

AC2++ ICPC Team Notebook

Contents

1	Templates	2
1.1	Plantilla C++	2
1.2	Plantilla Python	2
1.3	Plantilla C++ Max	2
2	Data Structures	3
2.1	Binary Indexed Tree	3
2.2	DSU RollBack	3
2.3	Fenwick Tree	3
2.4	Order Statistics Tree	3
2.5	Sparse Table	4
2.6	Segment Tree	4
2.7	Lazy Segment Tree	4
2.8	LazyRMQ	5
2.9	Sparse Segment Tree	5
2.10	Persistent Lazy Propagation	5
2.11	Iterative Segment Tree	6
3	Math	7
3.1	Operaciones con Bits	7
3.2	Combinaciones	7
3.3	Algoritmo Ext. de Euclides	7
3.4	FFT	7
3.5	Linear Diophantine	8
3.6	Matrix	8
3.7	Operaciones con MOD	8
3.8	Numeros Primos	8
3.9	Simpson	9
4	Strings	10
4.1	Aho-Corasick	10
4.2	Dynamic Aho-Corasick	11
4.3	Hashing	11
4.4	KMP	11
4.5	Manacher	12
4.6	Suffix Array	12
4.7	Suffix Automaton	13
4.8	Suffix Tree	13
4.9	Trie	14
4.10	Z-Algorithm	14
5	Dynamic Programming	15
5.1	2D Sum	15
5.2	Tecnica con Deque	15
5.3	DP con digitos	15
5.4	Knapsack	15
5.5	Longest Increasing Subsequence	15
5.6	Monotonic Stack	15
5.7	Travelling Salesman Problem	16
6	Graphs	17
6.1	2SAT	17
6.2	Bridge Detection	17
6.3	Kosaraju (SCC)	17
6.4	Tarjan (SCC)	17
6.5	General Matching	18
6.6	Hopcroft Karp	18
6.7	Hungaro	19
6.8	Kuhn	19
6.9	Kruskal (MST)	19
6.10	Prim (MST)	19
6.11	Dinic	19
6.12	Min Cost Max Flow	20
6.13	Push Relabel	20
6.14	Bellman-Ford	20
6.15	Dijkstra	21
6.16	Floyd-Warshall	21
6.17	Binary Lifting LCA	21
6.18	Euler Tour	21
6.19	Find Centroid	21
6.20	Hierholzer	21
6.21	Orden Topologico	22
7	Geometry	23
7.1	Punto	23
7.2	Linea	23
7.3	Poligono	23
7.4	Fracciones	24
7.5	Convex Hull	24
7.6	Punto 3D	24
8	Extras	25
8.1	Fechas	25
8.2	HashPair	25
8.3	Trucos	25

1 Templates

1.1 Plantilla C++

```
#include <bits/stdc++.h>
using namespace std;
// AC2++
using ll = long long;
using pi = pair<int, int>;
using vi = vector<int>;

#define fi first
#define se second
#define pb push_back
#define SZ(x) int((x).size())
#define ALL(x) begin(x), end(x)
#define FOR(i, a, b) for (int i = (a); i < (b); ++i)
#define ROF(i, a, b) for (int i = (a)-1; i >= (b); --i)
#define ENDL '\n'

constexpr int MOD = 1e9 + 7;
constexpr int MAXN = 1e5 + 5;
constexpr int INF = 1e9;
constexpr ll LLINF = 1e18;

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

    return 0;
}
```

1.2 Plantilla 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
```

1.3 Plantilla C++ Max

```
#include <bits/stdc++.h>
using namespace std;
// AC2++
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 fi first
#define se 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 FOR(i, a) FOR(i, 0, a)
#define ROF(i, a, b) for (int i = (a)-1; i >= (b); --i)
#define ROF(i, a) ROF(i, a, 0)

#define ENDL '\n'
#define LOne(S) ((S) & -(S))
```

```
#define MSET(arr, val) memset(arr, val, sizeof arr)

const int MOD = 1e9 + 7;
const int MAXN = 1e5 + 5;
const int INF = 1e9;
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;
}
```

2 Data Structures

2.1 Binary Indexed Tree

```
/**
 * Descripcion: arbol binario indexado, util para consultas
 * en
 * donde es posible hacer inclusion-exclusion, suma,
 * multiplicacion,
 * etc. Utilizar indices 1-indexados, query -> [1, index]
 * La diferencia entre BIT.cpp y este es la cantidad de
 * operaciones
 * soportadas, con este, son posibles las actualizaciones en
 * rango.
 * Tiempo: O(log n)
 */

int n, bit[MAX];

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

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

2.2 DSU RollBack

```
/**
 * Descripcion: Estructura de conjuntos disjuntos con la
 * capacidad de regresar a estados anteriores.
 * Si no es necesario, ignorar st, time() y rollback().
 * Uso: int t = uf.time(); ...; uf.rollback(t)
 * Tiempo: O(log n)
 */

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

2.3 Fenwick Tree

```
/**
 * Descripcion: arbol binario indexado, util para consultas
 * en
 * donde es posible hacer inclusion-exclusion, suma,
 * multiplicacion,
 * etc. Utilizar indices 1-indexados, query -> [1, index]
 * La diferencia entre BIT.cpp y este es la cantidad de
 * operaciones
 * soportadas, con este, son posibles las actualizaciones en
 * rango.
 * Tiempo: O(log n)
 */

#define LSONe(S) ((S) & -(S))

class FenwickTree {
private:
    vll ft;

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

    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); }

    FenwickTree(int m, const vi &s) {
        vll f(m + 1, 0);
        FOR(i, (int)s.size()) {
            ++f[s[i]];
        }
        build(f);
    }

    ll query(int 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 { // 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 { // 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 rupq.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
```

2.4 Order Statistics Tree

```
/**
 * Descripcion: es una variante del BST, que ademas soporta
 * 2
 * operaciones extra ademas de insercion, busqueda y
 * eliminacion:
 * Select(i) - find_by_order: encontrar el i-esimo elemento
 * (0-indexado)
 * del conjunto ordenado de los elementos, retorna un
 * iterador.
 * Rank(x) - order_of_key: encontrar el rango de x en el
 * conjunto,
 * es decir, retorna su indice en la lista ordenada de los
 * elementos.
 * Uso:
 * oset<int> OST
```

```

* Funciona como un set, por lo que nativamente no soporta
  elementos
* repetidos. Si se necesitan repetidos, pero no eliminar
  valores,
* cambiar la funcion comparadora por less_equal<T>. Si se
  necesitan
* repetidos y tambien la eliminacion, agregar una dimension
  a T en
* en donde el ultimo parametro sea el diferenciador (por
  ejemplo,
* si estamos con enteros, utilizar un pair donde el second
  sea unico).
* Modificar el primer y tercer parametro (tipo y funcion
  comparadora),
* si se necesita un mapa, en lugar de null_type, escribir
  el tipo a mapear.
* Tiempo: O(log n)
*/
#include <bits/extc++.h>
using namespace __gnu_pbds;

template <class T>
using oset = tree<T, null_type, less<T>, rb_tree_tag,
  tree_order_statistics_node_update>;

```

2.5 Sparse Table

```

/**
 * Descripcion: nos permite realizar eficientemente
   consultas
 * en rango del minimo o maximo elemento, cuando no se
   actualiza
 * nunca el arreglo, ST[k][i] = min/max(A[i]...A[i + 2^k -
   1]);
 * Tiempo: O(n log n) en construccion y O(1) por query
 */

template <typename T>
struct SparseTable {
    vector<vector<T>> ST;

    void build(vector<T> &A) {
        const int MAX_LOG = __builtin_clz(A.size()) + 2;
        ST.assign(MAX_LOG, vector<T>(SZ(A), 0));

        for (int i = 0; i < SZ(A); i++)
            ST[0][i] = A[i];

        for (int k = 1; k < MAX_LOG; k++)
            for (int i = 0; i + (1 << k) <= SZ(A); i++)
                ST[k][i] = min(ST[k - 1][i], ST[k - 1][i + (1 << (k
                    - 1))]);

        T query(int l, int r) { // [l, r]
            int p = 31 - __builtin_clz(r - l + 1);
            return min(ST[p][l], ST[p][r - (1 << p) + 1]);
        }
    };
};

```

2.6 Segment Tree

```

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

```

```

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

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

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

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

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

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

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

        int mid = (start + end) / 2;

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

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

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

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

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

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

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

```

2.7 Lazy Segment Tree

```

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

```

```

* Tiempo: O(log n)
*/

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

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

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

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

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

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

        int mid = (start + end) / 2;

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

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

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

        int mid = (start + end) / 2;

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

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

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

```

```

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

```

2.8 LazyRMQ

```

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

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

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

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

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

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

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

            void update(int p, int L, int R, int i, int j, int val) {
                propagate(p, L, R);
                if (i > j)
                    return;
                if ((L >= i) && (R <= j)) {
                    lazy[p] = val;
                    propagate(p, L, R);
                }
            }
        }
    }

```

```

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

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

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

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

```

2.9 Sparse Segment Tree

```

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

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

    T query(int lo, int hi, int L = 0, int R = SZ - 1) { // [
        L, r]
        if (hi < L || R < lo) return 0;
        if (lo <= L && R <= hi) return val;
        int M = (L + R) / 2;
        T res = 0;
        if (c[0]) res += c[0]->query(lo, hi, L, M);
        if (c[1]) res += c[1]->query(lo, hi, M + 1, R);
        return res;
    }
}

```

2.10 Persistent Lazy Propagation

```

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

ll arr[MAXN];

ll l[45 * MAXN], r[45 * MAXN], st[45 * MAXN], nodes = 0;

bool hasFlag[45 * MAXN];
ll flag[45 * MAXN];

ll root[MAXN];

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

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

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

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

    return p;
}

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

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

ll update(ll a, ll b, ll x, ll p, ll left, ll right) {

```

```

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

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

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

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

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

```

```

    if (start % 2 == 1) {
        ans = min(ans, ST[start++]);
    } // Operacion
    if (end % 2 == 1) {
        ans = min(ans, ST[--end]);
    } // Operacion
    return ans;
}
};

```

2.11 Iterative Segment Tree

```

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

    vector<T> ST;
    int len;

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

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

    // [start, end]
    T query(int start, int end) {
        end++;
        T ans = DEFAULT;
        for (start += len, end += len; start < end; start /= 2,
              end /= 2) {

```

3 Math

3.1 Operaciones con Bits

