AC2++ ICPC Team Notebook

Contents

1 Templates

1.1 Plantilla C++

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

using ll = long long;
using ull = unsigned long long;

using pi = pair<int, int>;
using pl = pair<ll, ll>;
using pd = pair<double, double>;

using vi = vector<int>;
using vb = vector<bool>;
using vl = vector<ll>;
using vd = vector<double>;
using vs = vector<string>;
using vpi = vector<pi>;
using vpl = vector<pl>;
using vpd = vector<pd>;

// pairs
#define mp make_pair
#define f first
#define s second

// vectors
#define sz(x) int((x).size())
#define bg(x) begin(x)
#define all(x) bg(x), end(x)
#define rall(x) x.rbegin(), x.rend()
#define ins insert
#define ft front()
#define bk back()
#define pb push_back
#define eb emplace_back
#define lb lower_bound
#define ub upper_bound
#define tcT template <class T
tcT > int lwb(vector<T> &a, const T &b) { return int(lb(all(a), b) - bg(a)); }

// loops
#define FOR(i, a, b) for (int i = (a); i < (b); ++i)
#define F0R(i, a) FOR(i, 0, a)
#define ROF(i, a, b) for (int i = (a)-1; i >= (b); --i)
#define R0F(i, a) ROF(i, a, 0)

#define ENDL '\n'
#define LSOne(S) ((S) & -(S))

const int MOD = 1e9 + 7;
const int MAXN = 1e5 + 5;
const int INF = 1 << 28;
const ll LLINF = 1e18;
const int dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1}; // abajo, derecha,
    arriba, izquierda

template <class T>
using pqg = priority_queue<T, vector<T>, greater<T>>;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(nullptr);

    return 0;
}
```

1.2 Plantilla Phyton

```python
import sys
import math
import bisect
from sys import stdin, stdout
from math import gcd, floor, sqrt, log
from collections import defaultdict as dd
from bisect import bisect_left as bl, bisect_right as br

sys.setrecursionlimit(100000000)


def inp(): return int(input())
def strng(): return input().strip()


def jn(x, l): return x.join(map(str, l))
def strl(): return list(input().strip())


def mul(): return map(int, input().strip().split())
def mulf(): return map(float, input().strip().split())
def seq(): return list(map(int, input().strip().split()))


def ceil(x): return int(x) if (x == int(x)) else int(x)+1
def ceildiv(x, d): return x//d if (x % d == 0) else x//d+1


def flush(): return stdout.flush()
def stdstr(): return stdin.readline()
def stdint(): return int(stdin.readline())


def stdpr(x): return stdout.write(str(x))


mod = 1000000007
```

2 Data Structures

2.1 Trie

```
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) {
        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) {
        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) {
        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) {
        TrieNode *curr = root;
        for (char c : prefix) {
            if (curr->children.find(c) == curr->children.end()) {
                return 0;
            }
            curr = curr->children[c];
        }
        return curr->numPrefix;
    }
};
```

2.2 Fenwick Tree

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

class FenwickTree {
  private:
    vll ft;

  public:
    FenwickTree(int m) { ft.assign(m + 1, 0); } // Constructor de ft vacio

    void build(const vll &f) {
        int m = (int)f.size() - 1;
        ft.assign(m + 1, 0);
        FOR(i, 1, m + 1) {
            ft[i] += f[i];
            if (i + LSOne(i) <= m)
                ft[i + LSOne(i)] += ft[i];
        }
    }

    FenwickTree(const vll &f) { build(f); } // Constructor de ft basado en otro
        ft

    FenwickTree(int m, const vi &s) { // Constructor de ft basado en un vector
        int
        vll f(m + 1, 0);
        FOR(i, (int)s.size()) {
            ++f[s[i]];
        }
        build(f);
    }

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

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

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

    int select(ll k) {
        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 { // Arbol de Fenwick de consulta de punto y actualizacion de rango
  private:
    FenwickTree ft;

  public:
    RUPQ(int m) : ft(FenwickTree(m)) {}

    void range_update(int ui, int uj, ll v) {
        ft.update(ui, v);
        ft.update(uj + 1, -v);
    }

    ll point_query(int i) {
        return ft.query(i);
    }
}

class RURQ { // Arbol de Fenwick de consulta de rango y actualizacion de rango
  private:
    RUPQ(int m) : rupq(RUPQ(m)), purq(FenwickTree(m)) {}

    void range_update(int ui, int uj, ll v) {
        rupq.range_update(ui, uj, v);
        purq.update(ui, v * (ui - 1));
        purq.update(uj + 1, -v * uj);
    }

    ll query(int j) {
        return ruqp.point_query(j) * j -
               purq.query(j);
    }

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

// Implementacion
vll f = {0, 0, 1, 0, 1, 2, 3, 2, 1, 1, 0}; // index 0 siempre sera 0
FenwickTree ft(f);
printf("%lld\n", ft.rsq(1, 6));  // 7 => ft[6]+ft[4] = 5+2 = 7
printf("%d\n", ft.select(7));    // index 6, query(1, 6) == 7, el cual es >= 7
ft.update(5, 1);                 // update {0,0,1,0,2,2,3,2,1,1,0}
printf("%lld\n", ft.rsq(1, 10)); // 12
printf("=====\n");
RUPQ rupq(10);
RURQ rurq(10);
rupq.range_update(2, 9, 7); // indices en [2, 3, .., 9] actualizados a +7
rurq.range_update(2, 9, 7);
rupq.range_update(6, 7, 3); // indices 6&7 son actualizados a +3 (10)
rurq.range_update(6, 7, 3);
// idx = 0 (unused) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10
// val = -          | 0 | 7 | 7 | 7 | 7 |10 |10 | 7 | 7 | 0
for (int i = 1; i <= 10; i++)
    printf("%d -> %lld\n", i, rupq.point_query(i));
printf("RSQ(1, 10) = %lld\n", rurq.rsq(1, 10)); // 62
printf("RSQ(6, 7) = %lld\n", rurq.rsq(6, 7));   // 20
```

