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;                                 // Primero se aplica modulo a la base
    ll ans = 1;                             // Caso base p = 0
    while(p){
        if(p & 1) ans = mod(ans * b, m);    // (ans * b) % m, si p es impar
        b = mod(b * b, m);                  // (b ^ 2) % m
        p >>= 1;                            // p /= 2
    }
    return ans;                             // Retorna el resultado
}

int extEuclid(int a, int b, int &x, int &y) {   // Pasa x e y por referencia
    int xx = y = 0;
    int yy = x = 1;
    while (b) {                                 // Repetir hasta que b == 0
        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

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

ll _sieve_size;
bitset<10000010> bs;                          // 10^7 es el limite
vll p;                                        // Vector de primos

void sieve(ll upperbound) {                   // Rango = [0..upperbound]
    _sieve_size = upperbound+1;               // Para incluir el upperbound
    bs.set();                                 // Asigna 1 en todas las posiciones
    bs[0] = bs[1] = 0;                        // Excepto indice 0 y 1
    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);                       // Agrega el primo i a la lista
    }
}

bool isPrime(ll N) {                          // Prueba de primalidad suficientemente buena
    if (N < _sieve_size) return bs[N];        // O(1) para primos pequenios
    for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
        if (N%p[i] == 0)
        return false;
    return true;                              // Lento si N es un primo grande
}   // Nota: Solo se garantiza su funcionamiento para N <= (ultimo primo en vll p)^2

vll primeFactors(ll N) {                      // Pre-condicion, N >= 1
    vll factors;
    for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
        while (N%p[i] == 0) {                 // Se encontro un primo para N
            N /= p[i];                        // Se remueve de N
            factors.push_back(p[i]);
    }
    if (N != 1) factors.push_back(N);         // La N que queda es un primo
    return factors;
}

int numPF(ll 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) {
    int ans = 0;
    for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i) {
        if (N%p[i] == 0) ++ans;               // Cuenta este factor primo
        while (N%p[i] == 0) N /= p[i];        // Solo una vez
    }
    if (N != 1) ++ans;
    return ans;
}

ll sumPF(ll 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) {
    int ans = 1;                              // Inicia de 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) {
    ll ans = 1;                               // Inicia de 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) {
    ll ans = N;                                   // Inicia de 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];         // Cuenta los unicos
        while (N%p[i] == 0) N /= p[i];            // Factor primo
    }
    if (N != 1) ans -= ans/N;                     // Ultimo factor
    return ans;
}
```

2.3 Bit operations

```cpp
// 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

```cpp
//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

```python
# 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)]
```

```python
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

```cpp
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]; }
```

3 Data structure

3.1 Union find (DSU)

```cpp
// Union-Find Disjoint Sets escrito en POO, usando las heuristicas de compresion de camino y union por
       rango
typedef vector<int> vi;

class UnionFind {                                  // Estilo POO
 private:
    vi p, rank, setSize;                           // vi p es la parte clave
    int numSets;
 public:
    UnionFind(int N) {
        p.assign(N, 0); for (int i = 0; i < N; ++i) p[i] = i;
        rank.assign(N, 0);                         // Aceleracion opcional
        setSize.assign(N, 1);                      // Caracteristica opcional
        numSets = N;                               // Caracteristica opcional
    }
    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; }              // Opcional
    int sizeOfSet(int i) { return setSize[findSet(i)]; }    // Opcional
    void unionSet(int i, int j) {
        if (isSameSet(i, j)) return;               // i y j estan en el mismo set
        int x = findSet(i), y = findSet(j);        // Encuentra los representantes de ambos
        if (rank[x] > rank[y]) swap(x, y);         // Para mantener x mas pequenio que y
        p[x] = y;                                  // Set x bajo y
        if (rank[x] == rank[y]) ++rank[y];         // Acelaracion opcional
        setSize[y] += setSize[x];                  // Combina los tamanios de los sets en y
        --numSets;                                 // Una union reduce el numSets
    }
};
```

3.2 Union find (DSU) with rollback

```cpp
// 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

```cpp
// 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

```cpp
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

```cpp
#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

```cpp
/* Implementacion de segment tree para obtener la suma en un rango, pero es posible usar cualquier
operacion conmutativa como la multiplicacion, XOR, AND, OR, MIN, MAX, etc.*/
typedef vector<int> vi;

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

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

    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;
        int q1 = query(l(index), start, mid, i, j);
        int q2 = query(r(index), mid + 1, end, i, j);

        return q1 + q2;
    }

    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) {} // Constructor de st sin valores

    SegmentTree(const vi &initialArr) : SegmentTree((int)initialArr.size()) { // Constructor de st con
         arreglo inicial
        arr = initialArr;
        build(1, 0, n - 1);
    }

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

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

3.7 Segment tree with lazy propagation

```cpp
// Implementacion de segment tree con lazy propagation
typedef vector<int> vi;

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

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

    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 update(int index, int start, int end, int i, int j, int val) {
        propagate(index, start, end);
        if ((end < i) || (start > j))
            return;

        if (start >= i && end <= j) {
            st[index] += (end - start + 1) * val;
            if (start != end) {
                lazy[l(index)] += val;
                lazy[r(index)] += val;
            }
            return;
        }
        int mid = (start + end) / 2;
        update(l(index), start, mid, i, j, val);
        update(r(index), mid + 1, end, i, j, val);

        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;
        int q1 = query(l(index), start, mid, i, j);
        int q2 = query(r(index), mid + 1, end, i, j);

        return (q1 + q2);
    }

  public:
    LazySegmentTree(int sz) : n(sz), st(4 * n), lazy(4 * n) {} // Constructor de st sin valores