```
/**
 * Descripcion: Algunas operaciones utiles con
 * desplazamiento de bits, si no trabajamos
 * con numeros enteros, usar 1LL o 1ULL, siendo la primer
 * parte
 * operaciones nativas y la segunda del compilador GNU (GCC)
 * , si no se
 * trabaja con enteros, agregar ll al final del nombre del
 * metodo
 * * Tiempo por operacion: O(1)
 */

#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)) // Siendo N 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))

#define countBitsOn(n) __builtin_popcount(x);
#define firstBitOn(n) __builtin_ffs(x);
#define countLeadingZeroes(n) __builtin_clz(n)
#define log2Floor(n) 31 - __builtin_clz(n)
#define countTrailingZeroes(n) __builtin_ctz(n)

/**
 * Descripcion: Si n <= 20 y manejamos subconjuntos, podemos
 * revisar
 * cada uno de ellos representandolos como una mascara de
 * bits, en
 * donde el i-esimo elemento es tomado si el i-esimo bit
 * esta encendido
 * * Tiempo: O(2^n)
 */
int LIMIT = 1 << (n + 1);
for (int i = 0; i < LIMIT; i++) {
}
```

3.2 Combinaciones

```
/**
 * Descripcion: Utilizando el metodo de ModOperations.cpp,
 * calculamos de manera
 * eficiente los inversos modulares de x (arreglo inv) y de
 * x! (arreglo invfact),
 * para toda x < MAXN, se utiliza el hecho de que comb(n, k)
 * = (n!) / (k! * (n - k)!)
 * * Tiempo: O(MAXN) en el precalculo de inversos modulares y
 * O(1) por query.
 */
ll invfact[MAXN];
void precalc_invfact() {
    precalc_inv();
    for (int i = 2; i < MAXN; i++)
        invfact[i] = invfact[i - 1] * inv[i] % MOD;
}

ll comb(int n, int k) {
```

```
    if (n < k)
        return 0;
    return fact[n] * invfact[k] % MOD * invfact[n - k] % MOD;
}

/**
 * Descripcion: Se basa en el teorema de lucas, se puede
 * utilizar cuando tenemos
 * una MAXN larga y un modulo m relativamente chico.
 * * Tiempo: O(m log_m(n))
 */
ll comb(int n, int k) {
    if (n < k || k < 0)
        return 0;
    if (n == k)
        return 1;
    return comb(n % MOD, k % MOD) * comb(n / MOD, k / MOD) %
MOD;
}

/**
 * Descripcion: Se basa en el triangulo de pascal, vale la
 * pena su uso cuando
 * no trabajamos con modulos (pues no tenemos una mejor
 * opcion), usa DP.
 * * Tiempo: O(n^2)
 */
ll dp[MAXN][MAXN];
ll comb(int n, int k) {
    if (k > n || k < 0)
        return 0;
    if (n == k || k == 0)
        return 1;
    if (dp[n][k] != -1)
        return dp[n][k];
    return dp[n][k] = comb(n - 1, k) + comb(n - 1, k - 1);
}
```

3.3 Algoritmo Ext. de Euclides

```
/**
 * Descripcion: Algoritmo extendido de Euclides, retorna gcd
 * (a, b) y encuentra
 * dos enteros (x, y) tal que ax + by = gcd(a, b), si solo
 * necesitas
 * el gcd, utiliza __gcd (c++14 o anteriores) o gcd (c++17
 * en adelante)
 * * Si a y b son coprimos, entonces x es el inverso de a mod
 * b
 * * Tiempo: O(log n)
 */
ll euclid(ll a, ll b, ll &x, ll &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    ll d = euclid(b, a % b, y, x);
    return y -= a / b * x, d;
}
```

3.4 FFT

```
/**
 * Descripcion: Este algoritmo permite multiplicar dos
 * polinomios de longitud n
 * * Tiempo: O(n log n)
 */
typedef double ld;
const ld PI = acos(-1.0L);
const ld one = 1;
```

```
typedef complex<ld> C;
typedef vector<ld> vd;

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

typedef vector<ll> vl;

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

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

3.5 Linear Diophantine

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

void solution(int a, int b, int n) {
    int x0, y0, g = euclid(a, b, x0, y0);
    if (n % g != 0) {
        cout << "No Solution Exists" << 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;
    }
}
```

3.6 Matrix

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

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

    VVT M;
    int n, m;

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

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

    Matrix operator^(ll p) const {
        assert(p >= 0);
```

```
Matrix ret(VVT(n, vector<T>(n))), B(*this);
FOR(i, 0, n)
    ret.M[i][i] = 1;

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

3.7 Operaciones con MOD

```
/**
 * Descripcion : Calcula a * b mod m para
 * cualquier 0 <= a, b <= c <= 7.2 * 10^18
 * Tiempo: O(1)
 */
using ull = unsigned long long;
ull modmul(ull a, ull b, ull m) {
    ll ret = a * b - m * ull(1.L / m * a * b);
    return ret + m * (ret < 0) - m * (ret >= (ll)m);
}

constexpr ll MOD = 1e9 + 7;

/**
 * Descripcion: Calcula a^b mod m, en O(log n)
 * Si hay riesgo de desbordamiento, multiplicar con modmul
 * Tiempo: O(log b)
 */
ll modpow(ll a, ll b) {
    ll res = 1;
    a %= MOD;
    while (b) {
        if (b & 1)
            res = (res * a) % MOD;
        a = (a * a) % MOD;
        b >>= 1;
    }
    return res;
}

/**
 * Descripcion: Precalculo de modulos inversos para toda
 * x <= LIM. Se asume que LIM <= MOD y que MOD es primo
 * Tiempo: O(LIM)
 */
constexpr LIM = 1e5 + 5;
ll inv[LIM + 1];
void precalc_inv() {
    inv[1] = 1;
    FOR(i, 2, LIM)
        inv[i] = MOD - (MOD / i) * inv[MOD % i] % MOD;
}

/**
 * Descripcion: Precalculo de un solo inverso, usa el primer
 * metodo si MOD es primo, y el segundo en caso contrario
 * Tiempo: O(log MOD)
 */
ll modInverse(ll b) { return modpow(b, MOD - 2, MOD) % MOD; }

ll modInverse(ll a) {
    ll x, y, d = euclid(a, MOD, x, y);
    assert(d == 1);
    return (x + MOD) % MOD;
}
```

3.8 Numeros Primos

```
/**
 * Descripcion: Estos 2 algoritmos encuentran por medio de
 * la Criba
 * de Eratostenes todos los numeros primos menor o iguales a
 * n, difieren
 * por su estrategia y por consecuente su complejidad
 * temporal.
 * Tiempo metodo #1: O(n log(log n))
 * Tiempo metodo #2: O(n)
 */
ll sieve_size;
vl primes;
void sieve(int n) {
    vector<bool> is_prime(n + 1, 1);

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

void sieve(int N) {
    vector<int> lp(N + 1);
    vector<int> pr;

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

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

// Primos menores a 1000:
//      2   3   5   7   11   13   17   19   23
//      29  31  37
//      41  43  47  53  59  61  67  71  73
//      79  83  89
//      97 101 103 107 109 113 127 131 137
//      139 149 151
//      157 163 167 173 179 181 191 193 197
//      199 211 223
//      227 229 233 239 241 251 257 263 269
//      271 277 281
```



```
// 283 293 307 311 313 317 331 337 347
// 349 353 359
// 367 373 379 383 389 397 401 409 419
// 421 431 433
// 439 443 449 457 461 463 467 479 487
// 491 499 503
// 509 521 523 541 547 557 563 569 571
// 577 587 593
// 599 601 607 613 617 619 631 641 643
// 647 653 659
// 661 673 677 683 691 701 709 719 727
// 733 739 743
// 751 757 761 769 773 787 797 809 811
// 821 823 827
// 829 839 853 857 859 863 877 881 883
// 887 907 911
// 919 929 937 941 947 953 967 971 977
// 983 991 997

// Otros primos:
// El primo mas grande menor que 10 es 7.
// El primo mas grande menor que 100 es 97.
// El primo mas grande menor que 1000 es 997.
// El primo mas grande menor que 10000 es 9973.
// El primo mas grande menor que 100000 es 99991.
// El primo mas grande menor que 1000000 es 999983.
// El primo mas grande menor que 10000000 es 9999991.
// El primo mas grande menor que 100000000 es 99999989.
// El primo mas grande menor que 1000000000 es 999999937.
// El primo mas grande menor que 10000000000 es
9999999967.
// El primo mas grande menor que 100000000000 es
99999999977.
// El primo mas grande menor que 1000000000000 es
999999999989.
// El primo mas grande menor que 10000000000000 es
9999999999971.
// El primo mas grande menor que 100000000000000 es
9999999999973.
// El primo mas grande menor que 1000000000000000 es
99999999999989.
// El primo mas grande menor que 10000000000000000 es
99999999999937.
// El primo mas grande menor que 100000000000000000 es
99999999999997.
// El primo mas grande menor que 1000000000000000000 es
999999999999989.
```

3.9 Simpson

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

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

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

4 Strings

4.1 Aho-Corasick

```
/*
 * Descripción: Este algoritmo te permite buscar rapidamente
 * multiples patrones en un texto
 * Tiempo: O(mk)
 */

