ui+1, .., uj] +v
    ll point_query(int i) { return ft.rsq(i); }     // rsq(i) es suficiente
};

class RURQ  {           // Variante RURQ
 private:               // Necesita dos FTs de ayuda
    RUPQ rupq;          // Un RUPQ y
    FenwickTree purq;   // un PURQ
 public:
    RURQ(int m) : rupq(RUPQ(m)), purq(FenwickTree(m)){} // Inicializacion
    void range_update(int ui, int uj, ll v) {
        rupq.range_update(ui, uj, v);                   // [ui, ui+1, .., uj] +v
        purq.update(ui, v*(ui-1));                      // -(ui-1)*v antes de ui
        purq.update(uj+1, -v*uj);                       // +(uj-ui+1)*v despues de uj
    }
    ll rsq(int j) {
        return rupq.point_query(j)*j - purq.rsq(j);     // Calculo optimista - factor de cancelacion
    }
    ll rsq(int i, int j) { return rsq(j) - rsq(i-1); }  // standard
};
```

3.6 Segment tree

```
/**
 * Descripcion: arbol de segmentos, bastante poderoso para
 * realizar consultas de rango y actualizaciones de punto,
 * se puede utilizar cualquier operacion conmutativa, es decir,
 * aquella en donde el orden de evaluacion no importe: suma,
 * multiplicacion, XOR, OR, AND, MIN, MAX, etc.
 * Tiempo: O(n log n) en construccion y O(log n) por consulta
 */

class SegmentTree {
 private:
    int n;
    vi arr, st;

    int l(int p) { return (p << 1) + 1; }
    int r(int p) { return (p << 1) + 2; }

    void build(int index, int start, int end) {
        if (start == end)
        st[index] = arr[start];
        else {
            int mid = (start + end) / 2;

            build(l(index), start, mid);
            build(r(index), mid + 1, end);

            st[index] = st[l(index)] + st[r(index)];
        }
    }

    int query(int index, int start, int end, int i, int j) {
        if (j < start || end < i)
            return 0;  // Si ese rango no nos sirve, retornar un valor que no cambie nada

        if (i <= start && end <= j)
            return st[index];

        int mid = (start + end) / 2;

        return query(l(index), start, mid, i, j) + query(r(index), mid + 1, end, i, j);
    }

   void update(int index, int start, int end, int idx, int val) {
        if (start == end)
            st[index] = val;
        else {
            int mid = (start + end) / 2;
            if (start <= idx && idx <= mid)
                update(l(index), start, mid, idx, val);
            else
                update(r(index), mid + 1, end, idx, val);

            st[index] = st[l(index)] + st[r(index)];
        }
   }

 public:
    SegmentTree(int sz) : n(sz), st(4 * n) {}

    SegmentTree(const vi &initialArr) : SegmentTree((int)initialArr.size()) {
        arr = initialArr;
        build(0, 0, n - 1);
    }

    void update(int i, int val) { update(0, 0, n - 1, i, val); }

    int query(int i, int j) { return query(0, 0, n - 1, i, j); }  // [i, j]
};
```

3.7 Segment tree with lazy propagation

```
/**
 * Descripcion: arbol de segmentos, bastante poderoso para
 * realizar consultas de suma en un rango y actualizaciones
 * de suma en un rango de manera eficiente. El metodo add
 * agrega x a todos los numeros en el rango [start, end].
 * Uso: LazySegmentTree ST(arr)
 * Tiempo: O(log n)
 */

class LazySegmentTree {
 private:
    int n;
    vi A, st, lazy;

    inline int l(int p) { return (p << 1) + 1; }  // ir al hijo izquierdo
    inline int r(int p) { return (p << 1) + 2; }  // ir al hijo derecho

    void build(int index, int start, int end) {
        if (start == end) {
            st[index] = A[start];
        } else {
            int mid = (start + end) / 2;
            build(l(index), start, mid);
            build(r(index), mid + 1, end);
            st[index] = st[l(index)] + st[r(index)];
        }
    }

    // Nota: Si se utiliza para el minimo o maximo de un rango no se le agrega el (end - start + 1)
    void propagate(int index, int start, int end) {
        if (lazy[index] != 0) {
            st[index] += (end - start + 1) * lazy[index];
            if (start != end) {
                lazy[l(index)] += lazy[index];
                lazy[r(index)] += lazy[index];
            }
            lazy[index] = 0;
        }
    }

    void add(int index, int start, int end, int i, int j, int x) {
        propagate(index, start, end);
        if ((end < i) || (start > j))
            return;

        if (start >= i && end <= j) {
            st[index] += (end - start + 1) * x;
            if (start != end) {
                lazy[l(index)] += x;
                lazy[r(index)] += x;
            }
            return;
```

```
        }
        int mid = (start + end) / 2;

        add(l(index), start, mid, i, j, x);
        add(r(index), mid + 1, end, i, j, x);

        st[index] = (st[l(index)] + st[r(index)]);
    }

    int query(int index, int start, int end, int i, int j) {
        propagate(index, start, end);
        if (end < i || start > j)
            return 0;
        if ((i <= start) && (end <= j))
            return st[index];

        int mid = (start + end) / 2;

        return query(l(index), start, mid, i, j) + query(r(index), mid + 1, end, i, j);
    }

 public:
    LazySegmentTree(int sz) : n(sz), st(4 * n), lazy(4 * n) {}

    LazySegmentTree(const vi &initialA) : LazySegmentTree((int)initialA.size()) {
        A = initialA;
        build(0, 0, n - 1);
    }
    // [i, j]
    void add(int i, int j, int val) { add(0, 0, n - 1, i, j, val); }  // [i, j]
    int query(int i, int j) { return query(0, 0, n - 1, i, j); }      // [i, j]
};
```

3.8 Segment tree RMQ with lazy propagation

```
/**
 * Descripcion: arbol de segmentos, bastante poderoso para
 * realizar consultas de min/max en un rango y actualizaciones
 * en un rango de manera eficiente.
 * Uso: LazyRMQ ST(arr)
 * Tiempo: O(log n)
 */

class LazyRMQ {
 private:
    int n;
    vi A, st, lazy;

    int l(int p) { return (p << 1) + 1; }
    int r(int p) { return (p << 1) + 2; }

    int conquer(int a, int b) {
        if (a == -1)
            return b;
        if (b == -1)
            return a;
        return min(a, b);
    }

    void build(int p, int L, int R) {
        if (L == R)
            st[p] = A[L];
        else {
            int m = (L + R) / 2;
            build(l(p), L, m);
            build(r(p), m + 1, R);
            st[p] = conquer(st[l(p)], st[r(p)]);
        }
    }

    void propagate(int p, int L, int R) {
        if (lazy[p] != -1) {
            st[p] = lazy[p];
            if (L != R)
                lazy[l(p)] = lazy[r(p)] = lazy[p];
            else
                A[L] = lazy[p];
            lazy[p] = -1;
        }
    }

    int query(int p, int L, int R, int i, int j) {
        propagate(p, L, R);
        if (i > j)
            return -1;
        if ((L >= i) && (R <= j))
            return st[p];
        int m = (L + R) / 2;
        return conquer(query(l(p), L, m, i, min(m, j)),
                       query(r(p), m + 1, R, max(i, m + 1), j));
    }

    void update(int p, int L, int R, int i, int j, int val) {
        propagate(p, L, R);
        if (i > j)
            return;
        if ((L >= i) && (R <= j)) {
            lazy[p] = val;
            propagate(p, L, R);
        } else {
            int m = (L + R) / 2;
            update(l(p), L, m, i, min(m, j), val);
            update(r(p), m + 1, R, max(i, m + 1), j, val);
            int lsubtree = (lazy[l(p)] != -1) ? lazy[l(p)] : st[l(p)];
            int rsubtree = (lazy[r(p)] != -1) ? lazy[r(p)] : st[r(p)];
            st[p] = (lsubtree <= rsubtree) ? st[l(p)] : st[r(p)];
        }
    }

 public:
    LazyRMQ(int sz) : n(sz), st(4 * n), lazy(4 * n, -1) {}

    LazyRMQ(const vi &initialA) : LazyRMQ((int)initialA.size()) {
        A = initialA;
        build(1, 0, n - 1);
    }

    void update(int i, int j, int val) { update(0, 0, n - 1, i, j, val); }  // [i, j]
    int query(int i, int j) { return query(0, 0, n - 1, i, j); }            // [i, j]
};
```

3.9 Sparse segment tree

```
/**
 * Descripcion: arbol de segmentos esparcido, es util cuando
 * el rango usado es bastante largo. Lo que cambia es que solo
 * se crean los nodos del arbol que se van utilizando, por lo
 * que se utilizan 2 punteros para los hijos de cada nodo.
 * Uso: node ST();
 * Complejidad: O(log n)
 */

const int SZ = 1 << 17;
template <class T>
struct node {
    T val = 0;
    node<T>* c[2];
    node() { c[0] = c[1] = NULL; }
    void upd(int ind, T v, int L = 0, int R = SZ - 1) {
        if (L == ind && R == ind) {
            val += v;
            return;
        }
        int M = (L + R) / 2;
        if (ind <= M) {
            if (!c[0])
                c[0] = new node();
            c[0]->upd(ind, v, L, M);
        } else {
            if (!c[1])
                c[1] = new node();
            c[1]->upd(ind, v, M + 1, R);
        }
        val = 0;
        for (int i = 0; i < 2; i++)
            if (c[i])
                val += c[i]->val;
    }
    T query(int lo, int hi, int L = 0, int R = SZ - 1) {  // [l, r]
        if (hi < L || R < lo) return 0;
        if (lo <= L && R <= hi) return val;
        int M = (L + R) / 2;
        T res = 0;
        if (c[0]) res += c[0]->query(lo, hi, L, M);
        if (c[1]) res += c[1]->query(lo, hi, M + 1, R);
        return res;
    }
};
```

3.10 Sparse lazy segment tree

```cpp
/**
 * Descripcion: arbol de segmentos esparcido, es util cuando el
 * rango usado es bastante largo, y que ademas haya operaciones
 * de rango.
 * Uso:
 * Inicializar el nodo 1 como la raiz -> segtree[1] = {0, 0, 1, 1e9}
 * utilizar los metodos update y query
 * Complejidad: O(log n)
 */

struct Node {
    int sum, lazy, tl, tr, l, r;
    Node() : sum(0), lazy(0), l(-1), r(-1) {}
};

const int MAXN = 123456;
Node segtree[64 * MAXN];
int cnt = 2;

void push_lazy(int node) {
    if (segtree[node].lazy) {
        segtree[node].sum = segtree[node].tr - segtree[node].tl + 1;
        int mid = (segtree[node].tl + segtree[node].tr) / 2;
        if (segtree[node].l == -1) {
            segtree[node].l = cnt++;
            segtree[segtree[node].l].tl = segtree[node].tl;
            segtree[segtree[node].l].tr = mid;
        }
        if (segtree[node].r == -1) {
            segtree[node].r = cnt++;
            segtree[segtree[node].r].tl = mid + 1;
            segtree[segtree[node].r].tr = segtree[node].tr;
        }
        segtree[segtree[node].l].lazy = segtree[segtree[node].r].lazy = 1;
        segtree[node].lazy = 0;
    }
}

void update(int node, int l, int r) {  // [l, r]
    push_lazy(node);
    if (l == segtree[node].tl && r == segtree[node].tr) {
        segtree[node].lazy = 1;
        push_lazy(node);
    } else {
        int mid = (segtree[node].tl + segtree[node].tr) / 2;
        if (segtree[node].l == -1) {
            segtree[node].l = cnt++;
            segtree[segtree[node].l].tl = segtree[node].tl;
            segtree[segtree[node].l].tr = mid;
        }
        if (segtree[node].r == -1) {
            segtree[node].r = cnt++;
            segtree[segtree[node].r].tl = mid + 1;
            segtree[segtree[node].r].tr = segtree[node].tr;
        }

        if (l > mid)
            update(segtree[node].r, l, r);
        else if (r <= mid)
            update(segtree[node].l, l, r);
        else {
            update(segtree[node].l, l, mid);
            update(segtree[node].r, mid + 1, r);
        }

        push_lazy(segtree[node].l);
        push_lazy(segtree[node].r);
        segtree[node].sum =
            segtree[segtree[node].l].sum + segtree[segtree[node].r].sum;
    }
}

int query(int node, int l, int r) {  // [l, r]
    push_lazy(node);
    if (l == segtree[node].tl && r == segtree[node].tr)
        return segtree[node].sum;
    else {
        int mid = (segtree[node].tl + segtree[node].tr) / 2;
        if (segtree[node].l == -1) {
            segtree[node].l = cnt++;
            segtree[segtree[node].l].tl = segtree[node].tl;
            segtree[segtree[node].l].tr = mid;
        }
        if (segtree[node].r == -1) {
            segtree[node].r = cnt++;
            segtree[segtree[node].r].tl = mid + 1;
            segtree[segtree[node].r].tr = segtree[node].tr;
        }

        if (l > mid)
            return query(segtree[node].r, l, r);
        else if (r <= mid)
            return query(segtree[node].l, l, r);
        else
            return query(segtree[node].l, l, mid) +
                query(segtree[node].r, mid + 1, r);
    }
}
```

3.11 Persistant lazy segment tree

```cpp
/**
 * Descripcion: arbol de segmentos persistente que
 * permite consultas de rango de manera eficiente.
 * Una estructura persistente es aquella que guarda
 * sus estados anteriores y puede volver a ellos.
 * Tiempo: O(log n) por consulta
 */

ll arr[MAXN], l[45 * MAXN], r[45 * MAXN], st[45 * MAXN], nodes = 0;
bool hasFlag[45 * MAXN];
ll flag[45 * MAXN], root[MAXN];

ll newLeaf(ll value) {
    ll p = ++nodes;
    l[p] = r[p] = 0;  // Nodo sin hijos
    st[p] = value;
    return p;
}

ll newParent(ll left, ll right) {
    ll p = ++nodes;
    l[p] = left;
    r[p] = right;
    st[p] = st[left] + st[right];
    return p;
}

ll newLazyKid(ll node, ll x, ll left, ll right) {
    ll p = ++nodes;
    l[p] = l[node];
    r[p] = r[node];
    flag[p] = flag[node];
    hasFlag[p] = true;

    flag[p] = x;
    st[p] = (right - left + 1) * x;  // <-- Si quieres cambiar todo el segmento por x
    // st[p] = st[node]+(right-left+1)*x <-- Si se quiere suma x a todo el segmento

    return p;
}

ll build(ll left, ll right) {
    if (left == right)
        return newLeaf(arr[left]);
    else {
        ll mid = (left + right) / 2;
        return newParent(build(left, mid), build(mid + 1, right));
    }
}

void propagate(ll p, ll left, ll right) {
    if (hasFlag[p]) {
        if (left != right) {
            ll mid = (left + right) / 2;
            l[p] = newLazyKid(l[p], flag[p], left, mid);
            r[p] = newLazyKid(r[p], flag[p], mid + 1, right);
        }
        hasFlag[p] = false;
    }
}

ll update(ll a, ll b, ll x, ll p, ll left, ll right) {
    if (b < left | right < a)
        return p;
    if (a <= left && right <= b)
        return newLazyKid(p, x, left, right);
    propagate(p, left, right);
    ll mid = (left + right) / 2;
    return newParent(update(a, b, x, l[p], left, mid),
                     update(a, b, x, r[p], mid + 1, right));
}

ll query(ll a, ll b, ll p, ll left, ll right) {
    if (b < left || right < a)
        return 0;
    if (a <= left && right <= b)
        return st[p];
```