2.3 Binary Indexed Tree

```
const int MAXN = 1e5 + 5;
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 update(int index, int val) {
    while (index <= n) {
        bit[index] += val;
        index += index & (-index);
    }
}
```

2.4 Order Statistics Tree

```
#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 n = 9;
int A[] = {2, 4, 7, 10, 15, 23, 50, 65, 71}; // as in Chapter 2
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)
```

2.5 Segment Tree

```
/* Implementado para RSQ, pero es posible usar cualquier operacion conmutativa
   (no importa el orden) como la multiplicacion, XOR, OR, AND, MIN, MAX, etc. */
```

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

    int l(int p) { return (p << 1) + 1; }      // Ir al hijo izquierdo
    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] = 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) {} // 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(0, 0, n - 1, i, val); }

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

2.6 Lazy Segment Tree

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

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

    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;

        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) {} // 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(0, 0, n - 1, i, j, val); }

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

2.7 Lazy Range Min/Max Query

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

    int l(int p) { return (p << 1) + 1; }    // Ir al hijo izquierdo
    int r(int p) { return (p << 1) + 2; }    // 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)                                  // Checar que no sea 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(0, 0, n - 1, i, j, val); }

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

2.8 Sparse Segment Tree

```
// Cuando el rango (0, n - 1) es muy largo y sea muy pesado guardar un arreglo
     de tamanio n * 4
// se puede utilizar un Sparse S. T., el cual usa punteros para los 2 hijos de
     cada nodo, siendo creados
// solo cuando se necesitan

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) { // add v
                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; F0R(i,2) if (c[i]) val += c[i]->val;
        }
        T query(int lo, int hi, int L = 0, int R = SZ-1) { // query sum of
             segment
                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 Math

3.1 Numeros Primos

```cpp
using ll = long long;
using vl = vector<ll>;

const int MAXN = 1e6 + 5;

ll sieve_size;
vl primes;

// O(n * sqrt(n))
void sieve(ll n) {
    sieve_size = n + 1;
    // faster than bitset
    bool is_prime[sieve_size];

    // primo hasta que se demuestre lo contrario B-)
    for (int i = 0; i < sieve_size; i++)
        is_prime[i] = 1;

    is_prime[0] = is_prime[1] = 0;
    for (ll p = 2; p < sieve_size; p++) {
        if (is_prime[p]) {
            for (ll i = p * p; i < sieve_size; i += p)
                is_prime[i] = 0;
            primes.push_back(p);
        }
    }
}

// O(sqrt(n))
bool isPrime(ll n) {
    for (ll i = 0; i * i <= n; i++)
        if (n % i == 0)
            return false;
    return true;
}

// O(sqrt(n))
vl primeFactors(ll n) {
    vl factors;
    ll idx = 2;
    while (n != 1) {
        while (n % idx == 0) {
            n /= idx;
            factors.pb(idx);
        }
        idx++;
    }
    return 0;
}

// Contar el numero de factores primos del entero N
// O(sqrt(n))
int numPF(ll n) {
    int ans = 0;
    for (int i = 0; (i < (int)primes.size()) && (primes[i] * primes[i] <= n); ++
         i)
        while (n % primes[i] == 0) {
            n /= primes[i];
            ans++;
        }
    return ans + (n != 1);
}
```

3.2 Operaciones con Bits

```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);
```

3.3 Formulas Rapidas

```cpp
/*
    En C++14 se puede utilizar el metodo de algorithm
    __gcd(m,n)

    A partir de C++17 se puede utilizar el metodo de numeric
    gcd(m,n)
    lcm(m,n)
*/
int gcd(int a, int b) { return (b ? gcd(b, a % b) : a); }
```

```cpp
ll lcm(int a, int b) { return ((a * b) / gcd(a, b)); }

// (a^b) mod m, O(log n)
ll fastpow(ll a, ll b, ll m) {
    ll res = 1;
    a %= m;
    while (b) {
        if (b & 1)
            res = (res * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
    return res;
}

// Coeficientes binomiales (Combinatoria) - O(n)
ll C(int n, int k) { // O(log p)
    if (n < k)
        return 0;
    return (((fact[n] * modInverse(fact[k])) % p) * modInverse(fact[n - k])) % p
        ;
}

// Codigo para calcular (a^-1)%m (si es que existe), O(log n)

// Si m es primo
int modInverse(int b, int m) { return fastpow(b, m - 2, m) % m; }

// Si m NO es primo
using iii = tuple<int, int, int>;

iii extendedGCD(int a, int b) {
    if (b == 0)
        return iii({a, 1, 0});
    auto [d, x, y] = extendedGCD(b, a % b);
    return {d, y, x - a / b * y};
}

int modInverse(int a, int m) {
    auto [d, x, y] = extendedGCD(a, m);
    if (d > 1)
        return 0; // Si no existe
    return (x + m) % m;
}
```

3.4 Numeros Catalan

```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)]


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]))
```

3.5 Linear Diophantine

```cpp
/*
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.
*/
int extended_gcd(int a, int b,int &x, int &y){
    if(b == 0){
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = extended_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

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

    // 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 << ENDL;
    }
}
```

4 Strings

4.1 Knuth-Morris-Prath