// Utilizar esta implementacion cuando las letras permitidas
// sean pocas
struct AhoCorasick {
    enum { alpha = 26,
           first = 'a' }; // change this!
    struct Node {
        // (nmatches is optional)
        int back, next[alpha], start = -1, end = -1, nmatches = 0;
        Node(int v) { memset(next, v, sizeof(next)); }
    };
    vector<Node> N;
    vi backp;
    void insert(string& s, int j) {
        assert(!s.empty());
        int n = 0;
        for (char c : s) {
            int& m = N[n].next[c - first];
            if (m == -1) {
                n = m = SZ(N);
                N.emplace_back(-1);
            } else {
                n = m;
            }
            if (N[n].end == -1) N[n].start = j;
            backp.push_back(N[n].end);
            N[n].end = j;
            N[n].nmatches++;
        }
        // O(sum(pat) * C)
        AhoCorasick(vector<string>& pat) : N(1, -1) {
            FOR(i, 0, SZ(pat)) {
                insert(pat[i], i);
                N[0].back = SZ(N);
                N.emplace_back(0);
            }
        }
        queue<int> q;
        for (q.push(0); !q.empty(); q.pop()) {
            int n = q.front(), prev = N[n].back;
            FOR(i, 0, alpha) {
                int &ed = N[n].next[i], y = N[prev].next[i];
                if (ed == -1) {
                    ed = y;
                } else {
                    N[ed].back = y;
                    (N[ed].end == -1 ? N[ed].end : backp[N[ed].start])
                        = N[y].end;
                    N[ed].nmatches += N[y].nmatches;
                    q.push(ed);
                }
            }
        }
    }
    // O(|word|)
    vi find(string word) {
        int n = 0;
        vi res; // ll count = 0;
        for (char c : word) {
            n = N[n].next[c - first];
            res.push_back(N[n].end);
            // count += N[n].nmatches;
        }
        return res;
    }
    vector<vi> findAll(vector<string>& pat, string word) {
        vi r = find(word);
```

```
        vector<vi> res(SZ(word));
        FOR(i, 0, SZ(word)) {
            int ind = r[i];
            while (ind != -1) {
                res[i - SZ(pat[ind]) + 1].push_back(ind);
                ind = backp[ind];
            }
        }
        return res;
    }
};

class Aho {
    struct Vertex {
        unordered_map<char, int> children;
        bool leaf;
        int parent, suffixLink, wordID, endWordLink;
        char parentChar;
    };
    Vertex() {
        children.clear();
        leaf = false;
        parent = suffixLink = wordID = endWordLink = -1;
    }
private:
    vector<Vertex*> Trie;
    vector<int> wordsLength;
    int size, root;
    void calcSuffixLink(int vertex) {
        // Procesar root
        if (vertex == root) {
            Trie[vertex]->suffixLink = root;
            Trie[vertex]->endWordLink = root;
            return;
        }
        // Procesamiento de hijos de la raiz
        if (Trie[vertex]->parent == root) {
            Trie[vertex]->suffixLink = root;
            if (Trie[vertex]->leaf) {
                Trie[vertex]->endWordLink = vertex;
            } else {
                Trie[vertex]->endWordLink =
                    Trie[Trie[vertex]->suffixLink]->endWordLink;
            }
            return;
        }
        // Para calcular el suffix link del vertice actual,
        // necesitamos el suffix link
        // del padre del vertice y el personaje que nos movio al
        // vertice actual.
        int curBetterVertex = Trie[Trie[vertex]->parent]->
            suffixLink;
        char chVertex = Trie[vertex]->parentChar;
        while (true) {
            if (Trie[curBetterVertex]->children.count(chVertex)) {
                Trie[vertex]->suffixLink = Trie[curBetterVertex]->
                    children[chVertex];
                break;
            }
            if (curBetterVertex == root) {
                Trie[vertex]->suffixLink = root;
                break;
            }
            curBetterVertex = Trie[curBetterVertex]->suffixLink;
        }
        if (Trie[vertex]->leaf) {
            Trie[vertex]->endWordLink = vertex;
        } else {
            Trie[vertex]->endWordLink = Trie[Trie[vertex]->
                suffixLink]->endWordLink;
        }
    }
public:
```

```
Aho() {
    size = root = 0;
    Trie.pb(new Vertex());
    size++;
}

void addString(string s, int wordID) {
    int curVertex = root;
    FOR(i, 0, s.length()) { // Iteracion sobre los
        caracteres de la cadena
        char c = s[i];
        if (!Trie[curVertex]->children.count(c)) {
            Trie.pb(new Vertex());
            Trie[size]->suffixLink = -1;
            Trie[size]->parent = curVertex;
            Trie[size]->parentChar = c;
            Trie[curVertex]->children[c] = size;
            size++;
        }
        curVertex = Trie[curVertex]->children[c]; // Mover al
            nuevo vertice en el trie
    }
    // Marcar el final de la palabra y almacene su ID
    Trie[curVertex]->leaf = true;
    Trie[curVertex]->wordID = wordID;
    wordsLength.pb(s.length());
}

void prepareAho() {
    queue<int> vertexQueue;
    vertexQueue.push(root);
    while (!vertexQueue.empty()) {
        int curVertex = vertexQueue.front();
        vertexQueue.pop();
        calcSuffixLink(curVertex);
        for (auto key : Trie[curVertex]->children) {
            vertexQueue.push(key.second);
        }
    }
}

int processString(string text) {
    int currentState = root;
    int result = 0;
    FOR(j, 0, text.length()) {
        while (true) {
            if (Trie[currentState]->children.count(text[j])) {
                currentState = Trie[currentState]->children[text[j]]];
            } else {
                break;
            }
            if (currentState == root) break;
            currentState = Trie[currentState]->suffixLink;
        }
        int checkState = currentState;
        // Tratar de encontrar todas las palabras posibles de
        // este prefijo
        while (true) {
            checkState = Trie[checkState]->endWordLink;
            // Si estamos en el vertice raiz, no hay mas
            // coincidencias
            if (checkState == root) break;
            result++;
            int indexOfMatch = j + 1 - wordsLength[Trie[
                checkState]->wordID];
            // Tratando de encontrar todos los patrones
            // combinados de menor longitud
            checkState = Trie[checkState]->suffixLink;
        }
    }
    return result;
}

int main() {
    ios_base::sync_with_stdio(0);
```

```

cin.tie(nullptr);

vector<string> patterns = {"abc", "bcd", "abcd"};
string text = "abcd";
Aho ahoAlg;
FOR(i, 0, patterns.size()) { ahoAlg.addString(patterns[i], i); }
ahoAlg.prepareAho();
cout << ahoAlg.processString(text) << endl;

return 0;
}

```

4.2 Dynamic Aho-Corasick

```

/*
 * Descripcion: Si tenemos N cadenas en el diccionario,
 * mantenga log(N) Aho Corasick
 * automatat. El i-esimo automata contiene las primeras 2^k
 * cadenas no incluidas en el
 * automatat anteriores. Por ejemplo, si tenemos N = 19,
 * necesitamos 3 automatat: {s[1]...s[16]},
 * {s[17]...s[18]} y {s[19]}. Para responder a la consulta,
 * podemos atravesar los automatat logN.
 * utilizando la cadena de consulta dada.
 * Para manejar la insercion, primero construya un automata
 * usando una sola cadena y luego
 * Si bien hay dos automatat con el mismo numero de cadenas,
 * los fusionamos mediante
 * un nuevo automata usando fuerza bruta.
 * Para manejar la eliminacion, simplemente insertamos un
 * valor -1 para almacenar en los puntos finales de cada
 * cadena agregada.
 * Tiempo: O(m*log(numero_de_inserciones))
 */

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

        Node() = default;
    };

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

    explicit AhoCorasick() : states(1) {}

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

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

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

```

```

}

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

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

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

            que.push(j);
        }
    }

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

    // list of (id, index)
    vector<pair<int, int>> match(const string& str) const {
        vector<pair<int, int>> ret;
        int i = 0;
        for (int k = 0; k < (int)str.size(); ++k) {
            char c = str[k];
            i = get_next(i, c);
            for (auto id : states[i].accept) {
                ret.emplace_back(id, k);
            }
        }
        return ret;
    }

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

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

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

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

        ac[k].build();
    }
}

```

```

long long count(const string& str) const {
    long long ret = 0;
    for (int i = 0; i < (int)ac.size(); ++i) ret += ac[i].count(str);
    return ret;
}
};

```

4.3 Hashing

```

/*
 * Hashing
 * Descripcion: El objetivo es convertir una cadena en un
 * numero entero
 * para poder comparar cadenas en O(1)
 * Tiempo: O(|s|)
 */

const int MX = 3e5 + 2; // Tamano maximo del string S

inline int add(int a, int b, const int &mod) { return a + b >= mod ? a + b - mod : a + b; }
inline int sbt(int a, int b, const int &mod) { return a - b < 0 ? a - b + mod : a - b; }
inline int mul(int a, int b, const int &mod) { return 1ll * a * b % mod; }

const int X[] = {257, 359};
const int MOD[] = {(int)1e9 + 7, (int)1e9 + 9};
vector<int> xpow[2];

struct hashing {
    vector<int> h[2];

    hashing(string &s) {
        int n = s.size();
        for (int j = 0; j < 2; ++j) {
            h[j].resize(n + 1);
            for (int i = 1; i <= n; ++i) {
                h[j][i] = add(mul(h[j][i - 1], X[j], MOD[j]), s[i - 1], MOD[j]);
            }
        }

        // Hash del substring en el rango [i, j]
        ll value(int l, int r) {
            int a = sbt(h[0][r], mul(h[0][l], xpow[0][r - l], MOD[0]), MOD[0]);
            int b = sbt(h[1][r], mul(h[1][l], xpow[1][r - l], MOD[1]), MOD[1]);
            return (ll(a) << 32) + b;
        }
    };

    // Llamar la funcion antes del hashing
    void calc_xpow(int mxlen = MX) {
        for (int j = 0; j < 2; ++j) {
            xpow[j].resize(mxlen + 1, 1);
            for (int i = 1; i <= mxlen; ++i) {
                xpow[j][i] = mul(xpow[j][i - 1], X[j], MOD[j]);
            }
        }
    }
}

```

4.4 KMP

```

/*
 * Descripcion: El prefix function para un string S es
 * definido como un arreglo phi donde phi[i] es
 * la longitud del prefijo propio de S mas largo de la
 * subcadena S[0..i] el cual tambien
 * es sufixo de esta subcadena
 */

```

```

* Tiempo:  $O(|s| + |pat|)$ 
*/

vi PI(const string& s) {
    vi p(SZ(s));
    FOR(i, 1, SZ(s)) {
        int g = p[i - 1];
        while (g && s[i] != s[g]) g = p[g - 1];
        p[i] = g + (s[i] == s[g]);
    }
    return p;
}

// Concatena s + \0 + pat para encontrar las ocurrencias
vi KMP(const string& s, const string& pat) {
    vi phi = PI(pat + '\0' + s), res;
    FOR(i, SZ(phi) - SZ(s), SZ(phi))
        if (phi[i] == SZ(pat)) res.push_back(i - 2 * SZ(pat));
    return res;
}