```cpp
    ll mid = (left + right) / 2;
    propagate(p, left, right);
    return (query(a, b, l[p], left, mid) + query(a, b, r[p], mid + 1, right));
}

// revert range [a:b] of p
int rangecopy(int a, int b, int p, int revert, int L = 0, int R = n - 1) {
    if (b < L || R < a) return p;         // keep version
    if (a <= L && R <= b) return revert;  // reverted version
    return newparent(rangecopy(a, b, l[p], l[revert], L, M),
                     rangecopy(a, b, r[p], r[revert], M + 1, R));
}

// Usage: (revert a range [a:b] back to an old version)
// int reverted_root = rangecopy(a, b, root, old_version_root);
```

3.12 Iterative segment tree

```cpp
/**
 * Descripcion: arbol de segmentos, bastante poderoso para
 * realizar consultas de rango y actualizaciones de punto,
 * se puede utilizar cualquier operacion conmutativa, es decir,
 * aquella en donde el orden de evaluacion no importe: suma,
 * multiplicacion, XOR, OR, AND, MIN, MAX, etc.
 * Tiempo: O(n log n) en construccion y O(log n) por consulta
 */
template <class T>
class SegmentTree {
 private:
    const T DEFAULT = 1e18;  // Causa overflow si T es int

    vector<T> ST;
    int len;

 public:
    SegmentTree(int len) : len(len), ST(len * 2, DEFAULT) {}
    SegmentTree(vector<T>& v) : SegmentTree(v.size()) {
        for (int i = 0; i < len; i++)
            set(i, v[i]);
    }

    void set(int ind, T val) {
        ind += len;
        ST[ind] = val;
        for (; ind > 1; ind /= 2)
            ST[ind / 2] = min(ST[ind], ST[ind ^ 1]);  // Operacion
    }

    // [start, end]
    T query(int start, int end) {
        end++;
        T ans = DEFAULT;
        for (start += len, end += len; start < end; start /= 2, end /= 2) {
            if (start % 2 == 1) {
                ans = min(ans, ST[start++]);
            }  // Operacion
            if (end % 2 == 1) {
                ans = min(ans, ST[--end]);
            }  // Operacion
        }
        return ans;
    }
};
```

3.13 Segment tree of DSU with rollback

```cpp
// Union find rollback y segment tree para poder responder queries del numero de componentes que hay
     en cada instante de tiempo
typedef vector<int> vi;

struct dsu_save {   // Struct de los datos de cada set
    int v, rnkv, u, rnku;
    dsu_save(int _v, int _rnkv, int _u, int _rnku) : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};

struct dsu_with_rollbacks { // Dsu con rollback
    vi p, rnk;              // Vectores de padres y rangos
    int comps;              // Numero de componentes
    stack<dsu_save> op;

    dsu_with_rollbacks(int n) {     // Constructor donde n es el numero inicial de sets
        p.resize(n), rnk.resize(n);
        for (int i = 0; i < n; i++) {
            p[i] = i;
            rnk[i] = 0;
        }
        comps = n;
    }

    int find_set(int v) { return (v == p[v]) ? v : find_set(p[v]); }    // Regresa si estan en el
         mismo set

    bool unite(int v, int u) {  // Une 2 sets
        v = find_set(v), u = find_set(u);
        if (v == u) return false;
        comps--;
        if (rnk[v] > rnk[u])    swap(v, u);
        op.push(dsu_save(v, rnk[v], u, rnk[u]));
        p[v] = u;
        if (rnk[u] == rnk[v])   rnk[u]++;
        return true;
    }

    void rollback() {   // Revierte la ultima union hecha
        if (op.empty()) return;
        dsu_save x = op.top(); op.pop();
        comps++;
        p[x.v] = x.v, rnk[x.v] = x.rnkv;
        p[x.u] = x.u, rnk[x.u] = x.rnku;
    }
};

struct query {      // Struct para las queries
    int v, u;       // v= primer elemento, u= segundo elemento
    bool united;    // Para saber si estan unidos
    query(int _v, int _u) : v(_v), u(_u) { }
};

// Time complexity build O(T(n)), delete (T(n) log n). T(n)= time
struct QueryTree {              // Struct de un segment tree para resolver las queries
    vector<vector<query>> t;    // Vector para almacenar las queries
    dsu_with_rollbacks dsu;     // DSU
    int T;                      // Tiempo

    QueryTree(int _T, int n) : T(_T) {  // Constructor donde _T es el rango de tiempo y n es el numero
         inicial de sets
        dsu = dsu_with_rollbacks(n);
        t.resize(4 * T + 4);
    }

    void add_to_tree(int v, int l, int r, int ul, int ur, query& q) {   // Metodo para agregar una
         query al tree
        if (ul > ur)
            return;
        if (l == ul && r == ur) {
            t[v].push_back(q);
            return;
        }
        int mid = (l + r) / 2;
        add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
        add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
    }

    // Las queries se agregan de la manera UF.add_query(query(v, u), l, r)
    // Donde v y u son los elementos a unir, mientras que l y r representan el rango de tiempo en el
         que estan unidos
    void add_query(query q, int l, int r) {
        add_to_tree(1, 0, T - 1, l, r, q);
    }

    void dfs(int v, int l, int r, vi& ans) {    // DFS para recorrer las queries
        for (query& q : t[v])
            q.united = dsu.unite(q.v, q.u);
        if (l == r)
            ans[l] = dsu.comps;
        else {
            int mid = (l + r) / 2;
            dfs(2 * v, l, mid, ans);
            dfs(2 * v + 1, mid + 1, r, ans);
        }
        for (query q : t[v])
            if (q.united)
                dsu.rollback();
    }

    vi solve() {   // Retorna un vector con el numero de componentes en cada instante de tiempo
        vi ans(T);
        dfs(1, 0, T - 1, ans);
        return ans;
    }
};

int main(){
    // Ejemplo de uso
    QueryTree UF(5,5);                  // Se crea el segment tree para resolver las queries
```

```
    UF.add_query(query(0, 1), 2, 3);    // Se agrega una querie indicando que los elementos v=0 y u=1
         estan unidos desde t=2 hasta t=3
    UF.add_query(query(2, 3), 1, 4);    // Se agrega una querie indicando que los elementos v=2 y u=3
         estan unidos desde t=1 hasta t=4
    vi res = UF.solve();                // Se llama el metodo para resolver las queries
    for(auto u : res)
        cout<<u<<" ";  // Se imprime 5 4 3 3 4, representando el numero de disjoint sets en cada
             instante de tiempo
    return 0;
}
```

3.14 Sparse table

```
// Time complexity: Build O(n log n), Query O(1)
typedef vector<int> vi;
inline int L2(int x) { return 31-__builtin_clz(x); }
inline int P2(int x) { return 1<<x; }

struct SparseTable {
    vi A;           // Vector inicial
    vector<vi> SpT; // Sparse table

    SparseTable() {}
    SparseTable(vi &initialA) : A(initialA){        // O(n log n)
        int n = (int)A.size(), L2_n = L2(n)+1;
        SpT = vector<vi>(L2(n)+1, vi(n));           // Inicializacion
        for (int j = 0; j < n; ++j)
            SpT[0][j] = j;                          // RMQ del sub array [j..j]
        for (int i = 1; P2(i) <= n; ++i)            // Para toda i s.t. 2^i <= n
            for (int j = 0; j+P2(i)-1 < n; ++j) {   // Para toda j valida
                int x = SpT[i-1][j];                // [j..j+2^(i-1)-1]
                int y = SpT[i-1][j+P2(i-1)];        // [j+2^(i-1)..j+2^i-1]
                SpT[i][j] = A[x] <= A[y] ? x : y;   // Guarda el indice del elemento menor
            }
    }

    int RMQ(int i, int j) {                 // O(1)
        int k = L2(j-i+1);                  // 2^k <= (j-i+1)
        int x = SpT[k][i];                  // Cubre [i..i+2^k-1]
        int y = SpT[k][j-P2(k)+1];          // Cubre [j-2^k+1..j]
        return A[x] <= A[y] ? x : y;        // Retorna el indice del elemento menor
    }
};
```

3.15 Order statistics tree

```
// Time complexity Insertion O(n log n), select-rank O(log n)
#include <bits/extc++.h>
#include <bits/stdc++.h>

using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> ost;
/*(Posiciones indexadas en 0).
Funciona igual que un set (todas las operaciones en O(log n)), con 2 operaciones extra:
obj.find_by_order(k) - Retorna un iterador apuntando al elemento k-esimo mas grande
obj.order_of_key(x) - Retorna un entero que indica la cantidad de elementos menores a x

Modificar unicamente primer y tercer parametro, que corresponden a el tipo de dato
del ost y a la funcion de comparacion de valores (less<T>, greater<T>, less_equal<T>
o incluso una implementada por nosotros)

Si queremos elementos repetidos, usar less_equal<T> (sin embargo, ya no servira la
funcion de eliminacion).

Si queremos elementos repetidos y necesitamos la eliminacion, utilizar una
tecnica con pares, donde el second es un numero unico para cada valor.
*/

// Implementacion
int main(){
    int n = 9;
    int A[] = {2, 4, 7, 10, 15, 23, 50, 65, 71};    // Arreglo de elementos
    ost tree;
    for (int i = 0; i < n; ++i) // O(n log n)
        tree.insert(A[i]);
    // O(log n) select
    cout << *tree.find_by_order(0) << "\n";     // 1-smallest = 2
    cout << *tree.find_by_order(n - 1) << "\n"; // 9-smallest/largest = 71
    cout << *tree.find_by_order(4) << "\n";     // 5-smallest = 15
    // O(log n) rank
    cout << tree.order_of_key(2) << "\n";  // index 0 (rank 1)
    cout << tree.order_of_key(71) << "\n"; // index 8 (rank 9)
    cout << tree.order_of_key(15) << "\n"; // index 4 (rank 5)
}
```

3.16 Binary search tree

```
// Implementacion de un BST con recorridos pre, in y post orden
class BST{
    int data;
    BST *left,*right;

    public:
        BST();
        BST(int);
        BST* insert(BST*,int);
        BST* deleteNode(BST*,int);
        void preorder(BST*);
        void inorder(BST*);
        void postorder(BST*);
        void printLeafNodes(BST*);
};

BST::BST(){
    data=0;
    left=right=NULL;
}

BST::BST(int value){
    data=value;
    left=right=NULL;
}

BST* BST::insert(BST* root, int value){
    if(!root){
        return new BST(value);
    }
    if(value>=root->data){
        root->right=insert(root->right,value);
    }
    else if(value<root->data){
        root->left=insert(root->left,value);
    }
    return root;
}

BST* BST::deleteNode(BST* root, int k)
{
    if (root == NULL)
        return root;

    if (root->data > k) {
        root->left = deleteNode(root->left, k);
        return root;
    }
    else if (root->data < k) {
        root->right = deleteNode(root->right, k);
        return root;
    }

    if (root->left == NULL) {
        BST* temp = root->right;
        delete root;
        return temp;
    }
    else if (root->right == NULL) {
        BST* temp = root->left;
        delete root;
        return temp;
    }

    else {

        BST* succParent = root;

        BST* succ = root->right;
        while (succ->left != NULL) {
            succParent = succ;
            succ = succ->left;
        }

        if (succParent != root)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;

        root->data = succ->data;
```

```
        delete succ;
        return root;
    }
}

void BST::preorder(BST* root){
    if(!root){
        return;
    }
    cout<<root->data<<" ";
    preorder(root->left);
    preorder(root->right);
}

void BST::inorder(BST* root){
    if(!root){
        return;
    }
    inorder(root->left);
    cout<<root->data<<" ";
    inorder(root->right);
}

void BST::postorder(BST* root){
    if(!root){
        return;
    }
    postorder(root->left);
    postorder(root->right);
    cout<<root->data<<" ";
}

void BST::printLeafNodes(BST* root){
    if (!root)
        return;
    if (!root->left && !root->right){
        cout << root->data << " ";
        return;
    }
    if (root->left)
       printLeafNodes(root->left);
    if (root->right)
       printLeafNodes(root->right);
}

int main(){
    int n,num;
    cin>>n;
    BST b,*root=NULL;
    for(int i=0;i<n;i++){
        cin>>num;
        if(i==0){
            root=b.insert(root,num);
            continue;
        }
        b.insert(root,num);
    }
    b.preorder(root);
    cout<<endl;
    b.inorder(root);
    cout<<endl;
    b.postorder(root);
    cout<<endl;
    return 0;
}
```

4 Graphs

4.1 Graph traversal

```
// Time complexity O(V + E)
// Source: Own work
typedef vector<int> vi;

// DFS
vector<vi> adj;
vector<bool> visited;

void dfs(int u){
    if(visited[u]) return;
    visited[u]=true;
    //process node
    for(auto &v : adj[u])
        dfs(v);
}
```

```
// BFS
const int MAXN = 1e6;

void bfs(int src){
    queue<int> q; q.push(src);
    vector<bool> visited(MAXN, false); visited[src] = true;

    while(!q.empty()){
        int u = q.front(); q.pop();
        // Process node
        for(auto &v : adj[u]){
            if(visited[v])  continue;
            visited[v] = true;
            q.push(v);
        }
    }
}