```
/*
Encuentra todas las ocurrencias del patron string p en el
texto string t. Tiempo de ejecucion es O(n + m), donde n y m
son las longitudes de p y t, respectivamente.
*/
void buildPi(string &p, vi& pi) {
    pi = vi(sz(p));
    int k = -2;
    F0R (i, sz(p)) {
        while (k >= -1 && p[k + 1] != p[i])
            k = (k == -1) ? -2 : pi[k];
        pi[i] = ++k;
    }
}

int KMP(string &t, string& p) {
    vi pi;
    buildPi(p, pi);
    int k = -1;
    F0R (i, sz(t)) {
        while (k >= -1 && p[k + 1] != t[i])
            k = (k == -1) ? -2 : pi[k];
        k++;
        if (k == sz(p) - 1) {
            // p hace match con t[i-m+1, ..., i]
            cout << "matched at index " << i - k << ": ";
            cout << t.substr(i - k, sz(p)) << ENDL;
            k = (k == -1) ? -2 : pi[k];
        }
    }
    return 0;
}

int main() {
    string a = "AABAACAADAABAABA", b = "AABA";
    KMP(a, b);
    //Matches esperados en: 0, 9, 12
}
```

5 Dynamic Programming

5.1 Problema de la mochila

```
/*
Algoritmo: Problema de la mochila
Tipo: DP
Complejidad: O(n^2)

Problema:
Se cuenta con una coleccion de N objetos donde cada uno tiene un peso y un valor
    ,
y una mochila a la que le caben C unidades de peso.
Escribe un programa que calcule la maxima suma de valores que se puede lograr
     guardando
objetos en la mochila sin superar su capacidad de peso.
*/

ii objeto[MAXN]; // {peso, valor}
int dp[MAXN][MAXN];
int n;

int mochila(int i, int libre) {
    if (libre < 0)
        return -INF;
    if (i == n)
        return 0;
    if (dp[i][libre] != -1)
        return dp[i][libre];

    int opcion1 = mochila(i + 1, libre);
    int opcion2 = objeto[i].second + mochila(i + 1, libre - objeto[i].first);

    return (dp[i][libre] = max(opcion1, opcion2));
}

/*
Ejemplo de uso:

memset(dp,-1,sizeof(dp));
cout << mochila(0,pmax);
*/
```

5.2 Longest Increasing Subsequence (LIS)

```
#define FOR(i, n) for(int i = 0; i < n; i++)
const int MAXN = 2e4;

//Si no se necesita imprimir la LIS por completo, eliminar p.
int p[MAXN], nums[MAXN];

void print_LIS(int i) {                              // backtracking routine
    if (p[i] == -1) {
        cout << A[i];
        return;
    }                                                // base case
    print_LIS(p[i]);                                 // backtrack
    cout << nums[i];
}

int main() {
    int n;
    cin >> n;
    FOR(i, n)
        cin >> nums[i];
    int lis_sz = 0, lis_end = 0;
    int L[n], L_id[n];
    FOR (i, n) {
        L[i] = L_id[i] = 0;
        p[i] = -1;
    }

    FOR (i, n) { // O(n)
        int pos = lower_bound(L, L + lis_sz, nums[i]) - L;
        L[pos] = nums[i];       // greedily overwrite this
        L_id[pos] = i;          // remember the index too

        p[i] = pos ? L_id[pos-1] : -1; // predecessor info

        if (pos == lis_sz) {  // can extend LIS?
            lis_sz = pos + 1; // k = longer LIS by +1
            lis_end = i;      // keep best ending i
        }
    }
    cout << lis_sz << ENDL;
}
```

5.3 Problema del viajero

```
/*
Problema de agente viajero TSP
El problema del agente viajero consiste en encontrar un recorrido
que visite todos los vertices de un grafo, sin repetir y a costo minimo.
Escribe un programa que resuelva la version del problema en la que el agente
viajero puede comenzar en cualquier vertice y no necesita regresar al vertice
     inicial.
*/

int dist[MAXN][MAXN];
int dp[MAXN][1 << (MAXN + 1)];
int n;

int solve(int idx, int mask) {
    if (mask == (1 << n) - 1)
        return 0;
    if (dp[idx][mask] != -1)
        return dp[idx][mask];

    int ret = INF;
    FOR(i, n) {
        if ((mask & (1 << i)) == 0) {
            int newMask = mask | (1 << i);
            ret = min(ret, solve(i, newMask) + dist[idx][i]);
        }
    }
    return dp[idx][mask] = ret;
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(nullptr);

    cin >> n;
    memset(dp, -1, sizeof(dp));
    FOR(i, n) {
        FOR(j, n) {
            cin >> dist[i][j];
        }
    }
    int ans = INF;
    FOR(i, n) {
```

```
        ans = min(ans, solve(i, (1 << (i))));
    }
    cout << ans;
    return 0;
}
```

5.4 Dp con Digitos

```
/* Enunciado.
   Dada una cadena s de la forma ?????d???d?d?? o ?d????d?d, donde d es un
        digito cualquiera
   asignar a los caracteres ? algun digito, para formar el numero mas pequenio
        posible
   que ademas sea divisible por D y no tenga ceros a la izquierda
*/

const int MAXN = 1e5 + 5;

string s; // cadena
int D;
stack<int> st;

bool dp[MAXN][MAXN]; // He pasado por aqui?
bool solve(int pos, int residuo) {
    if (dp[pos][residuo])
        return false;
    if (pos == s.length())
        return residuo == 0;

    if (s[pos] == '?') {
        for (int k = (pos == 0); k <= 9; k++) {
            if (solve(pos + 1, (residuo * 10 + k) % D)) {
                st.push(k);
                return true;
            }
        }
    } else {
        if (solve(pos + 1, (residuo * 10 + (s[pos] - '0')) % D)) {
            st.push(s[pos] - '0');
            return true;
        }
    }
    dp[pos][residuo] = true;
    return false;
}

int main() {
    cin >> s >> D;

    if (solve(0, 0)) {
        while (!st.empty()) {
            cout << st.top();
            st.pop();
        }
        cout << ENDL;
    } else
        cout << "*\n";

    return 0;
}
```

5.5 Tecnica con pila

```
#define FOR(i, a) for (int i = 0; i < n; i++)

