ICPC Team AC2++ Notebook

Contents

1 Templates 1
- 1.1 Plantilla C++ . . . 1
- 1.2 Plantilla Phyton . . . 1

2 Data Structures 1
- 2.1 Trie . . . 1
- 2.2 Fenwick Tree . . . 1
- 2.3 Binary Indexed Tree . . . 2
- 2.4 Order Statistics Tree . . . 2
- 2.5 Segment Tree . . . 2
- 2.6 Lazy Segment Tree . . . 3
- 2.7 Lazy Range Min/Max Query . . . 3

3 Math 3
- 3.1 Numeros Primos . . . 3
- 3.2 Operaciones con Bits . . . 4
- 3.3 Formulas Rapidas . . . 4
- 3.4 Operaciones de Matriz . . . 4
- 3.5 Numeros Catalan . . . 4

4 Dynamic Programming 4
- 4.1 Problema de la mochila . . . 4
- 4.2 Longest Increasing Subsequence (LIS) . . . 4
- 4.3 Dp con Digitos . . . 5
- 4.4 Tecnica con pila . . . 5

5 Graphs 5
- 5.1 Recorrido BFS y DFS . . . 5
- 5.2 Dijkstra . . . 5
- 5.3 Bellman-Ford . . . 5
- 5.4 Floyd-Warshall . . . 5
- 5.5 Lowest Common Ancestor . . . 6
- 5.6 Kruskal UnionFind . . . 6
- 5.7 Prim . . . 6
- 5.8 Bridge Detection . . . 6
- 5.9 Ordenamiento Topologico . . . 6
- 5.10 Ordenamiento Topologico Lexicografico . . . 7

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

```cpp
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

```cpp
#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

```
/*Esta implementado para obtener la suma en un rango, pero es posible usar cualquier
operacion conmutativa como la multiplicacion, XOR, AND, OR, MIN, MAX, etc.*/

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); }
};
```

2.6 Lazy Segment Tree

```
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); }
};
```

2.7 Lazy Range Min/Max Query

```
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); }
};

// Implementacion
vi A = {18, 17, 13, 19, 15, 11, 20, 99};
SegmentTree 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
```

3 Math

3.1 Numeros Primos

```
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;
}

// Sin calcular primos en 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
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

```
// 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++) {
  ....
}
*/
```

3.3 Formulas Rapidas

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

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

// Coeficientes binomiales (Combinatoria)
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)

// 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 Triple({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 Operaciones de Matriz

```
// Let A be an n*n order matrix and k the exponent, we can calculate A
      ^k in O(log k * n^3)
typedef vector<vi> vvi;
// A * B = C, 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;
}
// A^k, O(log k * n^3)
vvi matrixExponentiation(vvi &A, ll k) {
    int n = A.size();
    // ret -> identity matrix
    vvi ret(n, vi(n)), B = A;
    FOR(i, n)
    ret[i][i] = 1;

    while (k) {
        if (k & 1)
            ret = matrixMultiplication(ret, B);
        k >>= 1;
        B = matrixMultiplication(B, B);
    }
    return ret;
}
// Another faster approach could be use structs with fixed matrices
      overloading the * operator
```

3.5 Numeros Catalan

```
# 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]))
```

4 Dynamic Programming

4.1 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.

Ejemplo:

Entrada
5
4
4 4
1 3
3 2
9 5
1 3

Salida
6

*/

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);
*/
```

4.2 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;
}
```

4.3 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;
}
```

4.4 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 Graphs

5.1 Recorrido BFS y DFS

```
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);
        }
    }
}
```

5.2 Dijkstra

```
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) {
    FOR(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);
            }
        }
    }
}
```

5.3 Bellman-Ford

```
#define FOR(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];
    FOR(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
    FOR(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;
    FOR(i, n) {
        cout << (dist[i] == INF ? -1 : dist[i]) << " ";
    }

    return 0;
}
```

5.4 Floyd-Warshall

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

void floydWarshall() {
    FOR(k, N)
        FOR(i, N)
```

```
            FOR(j, N)
                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.
}
```

5.5 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];
}
```

5.6 Kruskal UnionFind

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

class UnionFind {
    private: int numSets, parent[MAXN], rank[MAXN], setSize[MAXN];
    public:
        UnionFind(int &N) {
            for(int i = 0; i < N; i++)
                parent[i] = i;
            numSets = N;
        }
        int get(int i) { return (parent[i] == i) ? i : (parent[i] = get(parent[i])); }
        bool isSame(int i, int j) { return get(i) == get(j); }
        int sizeOfSet(int i) { return setSize[get(i)]; }
        void unite(int i, int j) {
            if(!isSame(i, j)) {
                int x = get(i), y = get(j);
                if (rank[x] > rank[y]) swap(x, y);
                parent[x] = y;
                if (rank[x] == rank[y]) ++rank[y];
                setSize[y] += setSize[x];
                --numSets;
            }
        }
};

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

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

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

    UnionFind UF(V);
    Edge edges[V];

    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 && UF.numSets > 1; i++) {
        auto [w, u, v] = edges[i]; // desempaquetamiento de arista
        if (!UF.isSame(u, v)) {       // Si no estan en el mismo conjunto, la tomamos
            totalWeight += w;
            UF.unite(u, v);
        }
    }
    cout << "MST weight: " << totalWeight << '\n';
    return 0;
}
```

5.7 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;
}
```

5.8 Bridge Detection

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

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

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

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

5.9 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;
}
```

5.10 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;
```

```
}
```