// Bipartite graph check
bool bfs(int src){
    queue<int> q; q.push(src);
    vi color(MAXN, -1); color[src] = 0;

    while(!q.empty()){
        int u = q.front(); q.pop();
        for(auto &v : adj[u]){
            if(color[v] == -1){
                color[v] = color[u] ^ 1;
                q.push(v);
            }
            else if(color[v] == color[u])
                return false;
        }
    }

    return true;
}
```

4.2 Dijkstra

```
// Time complexity O(V + E log V)
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
const int INF = 1e9;

vector<vii> adj;

vi dijkstra(int source, int V){
    priority_queue<ii, vii, greater<ii>> pq;
    pq.push({0, source});
    vi dist(V, INF);
    dist[source] = 0;

    while(!pq.empty()){
        auto [d, u] = pq.top(); pq.pop();
        if(d > dist[u]) continue;
        for(auto &[v, w] : adj[u]){
            if(d+w >= dist[v])  continue;
            dist[v] = d+w;
            pq.push({dist[v], v});
        }
    }

    return dist;
}
```

4.3 Bellman-Ford

```
// Time complexity O(V*E)

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
const int INF = 1e9;

int main() {
    // Numero de nodos(V), numero de aristas(E), nodo inicio(s)
    vector<vii> AL(V, vii());

    // Ruta del Bellman Ford, basicamente relaja las E aristas V-1 veces
    vi dist(V, INF); dist[s] = 0;                  // Inicializacion en distancias infinitas
```

```
    for (int i = 0; i < V-1; ++i) {                 // total O(V*E)
        bool modified = false;                      // Optimizacion
        for (int u = 0; u < V; ++u)                 // Estos 2 ciclos = O(E)
        if (dist[u] != INF)                         // Verificacion importante
            for (auto &[v, w] : AL[u]) {
                if (dist[u]+w >= dist[v]) continue; // No hay mejora, saltar
                dist[v] = dist[u]+w;                // Operacion de relajacion
                modified = true;                    // Optimizacion
            }
        if (!modified) break;                       // Optimizacion
    }

    bool hasNegativeCycle = false;
    for (int u = 0; u < V; ++u)                     // Una pasada mas para verificar
        if (dist[u] != INF)
            for (auto &[v, w] : AL[u])
                if (dist[v] > dist[u]+w)            // Debe ser falso
                    hasNegativeCycle = true;        // Si true => Existe ciclo negativo
    printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

    if (!hasNegativeCycle)
        for (int u = 0; u < V; ++u)
            printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);

    return 0;
}
```

4.4 Floyd-Warshal

```
// Time complexity O(V^3)
const int INF = 1e9;
const int MAX_V = 450;  // Si |V| > 450, no se puede usar el Floyd-Warshall
int AM[MAX_V][MAX_V];   // Es mejor guardar un arreglo grande en el heap
int P[MAX_V][MAX_V];     // Arreglo para guardar el camino (Solo si es necesario)

void printPath(int i, int j) {
    if (i != j) printPath(i, P[i][j]);
    printf(" %d", v);
}

int main() {
    // Numero de nodos(V), numero de aristas(E)
    // Inicializar con AM[u][v] = INF, AM[u][u] = 0

    // Rutina del Floyd-Warshall

    for (int i = 0; i < V; ++i)
        for (int i = 0; j < V; ++j)
            p[i][j] = i; // Inicializacion del arreglo del camino

    for (int k = 0; k < V; ++k)                     // El orden del ciclo es k->u->v
        for (int u = 0; u < V; ++u)
            for (int v = 0; v < V; ++v){
                if (AM[u][k]+AM[k][v] < AM[u][v])   // Solo si se necesita imprimir el camino
                    P[u][v] = P[k][v];              // Se actualiza el camino
                AM[u][v] = min(AM[u][v], AM[u][k]+AM[k][v]);
            }

    for (int u = 0; u < V; ++u)
        for (int v = 0; v < V; ++v)
        printf("APSP(%d, %d) = %d\n", u, v, AM[u][v]);

    return 0;
}
```

4.5 Topological sort

```
// Time complexity O(V+E)
typedef vector<int> vi;

vector<vi> AL;
vector<bool> visited;
vi ts;

void toposort(int u) {
    visited[u] = 1;
    for (auto &v : AL[u])
        if (!visited[v])
            toposort(v);
    ts.push_back(u);    // Este es el unico cambio con respecto a un DFS
}

int main() {
    // El grafo tiene que ser DAG
    // Numero de nodos(V), numero de aristas(E)
    AL.assign(V, vi());

    visited.assign(V, 0);
    ts.clear();
    for (int u = 0; u < V; ++u)                     // Igual que para encontrar los CCs
        if (!visited[u])
            toposort(u);

    printf("Topological sort: \n");
    reverse(ts.begin(), ts.end());                  // Invertir ts o imprimir al reves
    for (auto &u : ts)
        printf(" %d", u);
    printf("\n");

    return 0;
}
```

4.6 Lexicographic topological sort

```
// Time complexity O(V+E)
typedef vector<int> vi;

int V, E;           // Numero de nodos y aristas
vector<vi> AL;      // Lista de adyacencia
vi in_degree;       // Grado de entrada de cada nodo
vi sorted_nodes;    // Nodos ordenados

void topo_sort() {
    priority_queue<int, vector<int>, greater<int>> q;

    for(int i=0; i<V; i++)
        if (in_degree[i] == 0)
            q.push(i);

    while (!q.empty()) {
        int u = q.top();
        q.pop();
        sorted_nodes.push_back(u);
        for (int v : AL[u]) {
            in_degree[v]--;
            if (in_degree[v] == 0)
                q.push(v);
        }
    }
}

int main() {
    // Numero de nodos(V), numero de aristas(E)
    AL.assign(V, vi());
    in_degree.assign(V, 0);

    // Leer el grafo e incrementar los grados de entrada en cada nodo

    topo_sort();

    if (sorted_nodes.size() < V) {
        cout << "El grafo tiene un ciclo" << '\n';
    } else {
        cout << "Orden topologico lexicograficamente menor: ";
        for (int u : sorted_nodes)
            cout << u << " ";
    }

    return 0;
}
```

4.7 Prim

```
// Time complexity O(E log E)
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

vector<vii> AL;
vi taken;
priority_queue<ii, vector<ii>, greater<ii>> pq;
int mst_cost = 0, num_taken = 0;

void process(int u){
    taken[u] = 1;
    for (auto &[v, w] : AL[u])
```

```
        if (!taken[v])
            pq.push({w, v});
}

void prim(vector<vii> AL, int src, int V){
    taken.assign(V+1, 0);
    process(src);
    while (!pq.empty()){
        auto [w, u] = pq.top();
        pq.pop();
        if (taken[u])
            continue;
        mst_cost += w;
        process(u);
        ++num_taken;
        if (num_taken == V - 1)
            break;
    }
}

int main(){
    int V, E;
    cin>>V>>E;
    AL.assign(V+1, vii());
    for (int i = 0; i < E; ++i){
        int u, v, w;
        cin>>u>>v>>w;
        AL[u].push_back({v,w});
        AL[v].push_back({u,w});
    }
    prim(AL,1,V);
    cout<<"MST cost= "<<mst_cost;
    return 0;
}
```

4.8 Kruskal

```
// Time complexity O(E log E)
typedef long long ll;
typedef vector<int> vi;
typedef tuple<int,int,int> iii;

// Union find utilizado para formar el MST
class UnionFind {
private:
    vi p, rank, setSize;
    int numSets;
public:
    UnionFind(int N) {
        p.assign(N, 0); for (int i = 0; i < N; ++i) p[i] = i;
        rank.assign(N, 0);
        setSize.assign(N, 1);
        numSets = N;
    }
    int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
    bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
    void unionSet(int i, int j) {
        if (isSameSet(i, j)) return;
        int x = findSet(i), y = findSet(j);
        if (rank[x] > rank[y]) swap(x, y);
        p[x] = y;
        if (rank[x] == rank[y]) ++rank[y];
        setSize[y] += setSize[x];
        --numSets;
    }
    int numDisjointSets() { return numSets; }
    int sizeOfSet(int i) { return setSize[findSet(i)]; }
};

int main() {
    int V, E;
    cin>>V>>E;
    vector<iii> EL(E);
    for (int i = 0; i < E; ++i) {
        int u, v, w;
        cin>>u>>v>>w;
        EL[i] = {w, u, v};
    }
    sort(EL.begin(), EL.end());
    ll mst_cost = 0, num_taken = 0;
    UnionFind UF(V+1);
    for (int i = 0; i < E; ++i) {
        auto [w, u, v] = EL[i];
        if (UF.isSameSet(u, v)) continue;
        mst_cost += w;
        UF.unionSet(u, v);
        ++num_taken;
        if (num_taken == V-1) break;
    }
    cout<<mst_cost<<" "<<num_taken;

  return 0;
}
```

4.9 Tarjan

```
// Time complexity O(V + E)
typedef vector<int> vi;

int dfsNumberCounter, numSCC;                        // Variables globales
vector<vi> AL;
vi dfs_num, dfs_low, visited;
stack<int> St;

void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter;      // dfs_low[u]<=dfs_num[u]
    dfsNumberCounter++;                              // Incrementa el contador
    St.push(u);                                      // Para recordar el orden
    visited[u] = 1;

    for (auto v : AL[u]) {
        if (dfs_num[v] == -1)                        // No visitado
            tarjanSCC(v);
        if (visited[v])                              // Condicion de actualizacion
            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
    }

    if (dfs_low[u] == dfs_num[u]) {                  // Raiz o inicio de un SCC
        ++numSCC;                                    // Se aumenta el numero de SCC
        while (1) {
            int v = St.top(); St.pop(); visited[v] = 0;
            if (u == v) break;
        }
    }
}

int main() {
    // Num_Nodos (V), Num_Aristas (E)
    AL.assign(V,vi());
    // Lectura del grafo (Dirigido)

    // Ejecucion del algoritmo de Tarjan
    dfs_num.assign(V, -1); dfs_low.assign(V, 0); visited.assign(V, 0);
    while (!St.empty()) St.pop();
    dfsNumberCounter = numSCC = 0;

    for (int u = 0; u < V; ++u)
        if (dfs_num[u] == -1)   // No visitado
            tarjanSCC(u);

    // Imprime cuantos SCC tiene el grafo
    printf("Number of SCC: %d\n", numSCC);

    return 0;
}
```

4.10 Kosaraju

```
/*
Descripcion: Busqueda de componentes fuertemente conexos (Grafo dirigido) - Kosaraju O(V + E)
Un SCC se define de la siguiente manera: si elegimos cualquier par de vertices u y v
en el SCC, podemos encontrar un camino de u a v y viceversa

El algoritmo de Kosaraju realiza dos pasadas DFS, la primera para almacenar el orden
de finalizacion decreciente (orden topologico) y la segunda se realiza en un grafo
transpuesto a partir del orden topologico para hallar los SCC

Source: CPH4 Steven Halim
*/

vi graph[MAXN], graph_T[MAXN], dfs_num, S;
int n, numSCC;

void Kosaraju(int u, int pass) { //pass = 1 (original), 2 (transpose)
    dfs_num[u] = 1;
    vi &neighbor = (pass == 1) ? graph[u] : graph_T[u];
    for (auto v : neighbor) {
        if (dfs_num[v] == -1)
            Kosaraju(v, pass);
    }
    S.push_back(u);
}
```

```cpp
int main() {
    S.clear();
    dfs_num.assign(n, -1); // First pass - visited(-1)

    FOR(u, n) { //Record post order of original Graph
        if (dfs_num[u] == -1)
            Kosaraju(u, 1);
    }

    dfs_num.assign(n, -1);
    numSCC = 0;

    FORR(i, n, 1) {  // Finding SCC from transpose Graph
        if (dfs_num[S[i]] == -1) {
            ++numSCC;
            Kosaraju(S[i], 2);
        }
    }

    cout << numSCC << ENDL;
}
```

4.11 Bridges and articulation points

```cpp
// Time complexity O(V+E)
// Source: CPH4 Steven Halim
typedef vector<int> vi;

vector<vi> AL;
vi dfs_num, dfs_low, dfs_parent;
vector<bool> articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u){
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++;              // dfs_low[u]<=dfs_num[u]
    for (auto v : AL[u]){
        if (dfs_num[v] == -1){                                 // a tree edge, no visitado
            dfs_parent[v] = u;
            if (u == dfsRoot)
                ++rootChildren;                                // Caso especial, raiz
            articulationPointAndBridge(v);
            if (dfs_low[v] >= dfs_num[u])                      // Es un punto de articulacion
                articulation_vertex[u] = 1;
            if (dfs_low[v] > dfs_num[u])                       // Es un puente
                printf(" Edge (%d, %d) is a bridge\n", u, v);
            dfs_low[u] = min(dfs_low[u], dfs_low[v]);          // Actualizacion
        }
        else if (v != dfs_parent[u])                           // Evitar ciclo trivial
            dfs_low[u] = min(dfs_low[u], dfs_num[v]);          // Actualizacion
    }
}

int main(){
    // Num_Nodos (V), Num_Aristas (E)
    AL.assign(V, vi());
    // Lectura del grafo (NO dirigido)

    dfs_num.assign(V, -1), dfs_low.assign(V, 0), dfs_parent.assign(V, -1), articulation_vertex.assign(
          V, 0);
    dfsNumberCounter = 0;

    printf("Bridges:\n");
    for (int u = 0; u < V; ++u)
        if (dfs_num[u] == -1){  // No visitado
            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);

    return 0;
}
```

4.12 Lowest common ancestor

```cpp
// Time complexity Preprocessing = O(n log n), Query = (log n)
typedef vector<int> vi;

int l;          // Logaritmo base 2 del numero de nodos del arbol, redondeado hacia arriba
vector<vi> adj; // Lista de adyacencia para representar el arbol
int timer;      // Tiempo en el que se visita cada nodo
vi tin, tout;   // Arreglos de tiempos de entrada y salida de cada nodo
vector<vi> up;  // Vector de los ancestros de cada nodo, donde up[i][j] es el ancestro 2^j del nodo i