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(nullptr);

    int n = 12, heights[n] = {1, 8, 4, 9, 9, 10, 3, 2, 4, 8, 1, 13}, leftSmaller
        [n];
    stack<int> st;
    FOR (i, n) {
        while (!st.empty() && heights[st.top()] > heights[i])
            st.pop();
        if (st.empty())
            leftSmaller[i] = -1;
        else
            leftSmaller[i] = st.top();
        st.push(i);
    }
    //Ahora leftSmaller[i] tiene el indice del elemento menor mas cercano a la
        izquierda de heights[i]
}
```

5.6 Tecnica con deque

```
// Retorna un vector b en donde b[i] es igual a
// max(a[i], a[i + 1], a[i + 2], ..., a[i + p - 1]);

vi max_1D(vi a, int p) {
  vi b(sz(a) - p + 1);
  deque<int> dq;
  FOR (i, sz(a)) {
        while (sz(dq) && a[dq.bk] <= a[i])
            dq.pop_back();
        dq.pb(i);
        if (dq.ft <= i - p)
            dq.pop_front();

        if (i + 1 >= p)
            b[i + 1 - p] = a[dq.ft];
  }
}
```

5.7 Suma en 2D

```
template <class T>
struct SubMatrix {
        vector<vector<T>> p; // Matriz modificada
        SubMatrix(vector<vector<T>>& v) {
                int R = sz(v), C = sz(v[0]);
                p.assign(R+1, vector<T>(C+1));
                rep(r,0,R) rep(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];
        }
};
```

6 Graphs

6.1 Recorrido BFS y DFS

```cpp
const int MAXN = 1e5 + 5;

vi grafo[MAXN];
int dist[MAXN]; //Desde un nodo elegido por nosotros a cualquier otro
//Importante inicializar en -1 para saber si no se ha visitado

void bfs(int node) {
    queue<int> q;
    q.push(node);
    dist[node] = 0;
    while (!q.empty()) {
        int s = q.front();
        q.pop();
        for (auto u : grafo[s]) {
            if (dist[u] == -1) { //Si no se ha visitado
                dist[u] = dist[s] + 1;
                q.push(u);
            }
        }
    }
}

void dfs(int s) { //asignar previamente dist[nodo_inicial] = 0
    for (auto u : grafo[s]) {
        if (dist[u] == -1) {
            dist[u] = dist[s] + 1;
            dfs(u);
        }
    }
}
```

6.2 Dijkstra

```cpp
using pi = pair<int, int>;
using vpi = vector<pi>;

const int MAXN = 1e5 + 5;

// Si se tiene un grafo sin peso, usar BFS.
vpi graph[MAXN]; // Grafo guardado como lista de adyascencia.
int dist[MAXN];

template <class T>
using pqg = priority_queue<T, vector<T>, greater<T>>;

/*Llena un arreglo (dist), donde dist[i] indica la distancia mas corta
que se tiene que recorrer desde un nodo 'x' para llegar al nodo 'i',
en caso de que 'i' no sea alcanzable desde 'x', dist[i] = -1

O(V + E log V)                                                    */
void dijkstra(int x) {
    F0R(i, MAXN)
    dist[i] = INF;
    dist[x] = 0;

    pqg<pi> pq;
    pq.emplace(0, x);
    while (!pq.empty()) {
        auto [du, u] = pq.top();
        pq.pop();
        if (du > dist[u])
            continue;

        for (auto &[v, dv] : graph[u]) {
            if (du + dv < dist[v]) {
                dist[v] = du + dv;
                pq.emplace(dist[v], v);
            }
        }
    }

    // Si la pq puede tener muchisimos elementos, utilizamos un set, en donde
     habra a lo mucho V elementos
    set<pi> pq;
    for (int u = 0; u < V; ++u)
        pq.emplace(dist[u], u);

    while (!pq.empty()) {
        auto [du, u] = *pq.begin();
        pq.erase(pq.begin());
        for (auto &[v, dv] : graph[u]) {
            if (du + dv < dist[v]) {
                pq.erase(pq.find({dist[v], v}));
                dist[v] = du + dv;
                pq.emplace(dist[v], v);
            }
        }
    }
}
```

6.3 Bellman-Ford

```cpp
#define F0R(k, n) for(int k = 0; k < n; k++)
#define ENDL '\n'

int main() {
    int n, m, A, B, W;
    cin >> n >> m;
    tuple<int, int, int> edges[m];
    F0R(i, m) {
        cin >> A >> B >> W;
        edges[i] = make_tuple(A, B, W);
    }
    vi dist(n + 1, INF);

    int x;
    cin >> x;
    dist[x] = 0; // Nodo de inicio
    F0R(i, n) {
        for (auto e : edges) {
            auto [a, b, w] = e;
            dist[b] = min(dist[b], dist[a] + w);
        }
    }

    for (auto e : edges) {
        auto [u, v, weight] = e;
        if (dist[u] != INF && dist[u] + weight < dist[v]) {
            cout << "Graph contains negative weight cycle" << endl;
            return 0;
        }
    }

    cout << "Shortest distances from source " << x << ENDL;
    F0R(i, n) {
        cout << (dist[i] == INF ? -1 : dist[i]) << " ";
    }
```

```
    return 0;
}
```

6.4 Floyd-Warshall

```
// Matrix adjacency necessary.
int graph[MAXN][MAXN];
int p[MAXN][MAXN]; // Guardar camino

void floydWarshall() {
    FOR(i, N) { // Inicializar el camino
        FOR(j, N) {
            p[i][j] = i;
        }
    }

    FOR(k, N) {
        FOR(i, N) {
            FOR(j, N) {
                if (graph[i][k] + graph[k][j] < graph[i][j]) // Solo utilizar si
                      necesitas el camino
                    p[i][j] = p[k][j];

                graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);
            }
        }
    }
    // Now, graph[a][b] has the min distance from node a to node b, for all a
         and b.
}

void printPath(int i, int j) {
    if (i != j)
        printPath(i, p[i][j]);
    cout << j << " ";
}
```

6.5 Lowest Common Ancestor