// A partir del phi de patron busca las ocurrencias en s
int KMP(const string& s, const string& pat) {
    vi phi = PI(pat);
    int matches = 0;
    for (int i = 0, j = 0; i < SZ(s); ++i) {
        while (j > 0 && s[i] != pat[j]) j = phi[j - 1];
        if (s[i] == pat[j]) ++j;
        if (j == SZ(pat)) {
            matches++;
            j = phi[j - 1];
        }
    }
    return matches;
}

/*
* Automaton KMP
* El estado en el es el valor actual de la prefix function,
* y la transicion de un
* estado a otro se realiza a traves del siguiente caracter
* Uso: aut[state][nextCharacter]
* Tiempo:  $O(|s| * C)$ 
*/
// Automaton  $O(|s| * C)$ 
vector<vector<int>>> aut;
void compute_automaton(string s) {
    s += '#';
    int n = s.size();
    vector<int> phi = PI(s);
    aut.assign(n, vector<int>(26));
    FOR(i, 0, n) {
        FOR(c, 0, 26) {
            if (i > 0 && 'a' + c != s[i])
                aut[i][c] = aut[phi[i - 1]][c];
            else
                aut[i][c] = i + ('a' + c == s[i]);
        }
    }
}

```

4.5 Manacher

```

/*
* Descripcion: longitud del palindromo mas grande centrado
* en cada caracter de la cadena
* y entre cada par consecutivo
* Tiempo:  $O(n)$ 
*/

vi manacher(string _S) {
    string S = char(64);
    for (char c : _S) S += c, S += char(35);
    S.back() = char(38);
    vi ans(SZ(S) - 1);
    int lo = 0, hi = 0;

```

```

FOR(i, 1, SZ(S) - 1) {
    if (i != 1) ans[i] = min(hi - i, ans[hi - i + lo]);
    while (S[i - ans[i] - 1] == S[i + ans[i] + 1]) ++ans[i];
    if (i + ans[i] > hi) lo = i - ans[i], hi = i + ans[i];
}
ans.erase(begin(ans));
FOR(i, 0, SZ(ans))
    if (i % 2 == ans[i] % 2) ++ans[i];
return ans;
}

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

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

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

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

    /*
    * Retorna el lower_bound de la subcadena sub en el Suffix
    * Array
    * Tiempo:  $O(|sub| \log n)$ 

```

4.6 Suffix Array

```

*/
int lower(string &sub) {
    int l = 0, r = n - 1;
    while (l < r) {
        int mid = (l + r) / 2;
        int res = S.compare(SA[mid], SZ(sub), sub);
        (res >= 0) ? r = mid : l = mid + 1;
    }
    return l;
}

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

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

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

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

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

```

```

    return S.substr(indice, maximo);
}

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

```

4.7 Suffix Automaton

```

/*
 * Descripcion: Construye un automata finito que reconoce
 * todos los
 * sufijos de una cadena. len corresponde a la longitud
 * maxima de una
 * cadena en la clase de equivalencia, pos corresponde a la
 * primera
 * posicion final de dicha cadena, lnk corresponde al sufijo
 * mas largo
 * que esta en una clase diferente. Los enlaces de sufijos
 * corresponden
 * al arbol de sufijos de la cadena invertida
 * Tiempo: O(n log sum)
 */

```

```

struct SuffixAutomaton {
    int N = 1;
    vi lnk(-1), len(0), pos(-1); // suffix link,
    // max length of state, last pos of first occurrence of
    // state
    vector<map<char, int>> nex{1};
    vector<bool> isClone(0);
    // transitions, cloned -> not terminal state
    vector<vi> iLnk; // inverse links
    int add(int p, char c) { // *p nonzero if p != -1
        auto getNext = [&]() {
            if (p == -1) return 0;
            int q = nex[p][c];
            if (len[p] + 1 == len[q]) return q;
            int clone = N++;
            lnk.pb(lnk[q]);
            lnk[q] = clone;
            len.pb(len[p] + 1), nex.pb(nex[q]),
            pos.pb(pos[q]), isClone.pb(1);
            for (; p && nex[p][c] == q; p = lnk[p]) nex[p][c] =
            clone;
            return clone;
        };
        // if (nex[p].count(c)) return getNext();
        // ^ need if adding > 1 string
        int cur = N++; // make new state
        lnk.emplace_back(), len.pb(len[p] + 1), nex.emplace_back
        (),
        pos.pb(pos[p] + 1), isClone.pb(0);
        for (; p && !nex[p].count(c); p = lnk[p]) nex[p][c] =
        cur;
        int x = getNext();
        lnk[cur] = x;
        return cur;
    }
    void init(string s) {
        int p = 0;
        for (char x : s) p = add(p, x);
    } // add string to automaton
    // inverse links
}

```

```

void genIlnk() {
    iLnk.resize(N);
    FOR(v, 1, N)
        iLnk[lnk[v]].pb(v);
}

// APPLICATIONS
void getAllOccur(vi& oc, int v) {
    if (!isClone[v]) oc.pb(pos[v]); // terminal position
    for (auto u : iLnk[v]) getAllOccur(oc, u);
}

vi allOccur(string s) { // get all occurrences of s in
    automaton
    int cur = 0;
    for (char x : s) {
        if (!nex[cur].count(x)) return {};
        cur = nex[cur][x];
    }
    // convert end pos -> start pos
    vi oc;
    getAllOccur(oc, cur);
    for (auto t : oc) t += 1 - SZ(s);
    sort(ALL(oc));
    return oc;
}

vector<ll> distinct;
ll getDistinct(int x) {
    // # distinct strings starting at state x
    if (distinct[x]) return distinct[x];
    distinct[x] = 1;
    for (auto y : nex[x]) distinct[x] += getDistinct(y.
    second);
    return distinct[x];
}

ll numDistinct() { // # distinct substrings including
    empty
    distinct.resize(N);
    return getDistinct(0);
}

ll numDistinct2() { // assert(numDistinct() == numDistinct2
    ());
    ll ans = 1;
    FOR(i, 1, N)
        ans += len[i] - len[lnk[i]];
    return ans;
}

SuffixAutomaton S;
vi sa;
string s;
void dfs(int x) {
    if (!S.isClone[x]) sa.pb(SZ(s) - 1 - S.pos[x]);
    vector<pair<char, int>> chr;
    for (auto t : S.iLnk[x]) chr.pb({s[S.pos[t] - S.len[x]], t
    });
    sort(ALL(chr));
    for (auto t : chr) dfs(t.second);
}

int main() {
    reverse(ALL(s));
    S.init(s);
    S.genIlnk();
    dfs(0);
}

```

4.8 Suffix Tree

```

/**
 * Descripcion: Algoritmo de Ukkonen para arbol de sufijos.
 * El sufijo no unico
 * mas largo de S tiene longitud len[p]+lef despues de cada
 * llamada a add.
 * Cada iteracion del bucle dentro de add esta cantidad
 * disminuye en uno
 * Tiempo: O(n log sum)
 */

```

```

*/
struct SuffixTree {
    string s;
    int N = 0;
    vi pos, len, lnk;
    vector<map<char, int>> to;

    SuffixTree(string _s) {
        make(-1, 0);
        int p = 0, lef = 0;
        for (char c : _s) add(p, lef, c);
        add(p, lef, 'S');
        s.pop_back(); // terminal char
    }

    int make(int POS, int LEN) { // lnk[x] is meaningful when
        // x!=0 and len[x] != MOD
        pos.pb(POS);
        len.pb(LEN);
        lnk.pb(-1);
        to.emplace_back();
        return N++;
    }

    void add(int& p, int& lef, char c) { // longest
        // non-unique suffix is at node p with lef extra chars
        s += c;
        ++lef;
        int lst = 0;
        for (; lef; p ? p = lnk[p] : lef--) { // if p != root
            then lnk[p]
            // must be defined
            while (lef > 1 && lef > len[to[p][s[SZ(s) - lef]]])
                p = to[p][s[SZ(s) - lef]], lef -= len[p];
            // traverse edges of suffix tree while you can
            char e = s[SZ(s) - lef];
            int& q = to[p][e];
            // next edge of suffix tree
            if (!q) q = make(SZ(s) - lef, MOD), lnk[lst] = p, lst
            = 0;
            // make new edge
            else {
                char t = s[pos[q] + lef - 1];
                if (t == c) {
                    lnk[lst] = p;
                    return;
                } // suffix not unique
                int u = make(pos[q], lef - 1);
                // new node for current suffix-1, define its link
                to[u][c] = make(SZ(s) - 1, MOD);
                to[u][t] = q;
                // new, old nodes
                pos[q] += lef - 1;
                if (len[q] != MOD) len[q] -= lef - 1;
                q = u, lnk[lst] = u, lst = u;
            }
        }
    }

    int maxPre(string x) { // max prefix of x which is
        substring
        for (int p = 0, ind = 0;;) {
            if (ind == SZ(x) || !to[p].count(x[ind])) return ind;
            p = to[p][x[ind]];
            FOR(i, 0, len[p]) {
                if (ind == SZ(x) || x[ind] != s[pos[p] + i]) return
                ind;
                ind++;
            }
        }
    }
    vi sa; // generate suffix array
    void genSa(int x = 0, int Len = 0) {
        if (!SZ(to[x]))
            sa.pb(pos[x] - Len); // found terminal node
        else
            for (auto t : to[x]) genSa(t.second, Len + len[x]);
    }
};

```

4.9 Trie

```

/*
 * Descripcion: Un trie es una estructura de datos de arbol
 * multidireccional
 * que se utiliza para almacenar cadenas en un alfabeto. La
 * coincidencia
 * de patrones se puede realizar de manera eficiente usando
 * trie
 * Tiempo: O(n)
 */

struct TrieNode {
    unordered_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;
    }
};

```

4.10 Z-Algorithm

```

/*
 * Descripcion: La Z-function es un arreglo donde el
 * elemento i es igual al numero mas
 * grande de caracteres que empiezan desde la posicion i que
 * coincide con el prefijo de S,
 * excepto Z[0] = 0. (abacaba -> 0010301)
 * Tiempo: O(|S|)
 */
vi Z(const string& S) {
    vi z(SZ(S));
    int l = -1, r = -1;
    FOR(i, 1, SZ(S)) {
        z[i] = i >= r ? 0 : min(r - i, z[i - l]);
        while (i + z[i] < SZ(S) && S[i + z[i]] == S[z[i]])
            z[i]++;
        if (i + z[i] > r)
            l = i, r = i + z[i];
    }
    return z;
}

```

5 Dynamic Programming

5.1 2D Sum

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

5.2 Tecnica con Deque