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;   // Si u es ancestro de v LCA(u,v)=u
    if (is_ancestor(v, u))  return v;   // Si v es ancestro de u LCA(u,v)=v

    for (int i = l; i >= 0; --i) {      // Se recorren los ancestros con saltos binarios
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    }

    return up[u][0];                    // Se retorna el LCA
}

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

4.13 Dinic

```cpp
/*
    Time complexity: O(V * E * log U), donde U = max|cap|.
    O(min(E^(1/2), V^(2/3) * E)) si U  = 1; O(sqrt(V)*E) para bipartite matching.
    Source: KACTL
*/
#define sz(x)   (int) x.size()
typedef vector<int> vi;
typedef long long ll;

struct Dinic {
    struct Edge {
        int to, rev;
        ll c, oc;
        ll flow() { return max(oc - c, 0LL); } // Si se necesitan los flujos
    };
    vi lvl, ptr, q;
    vector<vector<Edge>> adj;
    Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
    void addEdge(int a, int b, ll c, ll rcap = 0) {
        adj[a].push_back({b, sz(adj[b]), c, c});
        adj[b].push_back({a, sz(adj[a]) - 1, rcap, rcap});
    }
    ll dfs(int v, int t, ll f) {
        if (v == t || !f) return f;
        for (int& i = ptr[v]; i < sz(adj[v]); i++) {
            Edge& e = adj[v][i];
            if (lvl[e.to] == lvl[v] + 1)
                if (ll p = dfs(e.to, t, min(f, e.c))) {
                    e.c -= p, adj[e.to][e.rev].c += p;
                    return p;
                }
        }
        return 0;
    }
    ll calc(int s, int t) {
        ll flow = 0; q[0] = s;
        for(int L = 0; L < 31; L++) do { // 'int L=30' tal vez mas rapido para datos random
            lvl = ptr = vi(sz(q));
            int qi = 0, qe = lvl[s] = 1;
            while (qi < qe && !lvl[t]) {
                int v = q[qi++];
```

```
            for (Edge e : adj[v])
                if (!lvl[e.to] && e.c >> (30 - L))
                    q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
        }
        while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
    } while (lvl[t]);
    return flow;
  }
  bool leftOfMinCut(int a) { return lvl[a] != 0; }
};
```

4.14 Max-flow (Dinic)

```
// Time complexity O(V^2 * E)

typedef long long ll;
typedef vector<int> vi;
typedef pair<int, int> ii;
typedef tuple<int, ll, ll> edge;

const ll INF = 1e18;        // Suficientemente grande

class max_flow {
 private:
   int V;               // Numero de vertices
   vector<edge> EL;     // Lista de aristas
   vector<vi> AL;       // Lista de adyacencia con los indices de las aristas
   vi d, last;          // Vector de distancias y ultimas aristas
   vector<ii> p;        // Vector para el camino. first = id del nodo, second = indice en la lista de
        aristas

   bool BFS(int s, int t) {                          // Encontrar un augmenting path
       d.assign(V, -1); d[s] = 0;
       queue<int> q({s});
       p.assign(V, {-1, -1});                        // Guardar el sp tree del BFS
               while (!q.empty()) {
       int u = q.front(); q.pop();
           if (u == t) break;                        // Parar si se llega al sink t
           for (auto &idx : AL[u]) {                 // Explora los vecinos de u
               auto &[v, cap, flow] = EL[idx];       // Arista guardada en EL[idx]
               if ((cap-flow > 0) && (d[v] == -1))   // Arista residual positiva
               d[v] = d[u]+1, q.push(v), p[v] = {u, idx};  // 3 lineas en una B)
           }
       }
       return d[t] != -1;                            // Tiene un augmenting path
   }

   ll DFS(int u, int t, ll f = INF) {                // Ir de s->t
       if ((u == t) || (f == 0)) return f;
       for (int &i = last[u]; i < (int)AL[u].size(); ++i){ // Desde la ultima arista
           auto &[v, cap, flow] = EL[AL[u][i]];
           if (d[v] != d[u]+1) continue;             // No es parte del grafo de niveles
           if (ll pushed = DFS(v, t, min(f, cap-flow))) {
               flow += pushed;
               auto &rflow = get<2>(EL[AL[u][i]^1]);       // Arista de regreso
               rflow -= pushed;
               return pushed;
           }
       }
       return 0;
   }

 public:
   max_flow(int initialV) : V(initialV) {
       EL.clear();
       AL.assign(V, vi());
   }

   // Si se agrega una arista bidireccional u<->v con peso w en el grafo de flujo,
   // asigna directed = false. El valor por defecto es true (Arista dirigida)
   void add_edge(int u, int v, ll w, bool directed = true) {
       if (u == v) return;                          // Por seguridad: Evita ciclos en el mismo nodo
       EL.emplace_back(v, w, 0);                    // u->v, cap w, flow 0
       AL[u].push_back(EL.size()-1);                // Para recordar el indice
       EL.emplace_back(u, directed ? 0 : w, 0);     // Arista de regreso
       AL[v].push_back(EL.size()-1);                // Para recordar el indice
   }

   ll dinic(int s, int t) {
       ll mf = 0;                        // mf = Max flow
       while (BFS(s, t)) {               // Time complexity O(V^2*E)
           last.assign(V, 0);            // Aceleracion importante
           while (ll f = DFS(s, t))      // exhaust blocking flow
               mf += f;
       }
       return mf;
   }
};

int main() {
    // Leer numero de nodos(V), source(s), sink(t)
    // De preferencia asignar s = 0, t = V-1
    // max_flow mf(V);
    // Crear aristas usando el metodo add_edge(u, v, w);

  return 0;
}
```

4.15 Min-cost max-flow

```
// Time complexity O(V^2 * E^2)
typedef long long ll;
typedef tuple<int, ll, ll, ll> edge;
typedef vector<int> vi;
typedef vector<ll> vll;
const ll INF = 1e18;

class min_cost_max_flow {
 private:
    int V;
    ll total_cost;
    vector<edge> EL;
    vector<vi> AL;
    vll d;
    vi last, vis;

    bool SPFA(int s, int t) {   // SPFA para encontrar un augmenting path en el grafo residual
        d.assign(V, INF); d[s] = 0; vis[s] = 1;
        queue<int> q({s});
        while (!q.empty()) {
            int u = q.front(); q.pop(); vis[u] = 0;
            for (auto &idx : AL[u]) {                          // Explorar los vecinos de u
                auto &[v, cap, flow, cost] = EL[idx];          // Guardado en EL[idx]
                if ((cap-flow > 0) && (d[v] > d[u] + cost)) {  // Arista residual positiva
                    d[v] = d[u]+cost;
                    if(!vis[v]) q.push(v), vis[v] = 1;
                }
            }
        }
        return d[t] != INF;             // Tiene un augmenting path
    }

    ll DFS(int u, int t, ll f = INF) {                               // Ir de s->t
        if ((u == t) || (f == 0)) return f;
        vis[u] = 1;
        for (int &i = last[u]; i < (int)AL[u].size(); ++i) {          // Desde la ultima arista
            auto &[v, cap, flow, cost] = EL[AL[u][i]];
            if (!vis[v] && d[v] == d[u]+cost) {                       // En el grafo del nivel
                actual
                if (ll pushed = DFS(v, t, min(f, cap-flow))) {
                    total_cost += pushed * cost;
                    flow += pushed;
                    auto &[rv, rcap, rflow, rcost] = EL[AL[u][i]^1];    // Arista de regreso
                    rflow -= pushed;
                    vis[u] = 0;
                    return pushed;
                }
            }
        }
        vis[u] = 0;
        return 0;
    }

 public:
    min_cost_max_flow(int initialV) : V(initialV), total_cost(0) {
        EL.clear();
        AL.assign(V, vi());
        vis.assign(V, 0);
    }

    // Si se agrega una arista bidireccional u<->v con peso w en el grafo de flujo,
        // asigna directed = false. El valor por defecto es true (Arista dirigida)
    void add_edge(int u, int v, ll w, ll c, bool directed = true) {
        if (u == v) return;                     // Por seguridad: Evita ciclos en el mismo nodo
        EL.emplace_back(v, w, 0, c);            // u->v, cap w, flow 0, cost c
        AL[u].push_back(EL.size()-1);           // Para recordar el indice
        EL.emplace_back(u, 0, 0, -c);           // Arista de regreso
        AL[v].push_back(EL.size()-1);           // Para recordar el indice
        if (!directed) add_edge(v, u, w, c, true);  // Agregar de nuevo en reversa
    }

    pair<ll, ll> mcmf(int s, int t) {
        ll mf = 0;                              // mf = Max flow
        while (SPFA(s, t)) {                    // Time complexity O(V^2*E)
            last.assign(V, 0);                  // Aceleracion importante
```

```
            while (ll f = DFS(s, t))                 // exhaust blocking flow
                mf += f;
        }
        return {mf, total_cost};
    }
};

int main() {
    // Leer numero de nodos(V), source(s), sink(t)
    // De preferencia asignar s = 0, t = V-1
    // min_cost_max_flow mf(V);
    // Crear aristas usando el metodo add_edge(u, v, w, c);

    return 0;
}
```

4.16 Kuhn BPM

```
// Time complexity O(n*m)
// Source: CP algorithms
typedef vector<int> vi;

struct Kuhn{
    int n, k;           // #Nodos en la primera parte del grafo(n), #Nodos en la segunda parte del
          grafo(k)_
    vector<vi> adj;     // Lista de adyacencia del grafo
    vi mt;              // Conexiones del bpm, donde mt[i] es el nodo de la primera parte conectado al
           nodo i de la segunda parte
    vector<bool> used;  // Vector de visitados para el dfs

    Kuhn(int _n, int _k) : n(_n), k(_k), adj(max(_n, _k) + 1), mt(_k, -1) {}

    void add_edge(int u, int v) { adj[u].push_back(v); }

    bool try_kuhn(int v) {
        if (used[v])
            return false;
        used[v] = true;
        for (int to : adj[v]) {
            if (mt[to] == -1 || try_kuhn(mt[to])) {
                mt[to] = v;
                return true;    // Retorna true si encuentra un augmenting path
            }
        }
        return false;           // Retorna false en caso contrario
    }

    int mcbm(){
        for (int v = 0; v < n; ++v) {
            used.assign(n, false);
            try_kuhn(v);
        }

        int maxMatch = 0;

        for (int i = 0; i < k; ++i)
            if (mt[i] != -1){
                // printf("%d %d\n", mt[i] + 1, i + 1);
                maxMatch++;
            }
        return maxMatch;
    }
};
```

4.17 Hopcroft Karp

```
/*
    Descripcion: Algoritmo para resolver el problema de maximum bipartite
    matching. Los nodos para c1 y c2 deben comenzar desde el indice 1
    Tiempo: O(sqrt(|V|) * E)
*/

int dist[MAXN], pairU[MAXN], pairV[MAXN], c1, c2;
vi graph[MAXN];

bool bfs() {
  queue<int> q;

  for (int u = 1; u <= c1; u++) {
    if (!pairU[u]) {
      dist[u] = 0;
      q.push(u);
    } else
      dist[u] = INF;
  }

  dist[0] = INF;

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

    if (dist[u] < dist[0]) {
      for (int v : graph[u]) {
        if (dist[pairV[v]] == INF) {
          dist[pairV[v]] = dist[u] + 1;
          q.push(pairV[v]);
        }
      }
    }
  }

  return dist[0] != INF;
}

bool dfs(int u) {
  if (u) {
    for (int v : graph[u]) {
      if (dist[pairV[v]] == dist[u] + 1) {
        if (dfs(pairV[v])) {
          pairU[u] = v;
          pairV[v] = u;
          return true;
        }
      }
    }

    dist[u] = INF;
    return false;
  }
  return true;
}

int hopcroftKarp() {
  int result = 0;

  while (bfs())
    for (int u = 1; u <= c1; u++)
      if (!pairU[u] && dfs(u))
        result++;

  return result;
}
```

4.18 Hungarian

```
/*
    Descripcion: Dado un grafo bipartito con pesos, empareja cada nodo de la izquieda
    con un nodo de la derecha, de tal manera que no hay un nodo en 2 emparejamientos y
    la suma de los pesos de las aristas es minima.
    Toma coste[N][M], donde coste[i][j] = coste de L[i] para ser emparejado con R[j] y
    retorna (min cost, match), donde L[i] esta emparejado con R[match[i]].
    Negar los costos para el costo maximo.
    Time complexity: O(N^2 * M)
    Source: Kactl
*/
#define sz(x)   (int) x.size()
typedef vector<int> vi;

pair<int, vi> hungarian(const vector<vi> &a) {
    if (a.empty()) return {0, {}};
    int n = sz(a) + 1, m = sz(a[0]) + 1;
    vi u(n), v(m), p(m), ans(n - 1);
    for(int i = 1; i < n; i++){
        p[0] = i;
        int j0 = 0; // add "dummy" worker 0
        vi dist(m, INT_MAX), pre(m, -1);
        vector<bool> done(m + 1);
        do { // dijkstra
            done[j0] = true;
            int i0 = p[j0], j1, delta = INT_MAX;
            for(int j = 1; j < n; j++) if (!done[j]) {
                auto cur = a[i0 - 1][j - 1] - u[i0] - v[j];
                if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
                if (dist[j] < delta) delta = dist[j], j1 = j;
            }
            for(int j = 0; j < m; j++){
                if (done[j]) u[p[j]] += delta, v[j] -= delta;
                else dist[j] -= delta;
            }
```

```
            j0 = j1;
        } while (p[j0]);
        while (j0) { // update alternating path
            int j1 = pre[j0];
            p[j0] = p[j1], j0 = j1;
        }
    }
    for(int j = 1; j < m; j++)
        if (p[j]) ans[p[j] - 1] = j - 1;
    return {-v[0], ans}; // min cost
}
```