```
const LOG_MAXN = 25;
vi tree[MAXN];
int jump[MAXN][LOG_MAXN]; //Donde jump[u][h] es el ancestro 2^h del nodo u
int depth[MAXN];

// DFS para calcular la profundidad y guardar el padre directo en jump[u][0]
void dfs(int u, int padre = -1, int d = 0) {
    depth[u] = d;
    jump[u][0] = padre;
    for (auto &hijo : tree[u])
        if (hijo != padre)
            dfs(hijo, u, d + 1);
}

void build(int n) {
    memset(jump, -1, sizeof jump);

    dfs(0);

    // Construccion del binary-lifting
    for (int i = 1; i < LOG_MAXN; i++)
        for (int u = 0; u < n; u++)
            if (jump[u][i - 1] != -1)
                jump[u][i] = jump[jump[u][i - 1]][i - 1];
}

int LCA(int p, int q) {
    if (depth[p] < depth[q])
        swap(p, q);

    int dist = depth[p] - depth[q]; // Distancia necesaria para estar en la
         misma profundidad
    FORR(i, LOG_MAXN)
    if ((dist >> i) & 1)
        p = jump[p][i];

    if (p == q) // Verificar si el ancestro es la misma profundidad
        return p;

    // Busqueda por saltos binarios
    FORR(i, LOG_MAXN)
    if (jump[p][i] != jump[q][i]) {
        p = jump[p][i];
        q = jump[q][i];
    }

    return jump[p][0];
}
```

6.6 Kruskal UnionFind

```
struct DSU { // Indice base 0
    vi e;
    void init(int N) { e = vi(N, -1); }
    int get(int x) { return e[x] < 0 ? x : e[x] = get(e[x]); }
    bool sameSet(int a, int b) { return get(a) == get(b); }
    int size(int x) { return -e[get(x)]; }
    bool unite(int x, int y) { // union by size
        x = get(x), y = get(y);
        if (x == y)
            return 0;
        if (e[x] > e[y])
            swap(x, y);
        e[x] += e[y];
        e[y] = x;
        return 1;
    }
};

using Edge = tuple<int, int, int>;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(nullptr);

    int V, E;
    cin >> V >> E;

    DSU UF;
    UF.init(V);
    Edge edges[E];

    FOR(i, E) {
        int u, v, w;
        cin >> u >> v >> w;
        edges[i] = {w, u, v};
    }
    sort(edges, edges + E);

    int totalWeight = 0;
    for (int i = 0; i < E && V > 1; i++) {
```

```
        auto [w, u, v] = edges[i]; // desempaquetamiento de arista
        if (!UF.sameSet(u, v)) {   // Si no estan en el mismo conjunto, la
             tomamos
            totalWeight += w;
            V -= UF.unite(u, v);
        }
    }
    cout << "MST weight: " << totalWeight << '\n';
    return 0;
}
```

6.7 Prim

```
/*Grafo de ejemplo:
  5 7
  0 1 4
  0 2 4
  0 3 6
  0 4 6
  1 2 2
  2 3 8
  3 4 9
  Salida esperada: 18
*/

#define eb emplace_back;

template <class T>
using pqg = priority_queue<T, vector<T>, greater<T>>;

const int MAXN = 1e5 + 5;

vii graph[MAXN];
bool taken[MAXN]; //Inicialmente en false todos
pqg<ii> pq; //Para ir seleccionando las aristas de menor peso

void process(int u) {
    taken[u] = 1;
    for (auto &[v, w] : graph[u])
        if (!taken[v])
            pq.emplace(w, v);
}

int main() {
    int V, E; cin >> V >> E;
    FOR(i, E){
        int u, v, w;
        cin >> u >> v >> w;
        //u--; v--;
        graph[u].eb(v, w);
        graph[v].eb(u, w);
    }

    process(0);                                    // take+process vertex 0
    int totalWeight = 0, takenEdges = 0;              // no edge has been taken
    while (!pq.empty() && takenEdges != V - 1) {                          // up
         to O(E)
        auto [w, u] = pq.top(); //Se desempaqueta la arista con menor peso
        pq.pop();

        if (taken[u]) continue; //Si ha sido tomada

        totalWeight += w;
        process(u);
        ++takenEdges;
    }
    cout << "MST weight: " << totalWeight << '\n';
    return 0;
}
```

6.8 Bridge Detection

```
const int MAXN = 1e6 + 5;

int n;                    // number of nodes
vector<int> adj[MAXN]; // adjacency list of graph
bool articulation[MAXN];
int tin[MAXN], low[MAXN], timer, dfsRoot, rootChildren;

void dfs(int u, int p = -1) {
    tin[u] = low[u] = timer++;
    for (int to : adj[u]) {
        if (to == p)
            continue;
        if (tin[to] != -1)
            low[u] = min(low[u], tin[to]);
        else {
            if (u == dfsRoot)
                ++rootChildren; // Caso especial si es raiz

            dfs(to, u);

            if (low[to] >= tin[u]) // Busca si es un punto de articulacion
                articulation[u] = 1;
            if (low[to] > tin[u]) // Busca si es un puente
                IS_BRIDGE(u, to);

            low[u] = min(low[u], low[to]);
        }
    }
}

void find_bridges_articulations() {
    timer = 0;
    fill(tin, tin + n, - 1);
    fill(low, low + n, - 1);

    for (int i = 0; i < n; ++i) {
        if (tin[i] == -1) {
            dfsRoot = i;
            rootChildren = 0;
            dfs(i);
            articulation[dfsRoot] = (rootChildren > 1);
        }
    }
}
```

6.9 Ordenamiento Topologico