    LazySegmentTree(const vi &initialA) : LazySegmentTree((int)initialA.size()) { // Constructor de st
          con arreglo inicial
        A = initialA;
        build(1, 0, n - 1);
    }

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

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

3.8 Segment tree RMQ with lazy propagation

```cpp
// Implementacion de segment tree lazy para obtener una RMQ
typedef vector<int> vi;

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

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

    int conquer(int a, int b) {
        if (a == -1)
            return b;
        if (b == -1)
            return a;
        return min(a, b); // RMQ - Cambiar esta linea para modificar la operacion del st
    }

    void build(int p, int L, int R) { // O(n)
        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)                               // chechar que no es una hoja
                lazy[l(p)] = lazy[r(p)] = lazy[p]; // propagar hacia abajo
            else
                A[L] = lazy[p];
            lazy[p] = -1;
        }
    }

    int query(int p, int L, int R, int i, int j) { // O(log n)
        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) { // O(log n)
        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) {} // Constructor de st sin valores

    LazyRMQ(const vi &initialA) : LazyRMQ((int)initialA.size()) { // Constructor de st con arreglo
          inicial
        A = initialA;
        build(1, 0, n - 1);
    }

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

    int query(int i, int j) { return query(1, 0, n - 1, i, j); }
};

int main(){
    // Implementacion
    vi A = {18, 17, 13, 19, 15, 11, 20, 99};
    LazyRMQ st(A);

    st.query(1, 3); // RMQ(1,3);

    st.update(5, 5, 77); // actualiza A[5] a 77
```

```
    st.update(0, 3, 30); // actualiza A[0..3] a 30
    return 0;
}
```

3.9 Segment tree of DSU with rollback

```
// 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;
}
```

3.10 Sparse table

```
// Time complexity: Build O(n log n), Query O(1)
typedef vector<int> vi;

class SparseTable {
private:
    vi A, P2, L2;       // Vector base, potencias de 2 y logaritmos base 2
    vector<vi> SpT;     // La Sparse Table
public:
    SparseTable() {}    // Constructor default

    SparseTable(vi &initialA) : A(initialA){        // Rutina de preprocesamiento
        int n = (int)A.size(), L2_n = (int)log2(n)+1;
        P2.assign(L2_n, 0), L2.assign(1<<L2_n, 0);
        for (int i = 0; i <= L2_n; ++i) {
            P2[i] = (1<<i);                         // Para acelerar 2^i
            L2[(1<<i)] = i;                         // Para acelerar log_2(i)
        }
        for (int i = 2; i < P2[L2_n]; ++i)
            if (L2[i] == 0)
                L2[i] = L2[i-1];                    // Para llenar los vacios

        // Inicializacion
        SpT = vector<vi>(L2[n]+1, vi(n));
        for (int j = 0; j < n; ++j)
            SpT[0][j] = j;                          // RMQ del sub array [j..j]

        // Ciclos con complejidad total O(n log n)
        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) {
        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
    }
};
```

3.11 Order statistics tree

```cpp
// 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)
}
```

3.12 Binary search tree

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

    while(!pq.empty()){
        auto [d, u] = pq.top(); pq.pop();
        if(d > dist[u]) continue;
        for(auto &[v, w] : adj[u]){
            if(dist[u]+w >= dist[v])  continue;
            dist[v] = dist[u] + 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 pair<int, int> ii;
typedef vector<int> vi;
enum { UNVISITED = -1, VISITED = -2 };

vector<vi> AL;
vi dfs_num, ts;

void toposort(int u) {
    dfs_num[u] = VISITED;
    for (auto &v : AL[u])
        if (dfs_num[v] == UNVISITED)
            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());

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

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

```
// 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 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);
        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;
    }
};
```

4.13 Lowest common ancestor

```
// 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);
}
```

4.14 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;

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 bm, 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
int maxMatch = 0;   // tamanio del max matching

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 main() {
    // Lectura del grafo

    // Heuristica que consiste en tomar inicialmente cualquier matching valido y ejecutar el algoritmo
            a partir de ahi
    mt.assign(k, -1);
    vector<bool> used1(n, false);
    for (int v = 0; v < n; ++v) {
        for (int to : adj[v]) {
            if (mt[to] == -1) {
                mt[to] = v;
                used1[v] = true;
                break;
            }
        }
    }

    // Ejecucion del algoritmo
    for (int v = 0; v < n; ++v) {
        if (used1[v])
            continue;
        used.assign(n, false);
        try_kuhn(v);
    }

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

    printf("Max match = %d\n", maxMatch);
}
```

5 Strings

5.1 Knuth Morris Pratt (KMP)

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

vi kmpPreprocess(string &P) {   // Preprocesamiento
    int m = P.size();           // m = |P|
    vi b(m + 1);                // b = Back table
    int i = 0, j = -1; b[0] = -1;                       // Valores iniciales
    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;                                   // Valores iniciales
    int n = T.size(), m = P.size();                     // n = |T|, m = |P|
    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;    // Retorna el numero de coincidencias del patron en la cadena
}

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

```
// 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

```
// 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;
    }
};
```

6 Dynamic programming

6.1 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;
}
```

6.2 Knapsack 2

```
#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;
}
```

6.3 Longest increasing subsequence

```
// 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;
}
```

6.4 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;
}
```

6.5 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;
}
```

6.6 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;
}
```

6.7 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++)
```

```cpp
#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;
}
```