4.19 General matching

```
/**
 * Descripcion: Variante de la implementacion de Gabow para el algoritmo
 * de Edmonds-Blossom. Maximo emparejamiento sin peso para un grafo en
 * general, con 1-indexacion. Si despues de terminar la llamada a solve(),
 * white[v] = 0, v es parte de cada matching maximo.
 * Tiempo: O(NM), mas rapido en la practica.
 */
struct MaxMatching {
    int N;
    vector<vi> adj;
    vector<int> mate, first;
    vector<bool> white;
    vector<pi> label;

    MaxMatching(int _N) : N(_N), adj(vector<vi>(N + 1)), mate(vi(N + 1)), first(vi(N + 1)), label(
          vector<pi>(N + 1)), white(vector<bool>(N + 1)) {}

    void addEdge(int u, int v) { adj.at(u).pb(v), adj.at(v).pb(u); }

    int group(int x) {
        if (white[first[x]])
        first[x] = group(first[x]);
        return first[x];
    }
    void match(int p, int b) {
        swap(b, mate[p]);
        if (mate[b] != p)
            return;
        if (!label[p].second)
            mate[b] = label[p].first, match(label[p].first, b);  // vertex label
        else
            match(label[p].first, label[p].second), match(label[p].second, label[p].first);  // edge
                  label
    }
    bool augment(int st) {
        assert(st);
        white[st] = 1;
        first[st] = 0;
        label[st] = {0, 0};

        queue<int> q;
        q.push(st);

        while (!q.empty()) {
        int a = q.front();
        q.pop();  // outer vertex
        for (auto& b : adj[a]) {
            assert(b);
            if (white[b]) {
                int x = group(a), y = group(b), lca = 0;
                while (x || y) {
                    if (y)
                    swap(x, y);
                    if (label[x] == pi{a, b}) {
                    lca = x;
                    break;
                    }
                    label[x] = {a, b};
                    x = group(label[mate[x]].first);
                }
                for (int v : {group(a), group(b)})
                    while (v != lca) {
                    assert(!white[v]);  // make everything along path white
                    q.push(v);
                    white[v] = true;
                    first[v] = lca;
                    v = group(label[mate[v]].first);
                    }
            } else if (!mate[b]) {
                mate[b] = a;
                match(a, b);
                white = vector<bool>(N + 1);  // reset
                return true;
            } else if (!white[mate[b]]) {
                white[mate[b]] = true;
                first[mate[b]] = b;
                label[b] = {0, 0};
                label[mate[b]] = pi{a, 0};
                q.push(mate[b]);
            }
        }
        }
        return false;
    }
    int solve() {
        int ans = 0;
        for(int st = 1; st < N +1; st++)
            if (!mate[st])
                ans += augment(st);
        for(int st = 1; st < N +1; st++)
            if (!mate[st] && !white[st])
                assert(!augment(st));
        return ans;
    }
};
```

4.20 2 SAT

```
/*
    Time complexity O(N + E), donde N es el numero de variables booleanas y E es el numero de
          clausulas
    Las variables negadas son representadas por inversiones de bits (~x)
    Uso:
        TwoSat ts(numero de variables booleanas);
        ts.either(0, ~3);           La variable 0 es verdadera o la variable 3 es falsa
        ts.setValue(2);             La variable 2 es verdadera
        ts.atMostOne({0, ~1, 2});   <= 1 de vars 0, ~1 y 2 son verdedero
        ts.solve();                 Retorna verdadero si existe solucion
        ts.values[0..N-1]           Tiene los valores asignados a las variables
    Source: KACTL
*/
typedef vector<int> vi;

struct TwoSat {
    int N; vector<vi> adj;
    vi values; // 0 = false, 1 = true

    TwoSat(int n = 0) : N(n), adj(2*n) {}

    int addVar() { adj.emplace_back(); adj.emplace_back(); return N++; }    // Opcional

    // Agrega una disyuncion
    void either(int x, int y) { // Nota: (a v b), es equivalente a la expresion (~a -> b) n (~b -> a)
        x = max(2*x, -1-2*x), y = max(2*y, -1-2*y);
        adj[x].push_back(y^1), adj[y].push_back(x^1);
    }
    void setValue(int x) { either(x, x); }                           // La variable x debe tener el
          valor indicado
    void implies(int x, int y) { either(~x, y); }                    // La variable x implica a y
    void make_diff(int x, int y) { either(x, y); either(~x, ~y); }   // Los valores tienen que ser
          diferentes
    void make_eq(int x, int y) {either(~x, y); either(x, ~y); }      // Los valores tienen que ser
          iguales

    void atMostOne(const vi& li) { // Opcional
        if (li.size() <= 1) return;
        int cur = ~li[0];
        for(int i = 2; i < li.size(); i++){
            int next = addVar();
            either(cur, ~li[i]); either(cur, next);
            either(~li[i], next); cur = ~next;
        }
        either(cur, ~li[1]);
    }

    vi dfs_num, comp; stack<int> st; int time = 0;
    int tarjan(int u) {    // Tarjan para encontrar los SCCs
        int x, low = dfs_num[u] = ++time; st.push(u);
        for(int v : adj[u]) if (!comp[v])
            low = min(low, dfs_num[v] ?: tarjan(v));
        if (low == dfs_num[u]) do {
            x = st.top(); st.pop();
            comp[x] = low;
            if (values[x>>1] == -1)
                values[x>>1] = x&1;
        } while (x != u);
        return dfs_num[u] = low;
    }

    bool solve() {
        values.assign(N, -1), dfs_num.assign(2*N, 0), comp.assign(2*N, 0);
```

```cpp
        for(int i = 0; i < 2*N; i++)
            if (!comp[i])
                tarjan(i);
        for(int i = 0; i < N; i++)
            if (comp[2*i] == comp[2*i+1])
                return 0;
        return 1;
    }
};
```

4.21 Find centroid

```cpp
/*
    Descripcion: dado un arbol, encuentra su centroide
    Tiempo: O(V)
*/
int dfs(int u, int p) {
  for (auto v : tree[u])
    if (v != p)
      subtreeSZ[u] += dfs(v, u);
  return subtreeSZ[u] += 1;
}

int centroid(int u, int p) {
  for (auto v : tree[u])
    if (v != p && subtreeSZ[v] * 2 > n)
      return centroid(v, u);
  return u;
}
```

5 Strings

5.1 Knuth Morris Pratt (KMP)

```cpp
// Time complexity O(n + m)
typedef vector<int> vi;

vi kmpPreprocess(string &P) {   // Preprocesamiento
    int m = P.size();
    vi b(m + 1);                // b = Back table
    int i = 0, j = -1; b[0] = -1;
    while (i < m) {                                 // Preprocesamiento de P
        while ((j >= 0) && (P[i] != P[j])) j = b[j];    // Diferente, reset j
        ++i; ++j;                                   // Igual, avanzan ambos
        b[i] = j;
    }
    return b;
}

// T = Cadena donde se busca, P = Patron a buscar
int kmpSearch(string &T, string &P) {               // Busqueda del patron en la cadena
    vi b = kmpPreprocess(P);
    int freq = 0;
    int i = 0, j = 0;
    int n = T.size(), m = P.size();
    while (i < n) {                                 // Buscar a traves de T
        while ((j >= 0) && (T[i] != P[j])) j = b[j];    // Diferente, reset j
        ++i; ++j;                                   // Igual, avanzan ambos
        if (j == m) {                               // Una coincidencia es encontrada
            ++freq;
            // printf("P se encuentra en el indice %d de T\n", i-j);
            j = b[j];                               // Prepara j para la siguiente
        }
    }
    return freq;
}

int main() {
    string T="I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN";
    string P="SEVENTY SEVEN";

    printf("Knuth-Morris-Pratt, #match = %d\n", kmpSearch(T, P));

    return 0;
}
```

5.2 Hashing

```cpp
// Time complexity Hashing O(n), hashInterval O(1)
typedef long long ll;

// Operaciones con modulo
inline int add(int a, int b, int mod) { a += b; return a >= mod ? a - mod : a; }
inline int sub(int a, int b, int mod) { a -= b; return a < 0 ? a + mod : a; }
inline int mul(int a, int b, int mod) { return ((ll)a*b) % mod; }

const int MOD[] = {(int)1e9+7, (int)1e9+9};     // Modulos

struct H{
    int x, y;
    H(int _x = 0) : x(_x), y(_x) {}
    H(int _x, int _y) : x(_x), y(_y) {}
    inline H operator+(const H& o){ return {add(x, o.x, MOD[0]), add(y, o.y, MOD[1])}; }
    inline H operator-(const H& o){ return {sub(x, o.x, MOD[0]), sub(y, o.y, MOD[1])}; }
    inline H operator*(const H& o){ return {mul(x, o.x, MOD[0]), mul(y, o.y, MOD[1])}; }
    inline bool operator==(const H& o) { return x == o.x && y == o.y; }
};

const int MAXN = 2e5+5;     // Valor maximo de la longitud de un string
const H P = {257, 577};     // Bases primas
vector<H> pw;               // Vector con las potencias de las bases

void computePowers(){ pw.resize(MAXN + 1); pw[0] = {1, 1}; for(int i = 0; i < MAXN; i++) pw[i + 1] =
      pw[i] * P; }

struct Hash{
    vector<H> ha;
    Hash(string& s){    // O(n)
        if(pw.empty())  computePowers();
        int l = (int) s.size(); ha.resize(l + 1);
        for(int i = 0; i < l; i++) ha[i + 1] = ha[i] * P + s[i];
    }
    H hashInterval(int l, int r){ return ha[r] - ha[l] * pw[r - l]; }    // O(1), regresa el hash del
          intervalo [l, r)
};

H hashString(string& s){ H ret; for(char c : s) ret = ret * P + c; return ret; } // O(n)

// Para "concatenar" hashes, de tal manera que se pueda obtener el hash de la concatenacion de 2
      substrings,
// se puede hacer de la siguiente manera: hashIzq * pw[len] + hashDer, en donde len = longitud de
      hashDer
H combineHash(H hI, H hD, int len){ return hI * pw[len] + hD; } // O(1)
```

5.3 Trie

```cpp
// Implementacion del arbol de prefijos usando mapa
struct TrieNode {
    map<char, TrieNode *> children;
    bool isEndOfWord;
    int numPrefix;

    TrieNode() : isEndOfWord(false), numPrefix(0) {}
};

class Trie {
   private:
    TrieNode *root;

   public:
    Trie() : root(new TrieNode()) {}

    void insert(string word) {  // Inserta una palabra en el trie
        TrieNode *curr = root;
        for (char c : word) {
            if (curr->children.find(c) == curr->children.end())
                curr->children[c] = new TrieNode();
            curr = curr->children[c];
            curr->numPrefix++;
        }
        curr->isEndOfWord = true;
    }

    bool search(string word) {  // Busca si una palabra esta en el trie
        TrieNode *curr = root;
        for (char c : word) {
            if (curr->children.find(c) == curr->children.end())
                return false;
            curr = curr->children[c];
        }
        return curr->isEndOfWord;
    }

    bool startsWith(string prefix) {    // Busca si alguna palabra del trie inicia con un prefijo
        TrieNode *curr = root;
```

```
        for (char c : prefix) {
            if (curr->children.find(c) == curr->children.end())
                return false;
            curr = curr->children[c];
        }
        return true;
    }

    int countPrefix(string prefix) {    // Cuenta la cantidad de palabras que inician con un prefijo
        TrieNode *curr = root;
        for (char c : prefix) {
            if (curr->children.find(c) == curr->children.end())
                return 0;
            curr = curr->children[c];
        }
        return curr->numPrefix;
    }
};
```

5.4 Aho-Corasick

```
// Implementacion de Aho-Corasick y Aho-Corasick dinamico

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef long long ll;

class AhoCorasick {
public:
    struct Node {
        map<char, int> ch;
        vi accept;
        int link = -1;
        int cnt = 0;

        Node() = default;
    };

    vector<Node> states;
    map<int, int> accept_state;

    explicit AhoCorasick() : states(1) {}

    void insert(const string& s, int id = -1) { // O(|s|)
        int i = 0;
        for (char c : s) {
            if (!states[i].ch.count(c)) {
                states[i].ch[c] = states.size();
                states.emplace_back();
            }
            i = states[i].ch[c];
        }
        ++states[i].cnt;
        states[i].accept.push_back(id);
        accept_state[id] = i;
    }

    void clear() {
        states.clear();
        states.emplace_back();
    }

    int get_next(int i, char c) const {
        while (i != -1 && !states[i].ch.count(c)) i = states[i].link;
        return i != -1 ? states[i].ch.at(c) : 0;
    }

    void build() {  // O(sum(|s|))
        queue<int> que;
        que.push(0);
        while (!que.empty()) {
            int i = que.front();
            que.pop();

            for (auto [c, j] : states[i].ch) {
                states[j].link = get_next(states[i].link, c);
                states[j].cnt += states[states[j].link].cnt;

                auto& a = states[j].accept;
                auto& b = states[states[j].link].accept;
                vi accept;
                set_union(a.begin(), a.end(), b.begin(), b.end(), back_inserter(accept));
                a = accept;

                que.push(j);
            }
        }
    }

    ll count(const string& str) const { // O(|str| + sum(|s|))
        ll ret = 0;
        int i = 0;
        for (auto c : str) {
            i = get_next(i, c);
            ret += states[i].cnt;
        }
        return ret;
    }

    // Lista de (id, index)
    vector<ii> match(const string& str) const { // O(|str| + sum(|s|))
        vector<ii> ret;
        int i = 0;
        for (int k = 0; k < (int) str.size(); ++k) {
            char c = str[k];
            i = get_next(i, c);
            for (auto id : states[i].accept) {
                ret.emplace_back(id, k);
            }
        }
        return ret;
    }
};

class DynamicAhoCorasick {
    vector<vector<string>> dict;
    vector<AhoCorasick> ac;

public:
    void insert(const string& s) {  // O(|s| log n)
        int k = 0;
        while (k < (int) dict.size() && !dict[k].empty()) ++k;
        if (k == (int) dict.size()) {
            dict.emplace_back();
            ac.emplace_back();
        }

        dict[k].push_back(s);
        ac[k].insert(s);

        for (int i = 0; i < k; ++i) {
            for (auto& t : dict[i]) {
                ac[k].insert(t);
            }
            dict[k].insert(dict[k].end(), dict[i].begin(), dict[i].end());
            ac[i].clear();
            dict[i].clear();
        }

        ac[k].build();
    }

    ll count(const string& str) const { // O(|str| + sum(|s| log n))
        ll ret = 0;
        for (int i = 0; i < (int) ac.size(); ++i) ret += ac[i].count(str);
        return ret;
    }
};
```