```
const int MAXN = 1e5 + 5;
int n, m;                       // Numero de nodos y aristas
vi graph[MAXN];                 // Grafo
vi sorted_nodes;                // Arreglo de nodos ordenados topologicamente
bool visited[MAXN] = {false}; // Arreglo de visitados
stack<int> s;

// Funcion DFS para recorrer el grafo en profundidad
void dfs(int u) {
    visited[u] = true;
    for (auto v : graph[u]) {
        if (!visited[v])
```

```
            dfs(v);
        }
        s.push(u);
    }

    void topo_sort() {
        // Recorrido DFS para marcar los nodos visitados y llenar la pila
        FOR(i, n) {
            if (!visited[i])
                dfs(i);
        }

        // Llenado del arreglo
        while (!s.empty()) {
            sorted_nodes.push_back(s.top());
            s.pop();
        }
    }

    int main() {
        ios_base::sync_with_stdio(0);
        cin.tie(nullptr);

        cin >> n >> m;
        FOR(i, m) {
            int u, v;
            cin >> u >> v;
            graph[u].push_back(v);
        }
        topo_sort();
        if (sorted_nodes.size() < n) {
            cout << "El grafo tiene un ciclo" << ENDL;
        } else {
            cout << "Orden topologico: ";
            for (int u : sorted_nodes) {
                cout << u << " ";
            }
        }
        return 0;
    }
```

6.10 Ordenamiento Topologico Lexicografico

```
    const int MAXN = 1e5 + 5;
    int n, m;            // Numero de nodos y aristas
    vi graph[MAXN];      // Grafo
    int in_degree[MAXN]; // Grado de entrada de cada nodo
    vi sorted_nodes;     // Arreglo de nodos ordenados topologicamente

    void topo_sort() {
        priority_queue<int, vector<int>, greater<int>> q;
        FOR(i, n) {
            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 : graph[u]) {
                in_degree[v]--;
                if (in_degree[v] == 0)
                    q.push(v);
            }
        }
    }

    int main() {
        ios_base::sync_with_stdio(0);
        cin.tie(nullptr);

        cin >> n >> m;
        FOR(i, m) {
            int u, v;
            cin >> u >> v;
            graph[u].push_back(v);
            in_degree[v]++;
        }
        topo_sort();

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

        return 0;
    }
```

6.11 Algoritmo de Tarjan (SCC)

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

    La idea basica del algoritmo de Tarjan es que los SCC forman subarboles en el
        arbol de
    expansion de la DFS. Ademas de calcular tin(u) y low(u) para cada vertice,
        anadimos el
    vertice u al final de una pila y mantenemos la informacion de que vertices estan
         siendo
    explorados, mediante vi visited. Solo los vertices que estan marcados como
        visited (parte
    del SCC actual) pueden actualizar low(u). Ahora, si tenemos el vertice u en este
         arbol de
    expansion DFS con low(u) = tin(u), podemos concluir que u es la raiz de un SCC y
         los miembros
    de estos SCC se pueden identificar obteniendo el contenido actual de la pila,
        hasta que volvamos
    a llegar al vertice u
    */

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

    vector<int> tin, low, visited;
    int timer, numSCC;
    stack<int> pila;

    void tarjanSCC(int u) {
        tin[u] = low[u] = timer++;
        pila.push(u);
        visited[u] = 1;
        for (int to : adj[u]) {
            if (tin[to] == -1)
                tarjanSCC(to);
```

```
        if (visited[to])
            low[u] = min(low[u], low[to]);
    }
    if (low[u] == tin[u]) {
        ++numSCC;
        while (1) {
            int v = pila.top();
            pila.pop();
            visited[v] = 0;
            if (u == v)
                break;
        }
    }
}

int main() {
    timer = 0;
    tin.assign(n, -1);
    low.assign(n, 0);
    visited.assign(n, 0);
    while (!pila.empty())
        pila.pop();
    timer = numSCC = 0;
    FOR(i, n) {
        if (tin[i] == -1)
            tarjanSCC(i);
    }
}
```

6.12 Algoritmo de Kosaraju (SCC)

```
/*
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
*/

vi graph[MAXN];   // Grafo
vi graph_T[MAXN]; // Grafo transpuesto
vi dfs_num[MAXN];
vi S;
int N, numSCC;

void Kosaraju(int u, int pass) {
    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.pb(u);
}

int main() {
    S.clear();
    dfs_num.assign(N, -1);
    FOR(u, N) {
        if (dfs_num[u] == -1)
            Kosaraju(u, 1);
    }
    dfs_num.assign(N, -1);
    numSCC = 0;
    ROF(i, N) { // Segunda pasada
        if (dfs_num[S[i]] == -1) {
            ++numSCC;
            Kosaraju(S[i], 2);
        }
    }
    cout << numSCC << ENDL;
}
```

6.13 Hierholzer (Camino euleriano)

```
/*
Busqueda de un camino euleriano - Hierholzer O(E)
Un camino euleriano se define como el recorrido de un grafo que visita
cada arista del grafo exactamente una vez
Un grafo no dirigido es euleriano si, y solo si: es conexo y todos los
vertices tienen un grado par
Un grafo dirigido es euleriano si, y solo si: es conexo y todos los vertices
tienen el mismo numero de aristas entrantes y salientes. Si hay, exactamente,
un vertice u que tenga una arista saliente adicional y, exactamente, un
vertice v que tenga una arista entrante adicional, el grafo contara con un
camino euleriano de u a v
*/

int N;
vector<vi> graph; // Grafo dirigido

vi hierholzer(int s) {
    vi ans, idx(N, 0), st;
    st.pb(s);
    while (!st.empty()) {
        int u = st.back();
        if (idx[u] < (int)graph[u].size()) {
            st.pb(graph[u][idx[u]]);
            ++idx[u];
        } else {
            ans.pb(u);
            st.pop_back();
        }
    }
    reverse(all(ans));
    return ans;
}
```