```
/**
 * Descripcion: algoritmo que resuelve el problema de el
 *            minimo
 * o maximo valor de cada sub-array de longitud fija.
 * Enunciado:
 * Dado un arreglo de numeros A de longitud n y un numero k
 * <= n.
 * Encuentra el minimo para cada sub-array contiguo de
 * longitud k.
 * La estrategia se basa en el uso de una bicola monotona,
 * en donde en cada iteracion sacamos del final de la bicola
 * hasta que este vacia o nos encontremos con un A[j] > A[i]
 * ],
 * luego agregamos i, manteniendose de manera decreciente,
 * si el
 * frente se sale del rango, lo sacamos y el nuevo frente
 * seria
 * el mayor en el rango (A[i]...A[i + k - 1]).
 * Este algoritmo gana fuerza cuando se generaliza a mas
 * dimensiones:
 * digamos que queremos el mayor en una sub-matriz dada, se
 * puede
 * precalcular el B para cada fila y luego volvemos a correr
 * el algoritmo sobre dichos valores.
 * Retorna un vector B, en donde B[i] = j,
 * tal que A[j] >= A[i], ..., A[i + k - 1]
 * Tiempo: O(n)
 */
vector<int> solve(vector<int>& A, int k) {
    vector<int> B(A.size() - k + 1);
    deque<int> dq;
    for (int i = 0; i < A.size(); i++) {
        while (!dq.empty() && A[dq.back()] <= A[i])
            dq.pop_back();
        dq.pb(i);
        if (dq.front() <= i - k)
            dq.pop_front();

        if (i + 1 >= k)
            B[i + 1 - k] = A[dq.front()];
    }
}
```

```
}
}
```

5.3 DP con digitos

```
/**
 * Descripcion: algoritmo que resuelve un problema de DP de
 *            digitos.
 * La DP de digitos se requiere cuando se trabaja sobre
 *            cadenas
 *            (normalmente numeros) de una gran cantidad de digitos y
 *            se
 *            requiere saber cuantos numeros en un rango cumplen con
 *            cierta
 *            propiedad. Enunciado del problema resuelto:
 * Dada una cadena s que contiene numeros y caracteres ?
 * encontrar
 * el minimo entero, tal que se forme asignandole valores a
 * los ? y
 * ademas sea divisible por D; si no existe, imprimir un *
 * Tiempo: O(n^2)
 */
```

```
string s;
int D;
stack<int> st;
```

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

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

5.4 Knapsack

```
/**
 * Descripcion: algoritmo para resolver el problema de la
 *            mochila:
```

```
* se cuenta con una coleccion de N objetos donde cada uno
 * tiene
 * un peso y un valor asignado, y una mochila con capacidad
 * maxima C.
 * Se necesita maximizar la suma de valores que se puede
 * lograr
 * sin que se exceda C.
 * Tiempo: O(NC)
 */
```

```
int peso[MAXN], valor[MAXN], dp[MAXN][MAXC];
int N, C;

int solve(int i, int c) {
    if (c < 0)
        return -INF;
    if (i == N)
        return 0;
    int &ans = dp[i][c];
    if (ans != -1)
        return ans;

    return dp[i][c] = max(solve(i + 1, c), opcion2, valor[i] +
        solve(i + 1, c - peso[i]));
}
```

5.5 Longest Increasing Subsequence

```
/**
 * Descripcion: algoritmo para resolver el problema de la
 *            subsecuencia creciente mas larga de un arreglo (LIS) a
 *            partir de una estrategia de divide y venceras. Si no
 *            es necesario recuperar la subsecuencia, ignorar p.
 * Tiempo: O(n log n)
 */
```

```
int n, nums[MAX], L[MAX], L_id[MAX], p[MAX];

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 solve_LIS() {
    int lis_sz = 0, lis_end = 0;
    for (int i = 0; i < n; i++) {
        L[i] = L_id[i] = 0;
        p[i] = -1;

        for (int i = 0; i < n; i++) {
            int pos = lower_bound(L, L + lis_sz, nums[i]) - L;
            L[pos] = nums[i];
            L_id[pos] = i;

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

            if (pos == lis_sz) {
                lis_sz = pos + 1;
                lis_end = i;
            }
        }
        return lis_sz;
    }
}
```

5.6 Monotonic Stack

```
/*
```

```

* Descripcion: Usando la tecnica de la pila monotona para
    calcular para cada indice,
* el elemento menor a la izquierda
* Tiempo: O(n)
*/

```

```

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, 0, 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);
    }
}

```

5.7 Travelling Salesman Problem

```

/**
 * Descripcion: algoritmo para resolver el problema del
 * viajero (TSP):
 * consiste en encontrar un recorrido que visite todos los
 * vertices del
 * grafo, sin repeticiones y con el costo minimo. Este
 * codigo resuelve
 * una variante del TSP donde se puede comenzar en cualquier
 * vertice y
 * no necesita volver al inicial.
 * Tiempo: O(2^n * n)
 */

```

```

constexpr int MAX_NODES = 15;
int n, dist[MAX_NODES][MAX_NODES], dp[MAX_NODES][1 << (
    MAX_NODES + 1)];

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

    ans = INF;
    for (int k = 0; k < n; k++)
        if ((mask & (1 << k)) == 0)
            ans = min(ans, solve(k, mask | (1 << k)) + dist[i][k]);
    return ans;
}

int solveTSP() {
    int ans = INF;
    for (int i = 0; i < n; i++)
        ans = min(ans, solve(i, (1 << (i))));
    return ans;
}

```

6 Graphs

6.1 2SAT

```
/**
 * Descripcion: estructura para resolver el problema de
 * TwoSat:
 * dadas disyunciones del tipo (a or b) donde las variables
 * pueden
 * o no estar negadas, se necesita saber si es posible
 * asignarle un
 * valor a cada variable de tal modo que cada disyuncion se
 * cumpla.
 * Las variables negadas son representadas por inversiones
 * de bits (~x)
 *
 * Uso:
 * TwoSat ts(numero de variables booleanas);
 * ts.either(0, ~3);           La variable 0 es verdadera o
 *                             la variable 3 es falsa
 * ts.setValue(2);           La variable 2 es verdadera
 * ts.atMostOne({0, ~1, 2}); <= 1 de vars 0, ~1 y 2 son
 *                             verdadero
 * ts.solve();               Retorna verdadero si existe
 *                             solucion
 * ts.values[0..N-1]         Tiene los valores asignados a
 *                             las variables
 * Tiempo: O(N + E), donde N es el numero de variables
 *                             booleanas y E es el numero de clausulas
 */
```

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

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

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

```
    // Agregar una disyuncion
```

```
    void either(int x, int y) { // Nota: (a v b), es
        // equivalente a la expresion (~a -> b) n (~b -> a)
        x = max(2 * x, -1 - 2 * x); y = max(2 * y, -1 - 2 * y);
        adj[x].push_back(y ^ 1); adj[y].push_back(x ^ 1);
    }
```

```
    void setValue(int x) { either(x, x); }
    void implies(int x, int y) { either(~x, y); }
    void make_diff(int x, int y) {
        either(x, y);
        either(~x, ~y);
    }
```

```
    void make_eq(int x, int y) {
        either(~x, y);
        either(x, ~y);
    }
```

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

```
    vi dfs_num, comp;
    stack<int> st;
    int time = 0;
```

```
int tarjan(int u) {
    int x, low = dfs_num[u] = ++time;
    st.push(u);
    for (int v : adj[u])
        if (!comp[v])
            low = min(low, dfs_num[v] ? tarjan(v) :
                if (low == dfs_num[u]) {
                    do {
                        x = st.top();
                        st.pop();
                        comp[x] = low;
                        if (values[x >> 1] == -1)
                            values[x >> 1] = x & 1;
                        } while (x != u);
                    }
                return dfs_num[u] = low;
            }
    }
```

```
bool solve() {
    values.assign(N, -1);
    dfs_num.assign(2 * N, 0);
    comp.assign(2 * N, 0);
    for (int i = 0; i < 2 * N; i++)
        if (!comp[i])
            tarjan(i);
    for (int i = 0; i < N; i++)
        if (comp[2 * i] == comp[2 * i + 1])
            return 0;
    return 1;
}
```

6.2 Bridge Detection

```
/**
 * Descripcion: algoritmo para buscar puentes en un grafo
 * Tiempo: O(V + E)
 */
```

```
int n;
vector<int> g[MAXN];
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 v : g[u]) {
        if (v == p)
            continue;
        if (tin[v] != -1)
            low[u] = min(low[u], tin[v]);
        else {
            if (u == dfsRoot) // La raiz es un punto de
                articulation
            ++rootChildren;
        }
    }
    dfs(v, u);
}
```

```
    if (low[v] >= tin[u])
        articulation[u] = 1;
    if (low[v] > tin[u])
        // La arista (u, v) es un puente
        low[u] = min(low[u], low[v]);
    }
```

```
void find_bridges_articulations() {
    memset(tin, tin + n, -1);
    memset(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);
        }
    }
```

6.3 Kosaraju (SCC)

```
/**
 * Descripcion: sirve para la busqueda de componentes
 * fuertemente conexos (SCC),
 * este 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.
 * Tiempo: O(V + E)
 */
```

```
vi graph[MAXN]; // Grafo
vi graph_T[MAXN]; // Grafo transpuesto
vi dfs_num;
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;
}
```

6.4 Tarjan (SCC)

```
/**
 * Descripcion: sirve para la busqueda de componentes
 * fuertemente conexos (SCC)
 * 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
 * Explicacion: 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
* Tiempo:  $O(V + E)$ 
*/

```

```

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);
    }
}

```

6.5 General Matching

```

/**
* Descripcion: Variante de la implementacion de Gabow para
  el algoritmo
* de Edmonds-Blossom. Maximo emparejamiento sin peso para
  un grafo en
* general, con l-indexacion. Si despues de terminar la
  llamada a solve(),
* white[v] = 0, v es parte de cada matching maximo.
* Tiempo:  $O(NM)$ , mas rapido en la practica.
*/

```

```

struct MaxMatching {
    int N;
    vector<vi> adj;
    vector<int> mate, first;
    vector<bool> white;
    vector<pi> label;