5.5 Manacher

```
// Time complexity O(n), donde n es la longitud del string
typedef vector<int> vi;

struct Manacher{
    vi p;                          // Vector para guardar cual es el palindromo mas grande con centro
        en esa posicion
    string w;                      // String original
    int lpL, lpR;                  // Posicion de inicio y fin del palindromo mas grande
    Manacher(string& str) : w(str) {
        string s;                  // String a la que se aplicara el Manacher
        for(char c : str)          // Se agregan caracteres "no validos" entre medio para unificar
            longitudes pares e impares
            s += string("#") + c;
        s += "#";
        int n = s.size(), l = 1, r = 1, longestP = -1; p.assign(n, 1);
        for(int i = 0; i < n; i++){
            p[i] = max(0, min(r - i, p[max(0, l + r - i)]));
            while(i - p[i] >= 0 && i + p[i] < n && s[i - p[i]] == s[i + p[i]])
                p[i]++;
            if(i + p[i] > r)
                l = i - p[i], r = i + p[i];
            if(p[i] > longestP)
                longestP = p[i], lpL = l + 1, lpR = r;
        }
    }
```

```
    int getLongestAt(int center, bool odd){              // Regresa la longitud del palindromo mas
          grande con ese centro
        return p[2 * center + 1 + !odd] - 1;
    }
    string getPalindromeAt(int center, bool odd){        // Regresa el palindromo mas grande con ese
          centro
        int len = getLongestAt(center, odd);
        return w.substr(center + !odd - len / 2, len);
    }
    string getLongestPalindrome(){                       // Regresa el palindromo mas grande en toda el
          string
        return w.substr(lpL / 2, (lpR - lpL) / 2);
    }
    bool checkPalindrome(int l, int r){                  // Verifica si un rango del string original es
          palindromo
        return (r - l + 1) <= getLongestAt((l + r) / 2, r % 2 == l % 2);
    }
};
```

5.6 Suffix array

```
/*
 * Descripcion: Un SuffixArray es un array ordenado de todos los sufijos de un string
 * Tiempo: O(|S|)
 * Aplicaciones:
 *  - Encontrar todas las ocurrencias de un substring P dentro del string S - O(|P| log n)
 *  - Construir el longest common prefix-interval - O(n log n)
 *  - Contar todos los substring diferentes en el string S - O(n)
 *  - Encontrar el substring mas largo entre dos strings S y T - O(|S|+|T|)
 */

struct SuffixArray {
    vi SA, LCP;
    string S;
    int n;
    SuffixArray(string &s, int lim = 256) : S(s), n(SZ(s) + 1) {  // O(n log n)
        int k = 0, a, b;
        vi x(ALL(s) + 1), y(n), ws(max(n, lim)), rank(n);
        SA = LCP = y, iota(ALL(SA), 0);

        // Calcular SA
        for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
            p = j, iota(ALL(y), n - j);
            for(int i = 0; i < n; i++) {
                if (SA[i] >= j) y[p++] = SA[i] - j;
            }
            fill(ALL(ws), 0);
            for(int i = 0; i < n; i++) {
                ws[x[i]]++;
            }
            for(int i = 1; i < lim; i++) {
                ws[i] += ws[i - 1];
            }
            for (int i = n; i--;) SA[--ws[x[y[i]]]] = y[i];
            swap(x, y), p = 1, x[SA[0]] = 0;
            FOR(i, 1, n) {
                a = SA[i - 1];
                b = SA[i], x[b] = (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
            }
        }

        // Calcular LCP (longest common prefix)
        for(int i = 1; i < n; i++) {
        rank[SA[i]] = i;
        }
        for (int i = 0, j; i < n - 1; LCP[rank[i++]] = k)
            for (k &&k--, j = SA[rank[i] - 1]; s[i + k] == s[j + k]; k++)
                ;
    }

    /*
    * Retorna el lower_bound de la subcadena sub en el Suffix Array
    * Tiempo: O(|sub| log n)
    */
    int lower(string &sub) {
        int l = 0, r = n - 1;
        while (l < r) {
            int mid = (l + r) / 2;
            int res = S.compare(SA[mid], SZ(sub), sub);
            (res >= 0) ? r = mid : l = mid + 1;
        }
        return l;
    }

    /*
    * Retorna el upper_bound de la subcadena sub en el Suffix Array
    * Tiempo: O(|sub| log n)
    */
    int upper(string &sub) {
        int l = 0, r = n - 1;
        while (l < r) {
            int mid = (l + r) / 2;
            int res = S.compare(SA[mid], SZ(sub), sub);
            (res > 0) ? r = mid : l = mid + 1;
        }
        if (S.compare(SA[r], SZ(sub), sub) != 0) --r;
        return r;
    }

    /*
    * Busca si se encuentra la subcadena sub en el Suffix Array
    * Tiempo: O(|sub| log n)
    */
    bool subStringSearch(string &sub) {
        int L = lower(sub);
        if (S.compare(SA[L], SZ(sub), sub) != 0) return 0;
        return 1;
    }

    /*
    * Cuenta la cantidad de ocurrencias de la subcadena sub en el Suffix Array
    * Tiempo: O(|sub| log n)
    */
    int countSubString(string &sub) {
        return upper(sub) - lower(sub) + 1;
    }

    /*
    * Cuenta la cantidad de subcadenas distintas en el Suffix Array
    * Tiempo: O(n)
    */
    ll countDistinctSubstring() {
        ll result = 0;
        for(int i = 1; i < n; i++) {
            result += ll(n - SA[i] - 1 - LCP[i]);
        }
        return result;
    }

    /*
    * Busca la subcadena mas grande que se encuentra en el string T y S
    * Uso: Crear el SuffixArray con una cadena de la concatenacion de T
    * y S separado por un caracter especial (T + '#' + S)
    * Tiempo: O(n)
    */
    string longestCommonSubstring(int lenS, int lenT) {
        int maximo = -1, indice = -1;
        for(int i = 2; i < n; i++) {
            if ((SA[i] > lenS && SA[i - 1] < lenS) || (SA[i] < lenS && SA[i - 1] > lenS)) {
                if (LCP[i] > maximo) {
                    maximo = LCP[i];
                    indice = SA[i];
                }
            }
        }
        return S.substr(indice, maximo);
    }

    /*
    * A partir del Suffix Array se crea un Suffix Array inverso donde la
    * posicion i del string S devuelve la posicion del sufijo S[i..n) en el Suffix Array
    * Tiempo: O(n)
    */
    vi constructRSA() {
        vi RSA(n);
        for(int i = 0; i < n; i++) {
            RSA[SA[i]] = i;
        }
        return RSA;
    }
};
```

6 Geometry

6.1 Points and lines

```
const double INF = 1e9, EPS = 1e-9;

double DEG_to_RAD(double d) { return d*M_PI/180.0; }
double RAD_to_DEG(double r) { return r*180.0/M_PI; }

struct point_i {    //  Punto con coordenadas de valores enteros
```

```
    int x, y;
    point_i() { x = y = 0; }
    point_i(int _x, int _y) : x(_x), y(_y) {}
};

struct point {      // Punto con coordenadas de valores reales
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) {}
    bool operator < (point other) const {
        if (fabs(x-other.x) > EPS)
            return x < other.x;
        return y < other.y;
    }
    bool operator == (point other) const {
        return (fabs(x-other.x) < EPS && (fabs(y-other.y) < EPS));
    }
};

double dist(point p1, point p2) {                   // Distancia euclidiana
    return hypot(p1.x-p2.x, p1.y-p2.y);
}

// Rota el punto p theta grados en ccw con respecto al origen (0, 0)
point rotate(point p, double theta) {
    double rad = DEG_to_RAD(theta);
    return point(p.x*cos(rad) - p.y*sin(rad),  p.x*sin(rad) + p.y*cos(rad));
}

struct line { double a, b, c; };                 // Linea con la ecuacion ax + by + c = 0

// Convierte 2 puntos a una linea y la guarda en la linea pasada por referencia
void pointsToLine(point p1, point p2, line &l) {
    if (fabs(p1.x-p2.x) < EPS)                      // Linea vertical
        l = {1.0, 0.0, -p1.x};                      // Valores default
    else {
        double a = -(double)(p1.y-p2.y) / (p1.x-p2.x);
        l = {a, 1.0, -(double)(a*p1.x) - p1.y};     // Nota: b siempre es 1.0
    }
}

bool areParallel(line l1, line l2) {             // Checa si 2 lineas son paralelas
    return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS);
}

bool areSame(line l1, line l2) {                 // Checa si 2 lineas son iguales
    return areParallel(l1 ,l2) && (fabs(l1.c-l2.c) < EPS);
}

// Retorna true y el punto de intereseccion p, si 2 lineas se intersecan
bool areIntersect(line l1, line l2, point &p) {
    if (areParallel(l1, l2)) return false;
    p.x = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);    // Resuelve un sistema de 2 ecuaciones
          lineales con 2 incognitas
    if (fabs(l1.b) > EPS) p.y = -(l1.a*p.x + l1.c);             // Caso especial: prueba si es linea
         vertical para evitar la division entre 0
    else                 p.y = -(l2.a*p.x + l2.c);
    return true;
}

struct vec {    // Vector
    double x, y;
    vec(double _x, double _y) : x(_x), y(_y) {}
};

// Convierte 2 puntos en el vector a->b
vec toVec(const point &a, const point &b) { return vec(b.x-a.x, b.y-a.y); }

// Escala un vector por el valor s
vec scale(const vec &v, double s) { return vec(v.x*s, v.y*s); }

// Traslada p, de acuerdo a v
point translate(const point &p, const vec &v) { return point(p.x+v.x, p.y+v.y); }

// Convierte un punto y una pendiente en una linea
void pointSlopeToLine(point p, double m, line &l) {
    l.a = -m;
    l.b = 1;
    l.c = -((l.a * p.x) + (l.b * p.y));
}

// Obtiene el punto mas cercano entre una linea y un punto
void closestPoint(line l, point p, point &ans) {
    line perpendicular;                             // Esta linea es perpendicular a l y pasa a traves
          de p
    if (fabs(l.b) < EPS) {                          // Linea vertical
        ans.x = -(l.c);
        ans.y = p.y;
        return;
    }
    if (fabs(l.a) < EPS) {                          // Linea horizontal
        ans.x = p.x;
        ans.y = -(l.c);
        return;
    }
    pointSlopeToLine(p, 1/l.a, perpendicular);      // Linea normal
    // Interseca l con esta linea perpendicular y el punto de interseccion es el punto mas cercano
    areIntersect(l, perpendicular, ans);
}

// Retorna la reflexion de un punto sobre una linea
void reflectionPoint(line l, point p, point &ans) {
    point b;
    closestPoint(l, p, b);
    vec v = toVec(p, b);
    ans = translate(translate(p, v), v);          // Traslada p 2 veces
}

// Retorna el producto punto entre los vectores a & b
double dot(vec a, vec b) { return (a.x*b.x + a.y*b.y); }

// Retorna el cuadrado de la magnitud de un vector
double norm_sq(vec v) { return v.x*v.x + v.y*v.y; }

// Regresa el angulo (en radianes) formado entre 2 vectores formados por 3 puntos
double angle(const point &a, const point &o, const point &b) {
    vec oa = toVec(o, a), ob = toVec(o, b);       // a != o != b
    return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}

// Retorna la distancia desde un punto p a una linea definida por 2 punto a & b (deben ser diferentes)
// El punto mas cercano se guarda en el punto c
double distToLine(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    // Formula: c = a + u*ab
    c = translate(a, scale(ab, u));                // Traslada el punto a al punto c
    return dist(p, c);
}

// Retorna la distancia desde un punto p a un segmento de linea ab, definido por 2 puntos a & b (deben
      ser diferentes)
// El punto mas cercano se guarda en el punto c
double distToLineSegment(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    if (u < 0.0) {                                 // Mas cercano al punto a
        c = point(a.x, a.y);
        return dist(p, a);
    }
    if (u > 1.0) {                                 // Mas cercano al punto b
        c = point(b.x, b.y);
        return dist(p, b);
    }
    return distToLine(p, a, b, c);
}

// Retorna el producto cruz entre 2 vectores a & b
double cross(vec a, vec b) { return a.x*b.y - a.y*b.x; }

// Retorna 2 veces el area del triangulo A-B-C
// int area2(point p, point q, point r) {
//   return p.x * q.y - p.y * q.x +
//          q.x * r.y - q.y * r.x +
//          r.x * p.y - r.y * p.x;
// }

// Nota: Para aceptar puntos colineares, se debe cambiar el "> 0"
// Retorna true si el punto r se encuentra al lado izquierdo de la linea pq
bool ccw(point p, point q, point r) { return cross(toVec(p, q), toVec(p, r)) > -EPS; }

// Retorna true si el punto r esta en la linea pq
bool collinear(point p, point q, point r) { return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }
```

6.2 Triangles

```
// NOTA: Para las siguientes funciones se deben agregar los elementos de puntos, lineas y vectores
     necesarios

const double EPS = 1e-9;

double perimeter(double ab, double bc, double ca) { return ab + bc + ca; }

double perimeter(point a, point b, point c) { return dist(a, b) + dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {      // Formula de Heron
    double s = 0.5 * perimeter(ab, bc, ca);
    return sqrt(s) * sqrt(s-ab) * sqrt(s-bc) * sqrt(s-ca);
}
```

```cpp
double area(point a, point b, point c) { return area(dist(a, b), dist(b, c), dist(c, a)); }

// Retorna el radio del circulo que se forma tocando los lados de un triangulo
double rInCircle(double ab, double bc, double ca) { return area(ab, bc, ca) / (0.5 * perimeter(ab, bc,
      ca)); }

double rInCircle(point a, point b, point c) { return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

/*
    Retorna 1 si hay un centro del inCircle, retorna 0 de otra manera,
    si se retorna 1, ctr sera el centro del inCircle y r es el radio del mismo
*/
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
    r = rInCircle(p1, p2, p3);
    if (fabs(r) < EPS) return 0;                    // No hay centro

    line l1, l2;                        // Calcula los 2 angulos bisectores
    double ratio = dist(p1, p2) / dist(p1, p3);
    point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
    pointsToLine(p1, p, l1);

    ratio = dist(p2, p1) / dist(p2, p3);
    p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
    pointsToLine(p2, p, l2);

    areIntersect(l1, l2, ctr);            // Obtiene su punto de interseccion
    return 1;
}

// Retorna el radio del circulo en el que su circunferencia toca los 3 puntos del triangulo
double rCircumCircle(double ab, double bc, double ca) { return ab * bc * ca / (4.0 * area(ab, bc, ca))
     ; }

double rCircumCircle(point a, point b, point c) { return rCircumCircle(dist(a, b), dist(b, c), dist(c,
      a)); }

/*
    Retorna 1 si hay un centro del circumCircle, retorna 0 de otra manera
    si se retorna 1, ctr sera el centro del circumCircle y r es el radio del mismo
*/
int circumCircle(point p1, point p2, point p3, point &ctr, double &r){
    double a = p2.x - p1.x, b = p2.y - p1.y;
    double c = p3.x - p1.x, d = p3.y - p1.y;
    double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
    double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
    double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x - p2.x));
    if (fabs(g) < EPS) return 0;

    ctr.x = (d*e - b*f) / g;
    ctr.y = (a*f - c*e) / g;
    r = dist(p1, ctr);  // r = distancia del centro a 1 de los 3 puntos
    return 1;
}

// Retorna true si el punto d esta dentro del circumCircle definido por a, b, c
int inCircumCircle(point a, point b, point c, point d) {
    return (a.x - d.x) * (b.y - d.y) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y) * (c.y - d.y)) +
        (a.y - d.y) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y)) * (c.x - d.x) +
        ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.x - d.x) * (c.y - d.y) -
        ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.y - d.y) * (c.x - d.x) -
        (a.y - d.y) * (b.x - d.x) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y) * (c.y - d.y)) -
        (a.x - d.x) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y)) * (c.y - d.y) > 0 ? 1 :
             0;
}

// Retorna si 3 lados pueden formar un triangulo
bool canFormTriangle(double a, double b, double c) { return (a+b > c) && (a+c > b) && (b+c > a); }
```