6.14 EulerTour RSQ

```
/*
Problema: Tenemos un arbol de n nodos, en donde cada vertice tiene un valor
     inicial y podemos realizar 2 tipos de queries:
1 - update(x, val): al valor del nodo x agregale val, al de sus hijos restales
     val, al de los nietos sumales val, etc.
2 - get(x): obtener el valor actual del nodo x.

Approach: Euler tour, asignandole a cada nodo un indice (st[i]) y otro de salida
     (en[i]) para aplanar el arbol y realizar
operaciones con estructuras de rangos, en este caso usamos el Segment Tree.

Nos basamos en utilizar 2 Segment Trees:
ST1 - Guardar solamente valores de cada nodo en su indice de entrada, caso
     contrario ST1[i] = 0.
```

```
ST2 - Guardar solamente valores de cada nodo en su indice de salida, caso
     contrario ST2[i] = 0.

Para calcular la suma desde la raiz (0 en este caso) hasta x lo que podemos
     hacer es inclusion-exclusion, realizando un
ST1.RSQ(0, en[x]) - ST2.RSQ(0, en[x]) si prestamos atencion de manera detallada
     nos podemos dar cuenta que se van a "borrar"
las aristas que no pertenecen al camino deseado, puesto que hasta ese punto no
     se ha "subido" por ese camino aun y por lo tanto
los unicos valores que tendra ST2.RSQ(0, en[x]) son los valores "no deseados",
     es decir, que pueden ser hijos de algun nodo del
camino pero no deben de ser tomados en cuenta para la respuesta.

Ahora imaginemonos que el camino desde la raiz hasta x esta dado por los nodos
     u0(la raiz), u1, u2, u3, ..., x, una observacion
que se puede hacer es que si el numero de aristas entre u0 y x (o su profundidad
     ) es par, se calcula:
ans[x] = value[x] + sum_camino = value[x] + upd[u0] - upd[u1] + upd[u2] - upd[u3
     ] + ... + upd[x]
Y si es impar:
ans[x] = value[x] + sum_camino = value[x] - upd[u0] + upd[u1] - upd[u2] + upd[u3
     ] + ... + upd[x]

Entonces podemos simplemente actualizar los nodos de nuestros ST de la siguiente
      forma:
- Si depth[x] es par, hacemos upd[x] += val;
- Si no lo es, hacemos upd[x] -= val;
A final de cuentas lo que logramos es que sum_camino quede de la forma:
upd[u0] - upd[u1] + upd[u2] - upd[u3] + ... + upd[x]
Y como solo difiere por signos depende de la profundidad, multiplicamos por -1 a
      conveniencia.

Para RMQ en un sub-arbol el problema es mucho mas facil (lol)
*/

vi tree[MAXN];
int val[MAXN], depth[MAXN], st[MAXN], en[MAXN], toursz = 0;
void dfs(int u, int father) {
    st[u] = toursz++;

    for (auto& v : tree[u]) {
        if (v != father) {
            depth[v] = depth[u] + 1;
            dfs(v, u);
        }
    }

    en[u] = toursz++;
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(nullptr);

    int n, q; cin >> n >> q;
    F0R (i, n)
        cin >> val[i];

    F0R (i, n - 1) {
        int u, v; cin >> u >> v;
        tree[u - 1].pb(v - 1);
        tree[v - 1].pb(u - 1);
    }
    depth[0] = 0;
    dfs(0, -1);

    SegmentTree ST1(toursz), ST2(toursz);
    while (q--) {
        int op, x; cin >> op >> x;
        x--;
        int parity = ((depth[x] & 1) ? -1 : 1);
        if (op == 1) {
            int val; cin >> val;
            ST1.update(st[x], val * parity);
            ST2.update(en[x], -val * parity);
        }
        else
            cout << (ST1.query(0, st[x]) + ST2.query(0, st[x])) * parity + val[x
                 ] << ENDL;
    }

    return 0;
}
```

6.15 Hopcroft-Karp

```
// Algoritmo para resolver el problema de maximum bipartite matching
// Nota. Modelar bien el grafo :)
// 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;
}
```

6.16 Max Flow

```
using ll = long long;
using edge = tuple<int, ll, ll>;
using vi = vector<int>;
using pi = pair<int, int>;

const ll INF = 1e18;                                // large enough

class max_flow {
private:
  int V;
  vector<edge> EL;
  vector<vi> AL;
  vi d, last;
  vector<pi> p;

  bool BFS(int s, int t) {                          // find augmenting path
    d.assign(V, -1); d[s] = 0;
    queue<int> q({s});
    p.assign(V, {-1, -1});                          // record BFS sp tree
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;                            // stop as sink t reached
      for (auto &idx : AL[u]) {                     // explore neighbors of u
        auto &[v, cap, flow] = EL[idx];             // stored in EL[idx]
        if ((cap-flow > 0) && (d[v] == -1))         // positive residual edge
          d[v] = d[u]+1, q.push(v), p[v] = {u, idx}; // 3 lines in one!
      }
    }
    return d[t] != -1;                              // has an augmenting path
  }

  ll send_one_flow(int s, int t, ll f = INF) {      // send one flow from s->t
    if (s == t) return f;                           // bottleneck edge f found
    auto &[u, idx] = p[t];
    auto &cap = get<1>(EL[idx]), &flow = get<2>(EL[idx]);
    ll pushed = send_one_flow(s, u, min(f, cap-flow));
    flow += pushed;
    auto &rflow = get<2>(EL[idx^1]);                // back edge
    rflow -= pushed;                                // back flow
    return pushed;
  }

  ll DFS(int u, int t, ll f = INF) {                // traverse from s->t
    if ((u == t) || (f == 0)) return f;
    for (int &i = last[u]; i < (int)AL[u].size(); ++i) { // from last edge
      auto &[v, cap, flow] = EL[AL[u][i]];
      if (d[v] != d[u]+1) continue;                 // not part of layer graph
      if (ll pushed = DFS(v, t, min(f, cap-flow))) {
        flow += pushed;
        auto &rflow = get<2>(EL[AL[u][i]^1]);       // back edge
        rflow -= pushed;
        return pushed;
      }
    }
    return 0;
  }

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

  // if you are adding a bidirectional edge u<->v with weight w into your
  // flow graph, set directed = false (default value is directed = true)
  void add_edge(int u, int v, ll w, bool directed = true) {
    if (u == v) return;                             // safeguard: no self loop
    EL.emplace_back(v, w, 0);                       // u->v, cap w, flow 0
    AL[u].push_back(EL.size()-1);                   // remember this index
    EL.emplace_back(u, directed ? 0 : w, 0);        // back edge
    AL[v].push_back(EL.size()-1);                   // remember this index
  }

  ll edmonds_karp(int s, int t) {
    ll mf = 0;                                      // mf stands for max_flow
    while (BFS(s, t)) {                             // an O(V*E^2) algorithm
      ll f = send_one_flow(s, t);                   // find and send 1 flow f
      if (f == 0) break;                            // if f == 0, stop
      mf += f;                                      // if f > 0, add to mf
    }
    return mf;
  }

  ll dinic(int s, int t) {
    ll mf = 0;                                      // mf stands for max_flow
    while (BFS(s, t)) {                             // an O(V^2*E) algorithm
      last.assign(V, 0);                            // important speedup
      while (ll f = DFS(s, t))                      // exhaust blocking flow
        mf += f;
    }
    return mf;
  }
};
```