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

```

```

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

```

```

int group(int x) {
    if (white[first[x]])
        first[x] = group(first[x]);
    return first[x];
}

void match(int p, int b) {
    swap(b, mate[p]);
    if (mate[b] != p)
        return;
    if (!label[p].second)
        mate[b] = label[p].first, match(label[p].first, b);
    // vertex label
    else
        match(label[p].first, label[p].second), match(label[p]
            .second, label[p].first); // edge label
}

```

```

bool augment(int st) {
    assert(st);
    white[st] = 1;
    first[st] = 0;
    label[st] = {0, 0};

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

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

```

```

int solve() {
    int ans = 0;
    FOR(st, 1, N + 1)
        if (!mate[st])
            ans += augment(st);
    FOR(st, 1, N + 1)
        if (!mate[st] && !white[st])
            assert(!augment(st));
}

```

```

return ans;
}
};

```

6.6 Hopcroft Karp

```

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

```

```

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

```

```

bool bfs() {
    queue<int> q;
}

```

```

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

```

```

dist[0] = INF;

```

```

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

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

```

```

return dist[0] != INF;
}

```

```

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

```

```

int hopcroftKarp() {
    int result = 0;

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

    return result;
}

```

6.7 Hungaro

```
/**
 * Descripcion: Dado un grafo bipartito ponderado, empareja
 * cada nodo
 * en la izquierda con un nodo en la derecha, tal que ningun
 * nodo
 * pertenece a 2 emparejamientos y que la suma de los pesos
 * de las
 * aristas usadas es minima. Toma a[N][M], donde a[i][j] es
 * el costo de emparejar L[i] con R[j], retorna (costo
 * minimo, match),
 * donde L[i] es emparejado con R[match[i]], negar costos si
 * se requiere
 * el emparejamiento maximo, se requiere que N <= M.
 * Tiempo: O(N^2 M)
 */
pair<int, vi> hungarian(const vector<vi> &a) {
    if (a.empty()) return {0, {}};
    int n = SZ(a) + 1, m = SZ(a[0]) + 1;
    vi u(n), v(m), p(m), ans(n - 1);
    FOR(i, 1, n) {
        p[0] = i;
        int j0 = 0; // add "dummy" worker 0
        vi dist(m, INT_MAX), pre(m, -1);
        vector<bool> done(m + 1);
        do { // dijkstra
            done[j0] = true;
            int i0 = p[j0], j1, delta = INT_MAX;
            FOR(j, 1, m)
                if (!done[j]) {
                    auto cur = a[i0 - 1][j - 1] - u[i0] - v[j];
                    if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
                    if (dist[j] < delta) delta = dist[j], j1 = j;
                }
            FOR(j, 0, m) {
                if (done[j])
                    u[p[j]] += delta, v[j] -= delta;
                else
                    dist[j] -= delta;
            }
            j0 = j1;
        } while (p[j0]);
        while (j0) { // update alternating path
            int j1 = pre[j0];
            p[j0] = p[j1], j0 = j1;
        }
        FOR(j, 1, m)
            if (p[j]) ans[p[j] - 1] = j - 1;
        return {-v[0], ans}; // min cost
    }
}
```

6.8 Kuhn

```
/**
 * Descripcion: Soluciona el problema de maximo
 * emparejamiento bipartito,
 * se basa en el algoritmo que puede ser pensado como n DFS
 * siendo ejecutadas.
 * Tiempo: O(n^2)
 */
int n, k;
vector<vector<int>> g;
vector<int> mt;
vector<bool> used;

bool try_kuhn(int v) {
    if (used[v])
        return false;
    used[v] = true;
    for (int to : g[v]) {
        if (mt[to] == -1 || try_kuhn(mt[to])) {
```

```
            mt[to] = v;
            return true;
        }
    }
    return false;
}

int main() {
    //... reading the graph ...

    mt.assign(k, -1);
    int ans = 0;
    for (int v = 0; v < n; ++v) {
        used.assign(n, false);
        if (try_kuhn(v)) ans++;
    }

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

6.9 Kruskal (MST)

```
/**
 * Descripcion: tiene como principal funcion calcular la
 * suma del
 * peso de las aristas del arbol minimo de expansion (MST)
 * de un grafo,
 * la estrategia es ir construyendo gradualmente el MST,
 * donde
 * iterativamente se coloca la arista disponible con menor
 * peso y
 * ademas no conecte 2 nodos que pertenezcan al mismo
 * componente.
 * Tiempo: O(E log E)
 */

#include <../Data Structure/DSU.h>

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

int main() {
    int V, E;
    cin >> V >> E;

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

    for (int i = 0; i < E; i++) {
        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];
        if (!dsu.sameSet(u, v)) {
            totalWeight += w;
            V -= dsu.unite(u, v);
        }
    }
    cout << "MST weight: " << totalWeight << '\n';
}
```

6.10 Prim (MST)

```
/**
```

```
 * Descripcion: tiene como principal funcion calcular la
 * suma del
 * peso de las aristas del arbol minimo de expansion (MST)
 * de un grafo,
 * la estrategia es ir construyendo gradualmente el MST, se
 * selecciona un
 * nodo arbitrario y se agregan sus aristas con nodos que no
 * hayan
 * sido agregados con anterioridad y se va tomando la de
 * menor peso hasta
 * completar el MST.
 * Tiempo: O(E log E)
 */
```

```
int V, E;
vector<pi> graph[MAXN];
bool taken[MAXN];
priority_queue<pi> pq;

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

```
int prim() {
    process(0);
    int totalWeight = 0, takenEdges = 0;
    while (!pq.empty() && takenEdges != V - 1) {
        auto [w, u] = pq.top();
        pq.pop();

        if (taken[u]) continue;

        totalWeight -= w;
        process(u);
        ++takenEdges;
    }
    return totalWeight;
}
```

6.11 Dinic

```
/**
 * Descripcion: algoritmo para calcular el flujo maximo en
 * un grafo
 * Tiempo: O(V^2 E)
 */

struct Dinic {
    struct Edge {
        int to, rev;
        ll c, oc;
        ll flow() { return max(oc - c, 0LL); } // if you need
        flows
    };
    vi lvl, ptr, q;
    vector<vector<Edge>> adj;
    Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
    void addEdge(int a, int b, ll c, ll rcap = 0) {
        adj[a].push_back({b, SZ(adj[b]), c, c});
        adj[b].push_back({a, SZ(adj[a]) - 1, rcap, rcap});
    }
    ll dfs(int v, int t, ll f) {
        if (v == t || !f) return f;
        for (int& i = ptr[v]; i < SZ(adj[v]); i++) {
            Edge& e = adj[v][i];
            if (lvl[e.to] == lvl[v] + 1)
                if (ll p = dfs(e.to, t, min(f, e.c))) {
                    e.c -= p, adj[e.to][e.rev].c += p;
                    return p;
                }
        }
        return 0;
    }
}
```

```

ll calc(int s, int t) {
    ll flow = 0;
    q[0] = s;
    FOR(L, 0, 31)
    do { // 'int L=30' maybe faster for random data
        lvl = ptr = vi(SZ(q));
        int qi = 0, qe = lvl[s] = 1;
        while (qi < qe && !lvl[t]) {
            int v = q[qi++];
            for (Edge e : adj[v])
                if (!lvl[e.to] && e.c >> (30 - L))
                    q[qi++] = e.to, lvl[e.to] = lvl[v] + 1;
        }
        while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
    }
    while (lvl[t])
        ;
    return flow;
}

bool leftOfMinCut(int a) { return lvl[a] != 0; }
};

```

6.12 Min Cost Max Flow

```

/**
 * Descripcion: maximo flujo de coste minimo. Se permite que
 * cap[i][j] != cap[j][i], pero
 * las aristas dobles no lo estan, si los costos pueden ser
 * negativos, llamar a setpi antes
 * que calc, los ciclos con costos negativos no son
 * soportados.
 * Tiempo: aproximadamente  $O(E^2)$ 
 */

```

```

// #include <bits/extc++.h> importante de incluir

```

```

const ll INF = numeric_limits<ll>::max() / 4;
typedef vector<ll> VL;

```

```

struct MCMF {
    int N;
    vector<vi> ed, red;
    vector<VL> cap, flow, cost;
    vi seen;
    VL dist, pi;
    vector<pair<ll, ll>> par;

```

```

    MCMF(int N) : N(N), ed(N), red(N), cap(N, VL(N)), flow(cap),
        cost(cap), seen(N), dist(N), pi(N), par(N) {}

```

```

    void addEdge(int from, int to, ll cap, ll cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
        ed[from].push_back(to);
        red[to].push_back(from);
    }

```

```

    void path(int s) {
        fill(ALL(seen), 0);
        fill(ALL(dist), INF);
        dist[s] = 0;
        ll di;

```

```

        __gnu_pbds::priority_queue<pair<ll, int>> q;
        vector<decltype(q)::point_iterator> its(N);
        q.push({0, s});

```

```

        auto relax = [&](int i, ll cap, ll cost, int dir) {
            ll val = di - pi[i] + cost;
            if (cap && val < dist[i]) {
                dist[i] = val;
                par[i] = {s, dir};
                if (its[i] == q.end())
                    its[i] = q.push({-dist[i], i});
                else
                    q.modify(its[i], {-dist[i], i});
            }

```

```

        }
    };

    while (!q.empty()) {
        s = q.top().second;
        q.pop();
        seen[s] = 1;
        di = dist[s] + pi[s];
        for (int i : ed[s])
            if (!seen[i])
                relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
        for (int i : red[s])
            if (!seen[i])
                relax(i, flow[i][s], -cost[i][s], 0);
    }
    FOR(i, 0, N)
        pi[i] = min(pi[i] + dist[i], INF);
}

pair<ll, ll> calc(int s, int t) {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
        ll fl = INF;
        for (int p, r, x = t; tie(p, r) = par[x], x != s; x = p)
            fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p]);
        totflow += fl;
        for (int p, r, x = t; tie(p, r) = par[x], x != s; x = p)
            if (r)
                flow[p][x] += fl;
            else
                flow[x][p] -= fl;
    }
    FOR(i, 0, N)
        FOR(j, 0, N) totcost += cost[i][j] * flow[i][j];
    return {totflow, totcost};
}

void setpi(int s) {
    fill(ALL(pi), INF);
    pi[s] = 0;
    int it = N, ch = 1;
    ll v;
    while (ch-- && it--)
        FOR(i, 0, N)
            if (pi[i] != INF) for (int to : ed[i]) if (cap[i][to])
                if ((v = pi[i] + cost[i][to]) < pi[to])
                    pi[to] = v,
                    ch = 1;
    assert(it >= 0);
}
};

```

6.13 Push Relabel