6.3 Polygons

```cpp
const double EPS = 1e-9;

// NOTA: Para las siguientes funciones se deben agregar los elementos de puntos y vectores necesarios

/*
    Retorna el perimetro del poligono P, que es la suma de las distancias Euclideanas
    de los segmentos consecutivos de lineas (aristas del poligono)
*/
double perimeter(const vector<point> &P) {
    double ans = 0.0;
    for (int i = 0; i < (int)P.size()-1; ++i)      // Nota: P[n-1] = P[0]
        ans += dist(P[i], P[i+1]);
    return ans;
}

// Retorna el area del poligono P haciendo uso de la Shoelace formula
double area(const vector<point> &P) {
    double ans = 0.0;
    for (int i = 0; i < (int)P.size()-1; ++i)
        ans += (P[i].x*P[i+1].y - P[i+1].x*P[i].y);
    return fabs(ans)/2.0;
}

// Calculo del area del poligono P escrito en operaciones con vectores
double area_alternative(const vector<point> &P) {
    double ans = 0.0; point O(0.0, 0.0);           // O = El origen
    for (int i = 0; i < (int)P.size()-1; ++i)      // Suma de las areas
        ans += cross(toVec(O, P[i]), toVec(O, P[i+1]));
    return fabs(ans)/2.0;
}

/*
    Retorna true si el poligono es convexo, lo que se determina si siempre se hace
    una vuelta hacia el mismo lado, mientras se analizan las aristas del poligono una por una
*/
bool isConvex(const vector<point> &P) {
    int n = (int)P.size();
    // Un point/sz=2 o una linea/sz=3 no es convexo
    if (n <= 3) return false;
    bool firstTurn = ccw(P[0], P[1], P[2]);
    for (int i = 1; i < n-1; ++i)
        if (ccw(P[i], P[i+1], P[(i+2) == n ? 1 : i+2]) != firstTurn)
        return false;                              // Concavo
    return true;                                   // Convexo
}

/*
    Segun el punto pt, retorna 1 si esta dentro del poligono P, 0 si esta sobre
    un vertice o arista, y -1 si esta fuera del poligono
*/
int insidePolygon(point pt, const vector<point> &P) {
    int n = (int)P.size();
    if (n <= 3) return -1;                         // Evitar un punto o una linea
    bool on_polygon = false;
    for (int i = 0; i < n-1; ++i)                  // Sobre vertice/arista
        if (fabs(dist(P[i], pt) + dist(pt, P[i+1]) - dist(P[i], P[i+1])) < EPS)
        on_polygon = true;
    if (on_polygon) return 0;                      // pt esta sobre el poligono
    double sum = 0.0;                              // Primer = Ultimo punto
    for (int i = 0; i < n-1; ++i) {
        if (ccw(pt, P[i], P[i+1]))
        sum += angle(P[i], pt, P[i+1]);            // Vuelta a la izquierda/ccw
        else
        sum -= angle(P[i], pt, P[i+1]);            // Vuelta a la derecha/cw
    }
    return fabs(sum) > M_PI ? 1 : -1;              // 360d->dentro, 0d->fuera
}

// Retorna el punto de interseccion entre el segmento p-q y la linea A-B
point lineIntersectSeg(point p, point q, point A, point B) {
    double a = B.y-A.y, b = A.x-B.x, c = B.x*A.y - A.x*B.y;
    double u = fabs(a*p.x + b*p.y + c);
    double v = fabs(a*q.x + b*q.y + c);
    return point((p.x*v + q.x*u) / (u+v), (p.y*v + q.y*u) / (u+v));
}

/*
    Corta el poligono Q a traves de la linea formada por punto A->Punto B (El orden importa)
    EL poligono que se retorna es el poligono que queda en la parte izquierda del corte
    Nota: EL ultimo punto debe de ser igual al primero
*/
vector<point> cutPolygon(point A, point B, const vector<point> &Q) {
    vector<point> P;
    for (int i = 0; i < (int)Q.size(); ++i) {
        double left1 = cross(toVec(A, B), toVec(A, Q[i])), left2 = 0;
        if (i != (int)Q.size()-1) left2 = cross(toVec(A, B), toVec(A, Q[i+1]));
        if (left1 > -EPS) P.push_back(Q[i]);         // Q[i] esta a la izquierda
        if (left1*left2 < -EPS)                      // Cruza la linea AB
        P.push_back(lineIntersectSeg(Q[i], Q[i+1], A, B));
    }
    if (!P.empty() && !(P.back() == P.front()))
        P.push_back(P.front());                      // wrap around Envolver
    return P;
}

vector<point> CH_Graham(vector<point> &Pts) {       // O(n log n)
    vector<point> P(Pts);
    int n = (int)P.size();
    if (n <= 3) {                                   // punto/linea/triangulo
        if (!(P[0] == P[n-1])) P.push_back(P[0]);   // Caso esquina
        return P;                                   // EL CH es P
    }

    // Primer paso, hallar P0 = punto con la menor Y, y en caso de empate, la X mayor. O(n log n)
    int P0 = min_element(P.begin(), P.end())-P.begin();
    swap(P[0], P[P0]);                             // swap P[P0] y P[0]

    // Segundo paso, ordenar los puntos por su angulo alrededor de P0
```

```
    sort(++P.begin(), P.end(), [&](point a, point b) {
        return ccw(P[0], a, b);                        // Se usa P[0] como el pivote
    });

    // Tercer paso, las pruebas de ccw. O(n)
    vector<point> S({P[n-1], P[0], P[1]});          // S inicial
    int i = 2;                                      // Checamos el resto
    while (i < n) {                                 // n > 3, O(n)
        int j = (int)S.size()-1;
        if (ccw(S[j-1], S[j], P[i]))                // Vuelta CCW
            S.push_back(P[i++]);                    // Se acepta este punto
        else                                        // Vuelta CW
            S.pop_back();                           // Pop hasta que haya una vuelta CCW
    }
    return S;
}

vector<point> CH_Andrew(vector<point> &Pts) {       // O(n log n)
    int n = Pts.size(), k = 0;
    vector<point> H(2*n);
    sort(Pts.begin(), Pts.end());                   // Ordenar los puntos por x/y
    for (int i = 0; i < n; ++i) {                   // Construir el hull de abajo
        while ((k >= 2) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
        H[k++] = Pts[i];
    }
    for (int i = n-2, t = k+1; i >= 0; --i) {       // Construir el hull de arriba
        while ((k >= t) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
        H[k++] = Pts[i];
    }
    H.resize(k);
    return H;
}
```

6.4 Circles

```
double DEG_to_RAD(double d) { return d*M_PI/180.0; }
double RAD_to_DEG(double r) { return r*180.0/M_PI; }

struct point_i {    //  Punto con coordenadas de valores enteros
    int x, y;
    point_i() { x = y = 0; }
    point_i(int _x, int _y) : x(_x), y(_y) {}
};

struct point {      // Punto con coordenadas de valores reales
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) {}
};

// Regresa si el punto p se encuentra dentro del circulo con centro c y radio r
int insideCircle(point_i p, point_i c, int r) {
    int dx = p.x-c.x, dy = p.y-c.y;
    int Euc = dx*dx + dy*dy, rSq = r*r;
    return Euc < rSq ? 1 : Euc == rSq ? 0 : -1; // dentro/borde/fuera
}

/*
    Retorna si se intersecan 2 circulos de radio r con centro p1 y p2
    Si se intersecan, se obtiene el punto donde lo hacen en c
    Para obtener el segundo punto de interseccion, se invierten p1 y p2
*/
bool circle2PtsRad(point p1, point p2, double r, point &c) {
    double d2 = (p1.x-p2.x) * (p1.x-p2.x) + (p1.y-p2.y) * (p1.y-p2.y);
    double det = r*r / d2 - 0.25;
    if (det < 0.0) return false;
    double h = sqrt(det);
    c.x = (p1.x+p2.x) * 0.5 + (p1.y-p2.y) * h;
    c.y = (p1.y+p2.y) * 0.5 + (p2.x-p1.x) * h;
    return true;
}

// Retorna la longitud del arco formado por una circunferencia c y un angulo central de thetad
double arcLength(double c, double theta) { return theta/360.0 * c; }

// Retorna la longitud de la cuerda formada por un circulo de area A y un angulo central de thetad
double chordLength(double A, double theta) { return sqrt(A * (1 - cos(DEG_to_RAD(theta)))); }

// Retorna el area de un sector de thetad del circulo
double sectorArea(double A, double theta) { return A/360.0 * A; }
```

6.5 3D point

```
struct Point {
    double x, y, z;
    Point() {}
    Point(double xx, double yy, double zz) { x = xx, y = yy, z = zz; }
    // Operadores escalares
    Point operator*(double f) { return Point(x * f, y * f, z * f); }
    Point operator/(double f) { return Point(x / f, y / f, z / f); }
    // Operadores P3
    Point operator-(Point p) { return Point(x - p.x, y - p.y, z - p.z); }
    Point operator+(Point p) { return Point(x + p.x, y + p.y, z + p.z); }
    Point operator%(Point p) { return Point(y * p.z - z * p.y, z * p.x - x * p.z, x * p.y - y * p.x);
        }  /// (|p||q|sin(ang))* normal
    double operator|(Point p) { return x * p.x + y * p.y + z * p.z; }
    // Comparadores
    bool operator==(Point p) { return tie(x, y, z) == tie(p.x, p.y, p.z); }
    bool operator!=(Point p) { return !operator==(p); }
    bool operator<(Point p) { return tie(x, y, z) < tie(p.x, p.y, p.z); }
};
    Point zero = Point(0, 0, 0);

    // BASICAS
    double sq(Point p) { return p | p; }
    double abs(Point p) { return sqrt(sq(p)); }
    Point unit(Point p) { return p / abs(p); }

    // ANGULOS
    double angle(Point p, Point q) {  ///[0, pi]
    double co = (p | q) / abs(p) / abs(q);
        return acos(max(-1.0, min(1.0, co)));
    }
    double small_angle(Point p, Point q) {  ///[0, pi/2]
        return acos(min(abs(p | q) / abs(p) / abs(q), 1.0))
    }

    // 3D - ORIENTACION
    double orient(Point p, Point q, Point r, Point s) { return (q - p) % (r - p) | (s - p); }
    bool coplanar(Point p, Point q, Point r, Point s) {
    return abs(orient(p, q, r, s)) < eps;
    }
    bool skew(Point p, Point q, Point r, Point s) { // Skew = No se intersecan ni son paralelas
    return abs(orient(p, q, r, s)) > eps;           // Lineas: PQ, RS
    }
    double orient_norm(Point p, Point q, Point r, Point n) {    // n := normal to a given plane PI n =
         normal al plano dado PI
    return (q - p) % (r - p) | n;                               // Equivalente al producto cruz 2D
        sobre PI (De la proyeccion ortogonal)
    }
```

7 Dynamic programming

7.1 Knapsack

```
// Time complexity (N * W)
#define MAXN 1010
int N, capacidad;
int peso[MAXN], valor[MAXN];
int dp[MAXN][MAXN];

int mochila (int i , int libre ) {
    if ( libre < 0) return -100000000;  //Metimos un objeto demasiado pesado
    if ( i == 0) return 0;              //Si ya no hay objetos, ya no ganamos nada
    if ( dp [ i ][ libre ] != -1) return dp [ i ][ libre ]; //El DP
    //Si tomamos el item
    int tomar = valor [ i ] + mochila ( i - 1 , libre - peso [ i ]) ;
    //Si no tomamos el item
    int noTomar = mochila ( i - 1 , libre ) ;
    //Devolvemos el maximo (y lo guardamos en la matriz dp)
    return ( dp [ i ][ libre ] = max ( tomar , noTomar ) ) ;
}
int main(){
    memset (dp, -1, sizeof (dp));
    cin>>N;
    cin>>capacidad;
    for(int i=0;i<N;i++){
        int p,v;
        cin>>p>>v;
        peso[i+1]=p;
        valor[i+1]=v;
    }
    int solucion = mochila(N, capacidad) ;
    cout<<solucion;
    return 0;
}
```

7.2 Knapsack 2

```cpp
#include <bits/stdc++.h>
using namespace std;
#define ENDL '\n'
#define FOR(x, b) for(int x = 0; x <b; x++)
#define FORR(x, a, b) for(int x = a; x >= b; x--)
#define _ ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
typedef long long ll;
typedef vector<int> vi;

//Knapsack 2 - Problem from at coder dp educational contest
//Classical knapsack with W up to 1e9
//Change the the definition of dp

const ll INF = 1e18L + 5;

int main(){ _
    int n, w;
    cin >> n >> w;

    vi value(n);
    vi weight(n);
    int sum_values = 0;

    FOR(i, n){
        cin >> weight[i] >> value[i];

        sum_values += value[i];
    }

    vector<ll> dp(sum_values + 1, INF);
    //dp[i] - minimum total weight of items with value i
    dp[0] = 0; //if there no value there's no weight
    FOR(i, n){ //iterate over all items n
        FORR(curr_value, sum_values - value[i], 0){//iterate from total values - value[i] to 0
        dp[curr_value + value[i]] = min(dp[curr_value + value[i]], dp[curr_value] + weight[i]);
        }
    }

    ll ans = 0;
    //search the answer on dp table
    FOR(i, sum_values + 1){
        if(dp[i] <= w){
        ans = max(ans, ll(i));
        }
    }

    cout << ans << ENDL;
    return 0;
}
```