7 Extras

7.1 Operaciones con matrices

```
// Sea A una matriz de orden n*n y k un numero, podemos calcular A^k en O(log k
    * n^3)
typedef vector<vi> vvi;
// O(n^3)
vvi matrixMultiplication(vvi &A, vvi &B) {
    int n = A.size(), m = A[0].size(), k = B[0].size();
    vvi C(n, vi(k, 0));

    FOR(i, n)
    FOR(j, k)
    FOR(l, m)
    C[i][j] += (A[i][l] * B[l][j]) % MOD;
    return C;
}
// O(log k * n^3)
vvi matrixExponentiation(vvi &A, ll k) {
    int n = A.size();

    vvi ret(n, vi(n)), B = A; // Matriz identidad
    FOR(i, n)
    ret[i][i] = 1;

    while (k) {
        if (k & 1)
            ret = matrixMultiplication(ret, B);
        k >>= 1;
        B = matrixMultiplication(B, B);
    }
    return ret;
}
// Si el problema es el tiempo, implementar struct Matrix

/*
Estrategia para calcular el n-esimo fibonacci en O(log n):
  Utilizar la matriz de fibonacci:
A = {1, 1} ^ n
    {1, 0}
Donde
A = {F[n+1], F[n]  }
    {F[n]  , F[n-1]}
Utilizar exponenciacion binaria de la matriz, con 4 variables para que sea veloz
    .
*/
```

7.2 Fechas

```
// Routines for performing computations on dates. In these routines,
// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.
string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt(int m, int d, int y) {
    return 1461 * (y + 4800 + (m - 14) / 12) / 4 +
           367 * (m - 2 - (m - 14) / 12 * 12) / 12 - 3 * ((y + 4900 + (m - 14) /
                 12) / 100) / 4 +
           d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year
void intToDate(int jd, int &m, int &d, int &y) {
    int x, n, i, j;

    x = jd + 68569;
    n = 4 * x / 146097;
    x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001;
    x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447;
    d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x;
    y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay(int jd) {
    return dayOfWeek[jd % 7];
}

int main() {
    int jd = dateToInt(3, 24, 2004);
    int m, d, y;
    intToDate(jd, m, d, y);
    string day = intToDay(jd);

    // expected output:
    // 2453089
    // 3/24/2004
    // Wed
    cout << jd << endl
         << m << "/" << d << "/" << y << endl
         << day << endl;
}
```

7.3 Trucos

```
// Imprimir una cantidad especifica de digitos
// despues del punto decimal en este caso 5
cout.setf(ios::fixed); cout << setprecision(5);
cout << 100.0/7.0 << '\n';
cout.unsetf(ios::fixed);

// Imprimir el numero con su decimal y el cero a su derecha
// Salida -> 100.50, si fuese 100.0, la salida seria -> 100.00
cout.setf(ios::showpoint);
cout << 100.5 << '\n';
cout.unsetf(ios::showpoint);

// Imprime un '+' antes de un valor positivo
cout.setf(ios::showpos);
cout << 100 << ' ' << -100 << '\n';
cout.unsetf(ios::showpos);

// Imprime valores decimales en hexadecimales
cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;

// Redondea el valor dado al entero mas cercano
round(5.5);

// Se realiza la operacion y redondea, siempre que se pueda, hacia abajo el
    numero
cout << a / b;

// Se realiza la operacion y redondea, siempre que se pueda, hacia arriba el
    numero
cout << (a + b - 1) / b;
```

```
// Llena la estructura con el valor (unicamente puede ser -1 o 0)
memset(estructura, valor, sizeof estrutura);

// Llena el arreglo/vector x, con value en cada posicion.
fill(begin(x), end(x), value);

// True si encuentra el valor, false si no
binary_search(begin(x), end(x), value);

// Retorna un iterador que apunta a un elemento mayor o igual a value
lower_bound(begin(x), end(x), value);

// Retorna un iterador que apunta a un elemento MAYOR a value
upper_bound(begin(x), end(x), value);

// Retorna un pair de iteradores, donde first es el lower_bound y second el
    upper_bound
equal_range(begin(x), end(x), value);

// True si esta ordenado x, false si no.
is_sorted(begin(x), end(x));

// Ordena de forma que si hay 2 cincos, el primer cinco estara acomodado antes
    del segundo, tras ser ordenado
stable_sort(begin(x), end(x));
```