```

/**
 * Descripcion: algoritmo push-relabel para calcular el
 * flujo maximo en un grafo, bastante rapido en la practica
 * Tiempo:  $O(V^2 \sqrt{E})$ 
 */

```

```

struct PushRelabel {
    struct Edge {
        int dest, back;
        ll f, c;
    };
    vector<vector<Edge>> g;
    vector<ll> ec;
    vector<Edge*> cur;
    vector<vi> hs;
    vi H;
    PushRelabel(int n) : g(n), ec(n), cur(n), hs(2 * n), H(n) {}
};

```

```

void addEdge(int s, int t, ll cap, ll rcap = 0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0, cap});
    g[t].push_back({s, sz(g[s]) - 1, 0, rcap});
}

```

```

void addFlow(Edge& e, ll f) {
    Edge& back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f;
    e.c -= f;
    ec[e.dest] += f;
    back.f -= f;
    back.c += f;
    ec[back.dest] -= f;
}

```

```

ll calc(int s, int t) {
    int v = sz(g);
    H[s] = v;
    ec[t] = 1;
    vi co(2 * v);
    co[0] = v - 1;
    rep(i, 0, v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);

```

```

    for (int hi = 0;;) {
        while (hs[hi].empty())
            if (!hi--) return -ec[s];
        int u = hs[hi].back();
        hs[hi].pop_back();
        while (ec[u] > 0) // discharge u
            if (cur[u] == g[u].data() + sz(g[u])) {
                H[u] = 1e9;
                for (Edge& e : g[u])
                    if (e.c && H[u] > H[e.dest] + 1)
                        H[u] = H[e.dest] + 1, cur[u] = &e;
                if (++co[H[u]], !--co[hi] && hi < v)
                    rep(i, 0, v) if (hi < H[i] && H[i] < v) --co[H[i]], H[i] = v + 1;
                hi = H[u];
            } else if (cur[u]->c && H[u] == H[cur[u]->dest] + 1)
                addFlow(*cur[u], min(ec[u], cur[u]->c));
            else
                ++cur[u];
    }
    bool leftOfMinCut(int a) { return H[a] >= sz(g); }
};

```

6.14 Bellman-Ford

```

/**
 * Descripcion: calcula el costo minimo para ir de un nodo
 * hacia todos los
 * demas alcanzables. Puede detectar ciclos negativos, dando
 * una ultima
 * pasada y revisando si alguna distancia se acorta.
 * Tiempo:  $O(VE)$ 
 */

```

```

int main() {
    int n, m, A, B, W;
    cin >> n >> m;
    tuple<int, int, int> edges[m];
    for (int i = 0; i < m; i++) {
        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 (int i = 0; i < n; i++) {
        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 (int i = 0; i < n; i++) {
    cout << (dist[i] == INF ? -1 : dist[i]) << " ";
}

return 0;
}

```

6.15 Dijkstra

```

/**
 * Descripcion: calcula el costo minimo para ir de un nodo
 * hacia todos los demas alcanzables.
 * Tiempo:  $O(E \log V)$ 
 */

vector<pi> graph[MAXN];
int dist[MAXN];

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

    priority_queue<pi> pq;
    pq.emplace(0, x);
    while (!pq.empty()) {
        auto [du, u] = pq.top();
        du *= -1;
        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);
            }
        }
    }
}

```

6.16 Floyd-Warshall

```

/**
 * Descripcion: modifica la matriz de adyacencia graph[n][n]
 * tal que graph[i][j] pasa a indicar el costo minimo para
 * ir
 * desde el nodo i al j, para cualquier (i, j).
 * Tiempo:  $O(n^3)$ 
 */
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]);
            }
        }
    }

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

```

6.17 Binary Lifting LCA

```

/**
 * Descripcion: siendo jump[i][j] el ancestro  $2^j$  del nodo i
 * el binary lifting nos permite obtener el k-esimo ancestro
 * de cualquier nodo en tiempo logaritmico, una aplicacion
 * de
 * esto es para obtener el ancestro comun mas bajo (LCA).
 * Importante inicializar jump[i][0] para todo i.
 * Tiempo:  $O(n \log n)$  en construccion y  $O(\log n)$  por
 * consulta
 */

const LOG_MAXN = 25;
int jump[MAXN][LOG_MAXN];
int depth[MAXN];

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

    dfs(0);

    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];
for (int i = LOG_MAXN - 1; i >= 0; i--)
    if ((dist >> i) & 1)
        p = jump[p][i];

if (p == q)
    return p;

for (int i = LOG_MAXN - 1; i >= 0; i--)
    if (jump[p][i] != jump[q][i]) {
        p = jump[p][i];
        q = jump[q][i];
    }

return jump[p][0];
}

```

6.18 Euler Tour

```

/**
 * Descripcion: utilizando una DFS, es posible aplanar un
 * arbol,
 * esto se logra guardando en que momento entra y sale cada
 * nodo,
 * apoyandonos de una estructura para consultas de rango es
 * muy
 * util para consultas sobre un subarbol: saber la suma de
 * todos los nodos en el, el nodo con menor valor, etc.
 * Tiempo:  $O(n)$ 
 */

vi g[MAXN];
int val[MAXN], in[MAXN], out[MAXN], toursz = 0;
void dfs(int u, int p) {
    in[u] = toursz++;

    for (auto& v : g[u])
        if (v != p)
            dfs(v, u);

    out[u] = toursz++;
}

```

6.19 Find Centroid

```

/**
 * Descripcion: dado un arbol, encuentra su centroide
 * Tiempo:  $O(V)$ 
 */

int dfs(int u, int p) {
    for (auto v : tree[u])
        if (v != p)
            subtreeSZ[u] += dfs(v, u);
    return subtreeSZ[u] += 1;
}

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

```

6.20 Hierholzer

```

/**
 * Descripcion: busca un camino euleriano en el grafo dado.
 */

```

```

* 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
* Tiempo: O(E)
*/

```

```

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;
}

```

```

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);
    }
}

assert(sorted_nodes.size() == V);
}

```

6.21 Orden Topologico

```

/**
 * Descripcion: algoritmo para obtener el orden topologico
   de
 * un grafo dirigido, definido como el ordenamiento de sus
 * vertices tal que para cada arista (u, v), u este antes
 * que v en el ordenamiento. Si existen ciclos, dicho
 * ordenamiento no existe.
 * Tiempo: O(V + E)
 */

```

```

int V;
vi graph[MAXN];
vi sorted_nodes;
bool visited[MAXN];

void dfs(int u) {
    visited[u] = true;
    for (auto v : graph[u])
        if (!visited[v])
            dfs(v);
    sorted_nodes.push(u);
}

void toposort() {
    for (int i = 0; i < V; i++)
        if (!visited[i])
            dfs(i);
    reverse(ALL(sorted_nodes));

    assert(sorted_nodes.size() == V);
}

void lexicographic_toposort() {
    priority_queue<int> q;
    for (int i = 0; i < V; i++)

```

7 Geometry

7.1 Punto

```
constexpr double EPS = 1e-9; // 1e-9 es suficiente para
                             problemas de precision doble
constexpr double PI = acos(-1.0);

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

typedef double T;
struct Point {
    T x, y;
    Point operator+(Point& p) const { return {x + p.x, y + p.y}
; }
    Point operator-(Point& p) const { return {x - p.x, y - p.y}
; }
    Point operator*(T& d) const { return {x * d, y * d}; }
    Point operator/(T& d) const { return {x / d, y / d}; } //
    Solo para punto flotante

    bool operator<(Point& other) const {
        if (fabs(x - other.x) > EPS)
            return x < other.x;
        return y < other.y;
    }
    bool operator==(Point& other) const { return fabs(x -
        other.x) <= EPS && fabs(y - other.y) <= EPS; }
    bool operator!=(Point& other) const { return !(this ==
        other); }
};

T sq(Point p) { return p.x * p.x + p.y * p.y; }
double abs(Point p) { return sqrt(sq(p)); }

// Para poder hacer cout << miPunto
ostream& operator<<(ostream& os, Point p) { return os << "("
    << p.x << ", " << p.y << ")"; }

// Ejemplos de uso
Point a{3, 4}, b{2, -1};
cout << a + b << " " << a - b << "\n"; // (5,3) (1,5)
cout << a * -1 << " " << b / 2 << "\n"; // (-3,-4) (1,5,2)

// Operaciones generales:
Point translate(Point v, Point p) { return p + v; }
Point scale(Point c, double factor, Point p) { return c + (p
    - c) * factor; }
Point rotate(Point p, double a) { return {p.x * cos(a) - p.y
    * sin(a), p.x * sin(a) + p.y * cos(a)}; }
Point perpendicular(Point p) { return {-p.y, p.x}; }
double dist(Point p1, Point p2) { return hypot(p1.x - p2.x,
    p1.y - p2.y); }

// Operaciones vectoriales, en donde nuestro punto indica el
    fin del vector, siendo el origen su inicio
T dot(Point v, Point w) { return v.x * w.x + v.y * w.y; }
bool isPerp(Point v, Point w) { return dot(v, w) == 0; }

// Para c++17
double angle(Point v, Point w) { return acos(clamp(dot(v, w)
    / abs(v) / abs(w), -1.0, 1.0)); }
// C++14 o menor
double angle(Point v, Point w) {
    double cosTheta = dot(v, w) / abs(v) / abs(w);
    return acos(max(-1.0, min(1.0, cosTheta)));
}

T cross(Point v, Point w) { return v.x * w.y - v.y * w.x; }
T orient(Point a, Point b, Point c) { return cross(b - a, c
    - a); }

// Funcion signum: -1 si x es negativo, 0 si x = 0 y 1 si x
    es positivo
```

```
template <typename T>
int sgn(T x) {
    return (T(0) < x) - (x < T(0));
}

int manhattan(Point& p1, Point& p2) { return abs(p1.x - p2.x
    ) + abs(p1.y - p2.y); }