7.3 Longest increasing subsequence

```cpp
// Time complexity O(n log k)
typedef vector<int> vi;

int n;  // tamanio del vector
vi A;   // Vector original
vi p;   // Vector de predecesor

void print_LIS(int i) {                                    // Rutina de backtracking
    if (p[i] == -1) { printf("%d", A[i]); return; }        // Caso base
    print_LIS(p[i]);                                       // backtrack
    printf(" %d", A[i]);
}

int main() {
    // Solucion O(n log k), n <= 200K
    int k = 0, lis_end = 0;
    vi L(n, 0), L_id(n, 0);
    p.assign(n, -1);

    for (int i = 0; i < n; ++i) {                          // O(n)
        int pos = lower_bound(L.begin(), L.begin()+k, A[i]) - L.begin();        // Busqueda binaria
        L[pos] = A[i];                                     // greedily overwrite this
        L_id[pos] = i;                                 // remember the index too
        p[i] = pos ? L_id[pos-1] : -1;                 // predecessor info
        if (pos == k) {                                // can extend LIS?
        k = pos+1;                                         // k = longer LIS by +1
        lis_end = i;                                       // keep best ending i
        }
    }

    printf("Final LIS is of length %d: ", k);
    print_LIS(lis_end); printf("\n");

    return 0;
}
```

7.4 2D sum

```cpp
/**
   Calcula rapidamente la suma de una submatriz dadas sus
   esquinas superior izquierda e inferior derecha (no inclusiva)
   Uso: SubMatrix<int> m(matrix);
   m.sum(0, 0, 2, 2); // 4 elementos superiores
   Tiempo: O(n * m) en preprocesamiento y O(1) por query
*/
template <class T>
struct SubMatrix {
    vector<vector<T>> p;
    SubMatrix(vector<vector<T>>& v) {
        int R = sz(v), C = sz(v[0]);
        p.assign(R + 1, vector<T>(C + 1));
        FOR(r, 0, R)
           FOR(c, 0, C)
               p[r + 1][c + 1] = v[r][c] + p[r][c + 1] + p[r + 1][c] - p[r][c];
    }
    T sum(int u, int l, int d, int r) {
        return p[d][r] - p[d][l] - p[u][r] + p[u][l];
    }
};
```

7.5 Max sum rectangle

```cpp
/*
    Algoritmo para encontrar la suma maxima de un rectangulo en una matriz 2D.
    Se utiliza el algoritmo de Kadane que permite encontrar la maxima suma de un sub arreglo.
    Time complexity: O(n^3)
*/
typedef long long ll;

ll kadane(vector<ll>& rowSum, int& start, int& end, int& n){
    ll maxSum = LLONG_MIN, maxTillNow = 0;
    int tempStart = 0;

    for(int i = 0; i < n; i++){
        maxTillNow += rowSum[i];
        if(maxTillNow > maxSum)
            maxSum = maxTillNow, start = tempStart, end = i;
        if(maxTillNow < 0)
            maxTillNow = 0, tempStart = i + 1;
    }

    return maxSum;
}

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

    vector<vector<ll>> mat(n, vector<ll>(m));

    // Lectura de la matriz

    ll maxSum = LLONG_MIN;
    int top, bottom, left, right, start, end;

    for(int l = 0; l < m; l++){
        vector<ll> rowSum(n, 0);
        for(int r = l; r < m; r++){
            for(int i = 0; i < n; i++)
                rowSum[i] += mat[i][r];
            ll sum = kadane(rowSum, start, end, n);
            if(sum > maxSum)
                maxSum = sum, left = l, right = r, top = start, bottom = end;
        }
    }

    printf("Top - left (%d, %d)\nBottom - right (%d, %d)\n", top, left, bottom, right);
    printf("Max sum = %lld\n", maxSum);

    return 0;
}
```

7.6 Game DP

```
/**
 * Descripcion:
 * Hay un set A = {a1, a2, ..., an} que consiste de n enteros positivos, Taro y Jiro jugaran el
      siguiente juego.
 * Inicialmente la pila tiene k piedras. Los 2 jugadores realizaran la siguiente
 * operacion alternandose, iniciando Taro:
 * - Elegir un elemento x en A, y remover exactamente x piedras de la pila.
 * Un jugador pierde si ya no puede hacer movimiento. Ambos juegan optimamente.
*/

constexpr int MAXN = 1e5 + 1;
vi moves;
int dp[MAXN];

int solve(int stones){
    if(stones == 0)         return 0;
    if(dp[stones] != -1)    return dp[stones];

    int ans = 0;

    for(auto &x : moves)
        if(stones >= x && !solve(stones - x)){
            ans = 1;
            break;
        }

    return dp[stones] = ans;
}

int main(){_
    int n, k;
    cin>>n>>k;

    moves.resize(n);

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

    memset(dp, -1, sizeof dp);

    cout<<(solve(k) ? "First" : "Second")<<'\n';

    return 0;
}
```

7.7 Range DP

```
/**
 * Dada un palo de madera de longitud de n unidades. El palo esta etiquetado desde 0 hasta n
 * Dado un arreglo de enteros cuts, donde cuts[i] denota una posicion donde debes hacer un corte
 * El orden de los cortes se puede cambiar, como se desee.
 * El coste de un corte es la longitud del palo a ser cortado, el coste total es la suma de todos los
      cortes.
 * Cuando cortas un palo, se divide en 2 palos mas pequenios.
 * Retornar el minimo coste de hacer todos los cortes.
*/

int dp[105][105];

int solve(int l, int r, vector<int>& cuts){
    if(l + 1 >= r)
        return 0;
    if(dp[l][r] != -1)
        return dp[l][r];

    int ans = 1e9+5;
    for(int i = l + 1; i < r; i++){
        ans = min(ans, cuts[r] - cuts[l] + solve(l, i, cuts) + solve(i, r, cuts));
        }
    return dp[l][r] = ans;
}

int minCost(int n, vector<int>& cuts) {
    memset(dp, -1, sizeof dp);
    cuts.insert(cuts.begin(), 0), cuts.push_back(n);
    sort(cuts.begin(), cuts.end());
    return solve(0, cuts.size() - 1, cuts);
}
```

7.8 Sum of digits in a range

```
#include <bits/stdc++.h>
//Pura gente del coach moy
using namespace std;
#define ENDL '\n'
#define all(s) begin(s), end(s)
#define rall(n) n.rbegin(), n.rend()
#define FOR(x, b) for(int x = 0; x <b; x++)
#define FORE(x, a, b) for(int x = a; x <= b; x++)
#define FORR(x, a, b) for(int x = a; x >= b; x--)
#define deb(x)      cerr << #x << " = " << x << '\n';
#define deb2(x, y)  cerr << #x << " = " << x << ", " << #y << " = " << y << '\n';
#define _ ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
const ll MOD = 1e9+7, INF=1e18;

//Dado un rango de l...r, contar la suma de los digitos de todos los numeros en ese rango

ll dp[20][180][2];

ll solve(string& num, int pos, int sum, bool tight){
    if(pos==0)   return sum;
    if(dp[pos][sum][tight]!=-1) return dp[pos][sum][tight];

    int ub=tight ? (num[num.length()-pos]-'0') : 9;
    ll ans=0;

    for(int dig=0;dig<=ub;dig++){
        ans+=solve(num,pos-1,sum+dig,(tight & (dig==ub)));
    }

    return dp[pos][sum][tight]=ans;
}

int main(){_
    ll ln,rn;
    cin>>ln>>rn;
    ln--;
    string l=to_string(ln),r=to_string(rn);
    memset(dp,-1,sizeof dp);
    ll lans=solve(l,l.length(),0,1);
    memset(dp,-1,sizeof dp);
    ll rans=solve(r,r.length(),0,1);
    cout<<rans-lans<<ENDL;
    return 0;
}
```

7.9 Enigma regional 2017

```
#include <bits/stdc++.h>
//Pura gente del coach moy
using namespace std;
#define ENDL '\n'
#define all(s) begin(s), end(s)
#define rall(n) n.rbegin(), n.rend()
#define FOR(x, b) for(int x = 0; x <b; x++)
#define FORE(x, a, b) for(int x = a; x <= b; x++)
#define FORR(x, a, b) for(int x = a; x >= b; x--)
#define deb(x)      cerr << #x << " = " << x << '\n';
#define deb2(x, y)  cerr << #x << " = " << x << ", " << #y << " = " << y << '\n';
#define _ ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
const ll MOD = 1e9+7, INF=1e18;

string num;
int n;
int dp[1001][1001];

bool solve(int pos, int res){
    if(pos==0)              return res==0;
    if(dp[pos][res]!=-1)    return dp[pos][res];

    bool ans=false;

    if(num[num.length()-pos]!='?'){
        int dig=num[num.length()-pos]-'0';
        ans|=solve(pos-1,(res*10+dig)%n);
```

```
    }else{
        for(int dig=0;dig<=9;dig++){
            if(pos==0&&dig==0)  continue;
            ans|=solve(pos-1,(res*10+dig)%n);
        }
    }
    return dp[pos][res]=ans;
}

int main(){_
    memset(dp,-1,sizeof dp);
    cin>>num>>n;
    bool posible=solve(num.length(),0);
    if(!posible){
        cout<<'*'<<ENDL;
        return 0;
    }
    int mod=0;
    FOR(i,num.length()){
        if(num[i]!='?'){
            cout<<num[i];
            mod=(mod*10+(num[i]-'0'))%n;
            continue;
        }
        FORE(j,i==0,9){
            if(solve(num.length()-i-1,(mod*10+j)%n)){
                mod=(mod*10+j)%n;
                cout<<j;
                break;
            }
        }
    }
    cout<<ENDL;
    return 0;
}
```

7.10 Little elephant and T shirts - CodeChef

```
#include <bits/stdc++.h>
//Pura gente del coach moy
using namespace std;
#define ENDL '\n'
#define all(s) begin(s), end(s)
#define rall(n) n.rbegin(), n.rend()
#define FOR(x, b) for(int x = 0; x <b; x++)
#define FORE(x, a, b) for(int x = a; x <= b; x++)
#define FORR(x, a, b) for(int x = a; x >= b; x--)
#define deb(x)      cerr << #x << " = " << x << '\n';
#define deb2(x, y)  cerr << #x << " = " << x << ", " << #y << " = " << y << '\n';
#define _ ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
const ll MOD = 1e9+7, INF=1e18;

//Problema: Little Elephant and T-Shirts  --  CodeChef
//Descripcion: Hay n personas que tienen ciertas playeras las cuales cuentan con un cierto ID desde 1
     a 100
//En la entrada se dan cuantas personas hay y las playeras que tiene cada una de estas personas
//Se pide encontrar el numero de maneras en las que se pueden distribuir las personas y las playeras,
     de tal manera que, no haya 2 personas
//vistiendo la misma playera en ese conjunto. Al final imprimir modulo 1e9+7

//n, matriz para saber si una persona tiene una playera y matriz dp
int n;
bool tshirts[11][101];
ll dp[101][1<<11];

//Funcion para resolver el problema
ll solve(int shirt, int mask){
    //Si ya se le asigno a cada persona una playera, se retorna 1
    if(mask==((1<<n)-1))        return 1;

    //Si ya se recorrieron todas las playeras, se retorna 0
    if(shirt==100)              return 0;

    //Si ya se calculo anteriormente, se retorna lo almacenado en la dp
    if(dp[shirt][mask]!=-1)     return dp[shirt][mask];

    ll ans=0;

    //Para cada persona
    FOR(p,n){
        //Se verifica si esa persona aun no tiene una playera y si esta persona cuenta con la playera
             del parametro de la funcion
        if(!(mask&(1<<p))&&tshirts[p][shirt]){
            //Si cuenta con ella, se continua con la siguiente playera y se le asigna playera a la
                 persona p
            ans=(ans+solve(shirt+1,mask|(1<<p)))%MOD;
        }
    }
    //Tambien se calcula para en caso de no asignar esta playera a la persona y asignarle
         posteriormente otra de con las que cuenta
    ans=(ans+solve(shirt+1,mask))%MOD;

    return dp[shirt][mask]=ans;
}

int main(){_
    int t;
    cin>>t;
    while(t--){
        memset(dp,-1,sizeof dp);
        memset(tshirts,0,sizeof tshirts);
        cin>>n;
        string s;
        cin.ignore();
        FOR(i,n){
            getline(cin,s);
            stringstream in(s);
            int ts;
            while(in>>ts){
                tshirts[i][--ts]=1;
            }
        }
        cout<<solve(0,0)<<ENDL;
    }
    return 0;
}
```

7.11 O-Matching AtCoder

```
#include <bits/stdc++.h>
//Pura gente del coach moy
using namespace std;
#define ENDL '\n'
#define all(s) begin(s), end(s)
#define rall(n) n.rbegin(), n.rend()
#define FOR(x, b) for(int x = 0; x <b; x++)
#define FORE(x, a, b) for(int x = a; x <= b; x++)
#define FORR(x, a, b) for(int x = a; x >= b; x--)
#define deb(x)      cerr << #x << " = " << x << '\n';
#define deb2(x, y)  cerr << #x << " = " << x << ", " << #y << " = " << y << '\n';
#define _ ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
const ll MOD = 1e9+7, INF=1e18;

//Problema: O-Matching  AtCoder
//Descripcion: Te dan una matriz con las compatibilidades de parejas, donde i son los hombres y j son
     las mujeres,
//por lo tanto, a_ij indica si son compatibles con un 1, o si no lo son con un 0
//El problema pide el numero de parejas distintas que se pueden formar. Se aplica modulo 1e9+7 al
     resultado

int n;
vi adj[21];
ll dp[21][(1<<21)-1];

ll solve(int idx, int mask){
    //Si se llega a n, significa que todas las parejas han sido asignadas
    if(idx==n)              return 1;
    // if(mask==((1<<n)-1))    return 1;
    if(dp[idx][mask]!=-1)   return dp[idx][mask];

    ll ans=0;

    for(int i:adj[idx]){
        if(((mask&(1<<i))==0)){
            ans=(ans+solve(idx+1,mask|(1<<i)))%MOD;
        }
    }

    return dp[idx][mask]=ans;
}

int main(){_
    memset(dp,-1,sizeof dp);
    cin>>n;
```

```
int match;
FOR(i,n){
    FOR(j,n){
        cin>>match;
        if(match)
            adj[i].push_back(j);
    }
```

```
    }
    cout<<solve(0,0)<<ENDL;
    return 0;
}
```