// Vector desplazamiento desde el punto p1 a p2
Point toVector(Point& p1, Point& p2) { return p2 - p1; }
bool areCollinear(Point& p, Point& q, Point& r) {
    return abs(cross(toVector(p, q), toVector(p, r))) <= EPS;
}
```

7.2 Linea

```
struct Line {
    double a, b, c;
    bool operator<(Line& other) const {
        if (fabs(a - other.a) >= EPS)
            return a < other.a;
        if (fabs(b - other.b) >= EPS)
            return b < other.b;
        return c < other.c;
    }
};

Line pointsToLine(Point& p1, Point& p2) {
    if (abs(p1.x - p2.x) <= EPS)
        return Line{1.0, 0.0, -p1.x};
    double a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    return Line{a, 1.0, -(double)(a * p1.x - p1.y)};
}

Line pointSlopeToLine(Point& p, double& m) { return Line{-m,
    1, -((-m * p.x) + p.y)}; }

bool areParallel(Line& l1, Line& l2) { return (abs(l1.a - l2
    .a) <= EPS && (abs(l1.b - l2.b) <= EPS); }

bool areSame(Line& l1, Line& l2) { return areParallel(l1, l2
    ) && (abs(l1.c - l2.c) <= EPS); }

bool areIntersect(Line l1, Line l2, Point& p) {
    if (areParallel(l1, l2)) return false;
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a *
        l2.b);
    if (fabs(l1.b) > EPS)
        p.y = -(l1.a * p.x + l1.c);
    else
        p.y = -(l2.a * p.x + l2.c);
    return true;
}

// convert point and gradient/slope to Line
void pointSlopeToLine(Point p, double m, Line& l) {
    l.a = -m;
    l.b = 1;
    l.c = -((l.a * p.x) + (l.b * p.y));
}

void closestPoint(Line l, Point p, Point& ans) {
    Line perpendicular;
    if (fabs(l.b) < EPS) { // vertical Line
        ans.x = -(l.c);
        ans.y = p.y;
        return;
    }
    if (fabs(l.a) < EPS) { // horizontal Line
        ans.x = p.x;
        ans.y = -(l.c);
        return;
    }
    pointSlopeToLine(p, 1 / l.a, perpendicular); // normal
    Line
    areIntersect(l, perpendicular, ans);
}
```

7.3 Poligono

```
const double EPS = 1e-9;
double DEG_to_RAD(double d) { return d * M_PI / 180.0; }
double RAD_to_DEG(double r) { return r * 180.0 / M_PI; }

// Duplicar P[0] al final del vector de puntos
double perimeter(const vector<Point>& P) {
    double ans = 0.0;
    for (int i = 0; i < (int)P.size() - 1; ++i)
        ans += dist(P[i], P[i + 1]);
    return ans;
}

// Formula de Heron
double triangleArea(Point& p1, Point& p2, Point& p3) {
    double a = abs(p2 - p1), b = abs(p3 - p1), c = abs(p3 - p2
        ), s = (a + b + c) / 2.0;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}

// Con la magnitud del producto cruz
double triangleArea(Point& p1, Point& p2, Point& p3) {
    return cross(p2 - p1, p3 - p1) / 2;
}

double area(const vector<Point>& P) {
    double ans = 0.0;
    for (int i = 0; i < (int)P.size() - 1; ++i)
        ans += (P[i].x * P[i + 1].y - P[i + 1].x * P[i].y);
    return fabs(ans) / 2.0;
}

bool isConvex(const vector<Point>& P) {
    int n = (int)P.size();
    if (n <= 3) return false;
    bool firstTurn = ccw(P[0], P[1], P[2]);
    for (int i = 1; i < n - 1; ++i)
        if (ccw(P[i], P[i + 1], P[(i + 2) % n]) != firstTurn)
            return false;
    return true;
}

// Retorna 1/0/-1 si el punto p esta dentro/sobre/fuera de
// cualquier poligono P concavo/convexo
int insidePolygon(Point pt, const vector<Point>& P) {
    int n = (int)P.size();
    if (n <= 3) return -1;
    bool on_polygon = false;
    for (int i = 0; i < n - 1; ++i)
        if (fabs(dist(P[i], pt) + dist(pt, P[i + 1]) - dist(P[i]
            , P[i + 1])) < EPS)
            on_polygon = true;
    if (on_polygon) return 0;
    double sum = 0.0;
    for (int i = 0; i < n - 1; ++i) {
        if (ccw(pt, P[i], P[i + 1]))
            sum += angle(P[i], pt, P[i + 1]);
        else
            sum -= angle(P[i], pt, P[i + 1]);
    }
    return fabs(sum) > M_PI ? 1 : -1;
}
```

7.4 Fracciones

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

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

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

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

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

    Frac operator*(int& x) const { return Frac(a * x, b); }
    Frac operator/(int& x) const {
        Frac o(1, x);
        return (*this) * o;
    }

    bool operator<(Frac& other) const { // PROVISIONAL,
        IMPLEMENTARLA MEJOR SI HACEN FALTA LOWER BOUNDS
        if (a != other.a)
            return a < other.a;
        return b < other.b;
    }

    bool operator==(Frac& other) const {
        return a == other.a && b == other.b;
    }

    bool operator!=(Frac& other) const {
        return !(*this == other);
    }
};
```

7.5 Convex Hull

```
/**
 * Descripcion: encuentra la envolvente convexa de un
 * conjunto
 * de puntos dados. Una envolvente convexa es la minima
 * region
 * convexa que contiene a todos los puntos del conjunto.
 * Tiempo: O(n log n)
 */
int orientation(Point a, Point b, Point c) {
    double v = a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}
```

```
}

bool cw(Point a, Point b, Point c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}

bool ccw(Point a, Point b, Point c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

vector<Point> convex_hull(vector<Point>& a, bool
    include_collinear = false) {
    if (a.size() == 1)
        return a;

    sort(ALL(a));
    Point p1 = a[0], p2 = a.back();
    vector<Point> up, down;
    up.push_back(p1);
    down.push_back(p2);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2,
            include_collinear)) {
            while (up.size() >= 2 && !cw(up[up.size() - 2], up[up.size() - 1], a[i], include_collinear))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2,
            include_collinear)) {
            while (down.size() >= 2 && !ccw(down[down.size() - 2], down[down.size() - 1], a[i], include_collinear))
                down.pop_back();
            down.push_back(a[i]);
        }
    }

    if (include_collinear && up.size() == a.size()) {
        reverse(a.begin(), a.end());
        return a;
    }

    vector<Point> ans;
    for (int i = 0; i < (int)up.size(); i++)
        ans.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)
        ans.push_back(down[i]);

    return ans;
}
```

7.6 Punto 3D

```
struct Point {
    double x, y, z;
    Point() {}
    Point(double xx, double yy, double zz) { x = xx, y = yy, z = zz; }

    /// scalar operators
    Point operator*(double f) { return Point(x * f, y * f, z * f); }
    Point operator/(double f) { return Point(x / f, y / f, z / f); }

    /// p3 operators
    Point operator-(Point p) { return Point(x - p.x, y - p.y, z - p.z); }
    Point operator+(Point p) { return Point(x + p.x, y + p.y, z + p.z); }
    Point operator%(Point p) { return Point(y * p.z - z * p.y, z * p.x - x * p.z, x * p.y - y * p.x); } // (|p| * q sin(ang)) * normal
    double operator|(Point p) { return x * p.x + y * p.y + z * p.z; }

    /// Comparators
```

```
bool operator==(Point p) { return tie(x, y, z) == tie(p.x, p.y, p.z); }
bool operator!=(Point p) { return !operator==(p); }
bool operator<(Point p) { return tie(x, y, z) < tie(p.x, p.y, p.z); }

Point zero = Point(0, 0, 0);

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

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

/// 3D - ORIENT
double orient(Point p, Point q, Point r, Point s) { return (q - p) % (r - p) | (s - p); }
bool coplanar(Point p, Point q, Point r, Point s) {
    return abs(orient(p, q, r, s)) < eps;
}
bool skew(Point p, Point q, Point r, Point s) { // skew :=
    // neither intersecting/parallel
    return abs(orient(p, q, r, s)) > eps; // lines:
    // PQ, RS
}
double orient_norm(Point p, Point q, Point r, Point n) { //
    // n := normal to a given plane PI
    return (q - p) % (r - p) | n; //
    // equivalent to 2D cross on PI (of ortogonal proj)
}
```


8 Extras

8.1 Fechas

```
/**
 * Descripcion: rutinas para realizar calculos sobre fechas,
 * en estas rutinas, los meses son expresados como enteros
 * desde
 * el 1 al 12, los dias como enteros desde el 1 al 31, y los
 * anios
 * como enteros de 4 digitos.
 */
string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// Convierte fecha Gregoriana a entero (fecha Juliana)
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;
}

// Convierte entero (fecha Juliana) a Gregoriana: M/D/Y
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;
}

// Convierte entero (fecha Juliana) a dia de la semana
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);

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

8.2 HashPair

```
/**
 * Descripcion: funciones hash utiles, ya que std::
 * unordered_map
 * no las provee nativamente, es recomendable usar la
 * segunda
 * cuando se trate de un pair<int, int>
 */
struct hash_pair {
    template <class T1, class T2>
    size_t operator()(const pair<T1, T2>& p) const {
        auto hash1 = hash<T1>{}(p.first);
```

```
        auto hash2 = hash<T2>{}(p.second);

        if (hash1 != hash2) {
            return hash1 ^ hash2;
        }
        return hash1;
    }
};
unordered_map<pair<int, int>, bool, hash_pair> um;
struct HASH {
    size_t operator()(const pair<int, int>& x) const {
        return (size_t)x.first * 37U + (size_t)x.second;
    }
};
unordered_map<pair<int, int>, int, HASH> xy;

// 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));

// Retorna un iterador apuntando al menor elemento en el
// rango dado (cambiar a max si se desea el mayor), es
// posible pasarle un comparador.
min_element(begin(x), end(x));
```

8.3 Trucos

```
// Descripcion: algunas funciones/atajos utiles para c++

// 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 hexadecimal
cout << hex << 100 << " " << 1000 << " " << 10000 << dec <<
    endl;

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

// piso(a / b)
cout << a / b;

// techo(a / b)
cout << (a + b - 1) / b;

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

// 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));
```