# Club de Algoritmos de Sinaloa Notebook

## Contents

**1 Templates** 3
1.1 Template C++ . . . 3
1.2 Template Max . . . 3
1.3 Template Python . . . 3

**2 DataStructure** 4
2.1 DSU Rollback . . . 4
2.2 Fenwick Tree . . . 4
2.3 Fenwick Tree 2D . . . 4
2.4 Fraction . . . 4
2.5 Line Container . . . 4
2.6 Matrix . . . 4
2.7 Mo Queries . . . 5
2.8 Order Statistics Tree . . . 5
2.9 Segment Tree . . . 5
2.10 Segment Tree 2D . . . 5
2.11 Segment Tree Lazy . . . 5
2.12 Segment Tree Persistent . . . 6
2.13 Segment Tree Sparse . . . 6
2.14 Sparse Table . . . 6
2.15 Sub Matrix . . . 6

**3 NumberTheory** 8
3.1 CRT . . . 8
3.2 Euclid . . . 8
3.3 Linear Diophantine . . . 8
3.4 Linear Sieve . . . 8
3.5 Mobius . . . 8
3.6 Mod Inverse . . . 8
3.7 Mod Multiplication . . . 8
3.8 Mod Pow . . . 8
3.9 Sieve . . . 8

**4 Combinatorial** 9
4.1 Catalan . . . 9
4.2 Combinations . . . 9

**5 Numerical** 10
5.1 Determinant . . . 10
5.2 FFT . . . 10
5.3 Gauss . . . 10
5.4 Simplex . . . 11
5.5 Simpson . . . 11

**6 DP** 12
6.1 Deque Tecnique . . . 12
6.2 Digit . . . 12
6.3 Knapsack . . . 12
6.4 LIS . . . 12
6.5 Monotonic Stack . . . 12
6.6 TSP . . . 12

**7 Graphs** 14
7.1 2SAT . . . 14
7.2 Bridges And Articulation Points . . . 14
7.3 Maximal Cliques . . . 14
7.4 Maximum Clique . . . 14
7.5 SCC . . . 15
7.6 General Matching . . . 15
7.7 Hopcroft Karp . . . 15
7.8 Hungarian . . . 16
7.9 Kuhn . . . 16
7.10 Minimum Vertex Cover . . . 16
7.11 Kruskal . . . 16
7.12 Prim . . . 16
7.13 Hierholzer . . . 16
7.14 Topological Sort . . . 17
7.15 Dinic . . . 17
7.16 Johnson . . . 17
7.17 Min Cost Max Flow . . . 17
7.18 Push Relabel . . . 18
7.19 Bellman-Ford . . . 18
7.20 Dijkstra . . . 18
7.21 Floyd-Warshall . . . 19
7.22 Binary Lifting LCA . . . 19
7.23 Centroid Decomposition . . . 19
7.24 Euler Tour . . . 19
7.25 HLD . . . 19

**8 Strings** 21
8.1 Aho Corasick . . . 21
8.2 Dynamic Aho Corasick . . . 21
8.3 Evil Hashing . . . 22
8.4 Hashing . . . 22
8.5 Hashing Dynamic . . . 22

8.6 KMP . . . 23
8.7 KMPAutomaton . . . 23
8.8 Manacher . . . 23
8.9 Palindromic Tree . . . 23
8.10 Suffix Array . . . 24
8.11 Suffix Automaton . . . 24
8.12 Suffix Tree . . . 24
8.13 Trie . . . 25
8.14 ZAlgorithm . . . 25

**9 Geometry** 26
9.1 Circle . . . 26
9.2 Closest Pair . . . 26
9.3 Convex Hull . . . 26
9.4 Half Plane . . . 26
9.5 Line . . . 27
9.6 Point . . . 27
9.7 Point3D . . . 27
9.8 Polar Sort . . . 28
9.9 Polygon . . . 28
9.10 Segment . . . 28

**10 Extras** 30
10.1 Bits . . . 30
10.2 Busquedas . . . 30
10.3 Dates . . . 30
10.4 Hash Pair . . . 30
10.5 Random . . . 30
10.6 Trucos . . . 30
10.7 int128 . . . 31

# 1 Templates

## 1.1 Template C++

```cpp
#include <bits/stdc++.h>
using namespace std;
// Pura Gente del Coach Moy
using ll = long long;
using pi = pair<int, int>;
using vi = vector<int>;

#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 = (int)a; i < (int)b; ++i)
#define ROF(i, a, b) \
  for (int i = (int)a - 1; i >= (int)b; --i)
#define ENDL '\n'

signed main() {
  cin.tie(0)->sync_with_stdio(0);

  return 0;
}
```

## 1.2 Template Max

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

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

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

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

// pairs
#define mp make_pair
#define 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 F0R(i, a) FOR(i, 0, a)
#define ROF(i, a, b) for (int i = (a)-1; i >= (b); --i)
#define R0F(i, a) ROF(i, a, 0)
#define ENDL '\n'
#define LSOne(S) ((S) & -(S))
#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;
}
```

## 1.3 Template Python

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

sys.setrecursionlimit(100000000)


def inp():
    return int(input())


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


def jn(x, l):
    return x.join(map(str, l))


def strl():
    return list(input().strip())


def mul():
    return map(int, input().strip().split())


def mulf():
    return map(float, input().strip().split())


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


def ceil(x):
    return int(x) if (x == int(x)) else int(x) + 1


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


def flush():
    return stdout.flush()


def stdstr():
    return stdin.readline()


def stdint():
    return int(stdin.readline())


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


mod = 1000000007
```

# 2 DataStructure

## 2.1 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)
 * Status: testeadisimo
 */

struct RollbackDSU {
  vector<int> e;
  vector<pi> st;
  void init(int n) { e = vi(n, -1); }
  int size(int x) { return -e[get(x)]; }
  int get(int x) {
    return e[x] < 0 ? x : e[x] = get(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 = get(a), b = get(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;
  }
};
```

## 2.2 Fenwick Tree

```
/**
 * Descripcion: arbol binario indexado, util para
 * consultas en donde es posible hacer
 * inclusion-exclusion, suma, multiplicacion, etc.
 * Tiempo:
 * O(log n)
 */

struct FT {
  vector<ll> s;
  FT(int n) : s(n) {}
  void update(int pos, ll dif) {  // a[pos] += dif
    for (; pos < SZ(s); pos |= pos + 1) s[pos] += dif;
  }
  ll query(int pos) {  // sum of values in [0, pos)
    ll res = 0;
    for (; pos > 0; pos &= pos - 1) res += s[pos - 1];
    return res;
  }
  int lower_bound(
      ll sum) {  // min pos st sum of [0, pos] >= sum
    // Returns n if no sum is >= sum, or -1 if empty sum
    // is.
    if (sum <= 0) return -1;
    int pos = 0;
    for (int pw = 1 << 25; pw; pw >>= 1)
      if (pos + pw <= SZ(s) && s[pos + pw - 1] < sum)
        pos += pw, sum -= s[pos - 1];
    return pos;
  }
};
```

## 2.3 Fenwick Tree 2D

```
/**
 * Descripcion: arbol binario indexado 2D, util para
 * consultas en un espacio 2D como una matriz
 * Tiempo:
 * Construir el BIT: O(NM log(N)*log(M))
 * Consultas y Actualizaciones: O(log(N)*log(M))
 */

int ft[MAX + 1][MAX + 1];
void upd(int i0, int j0, int v) {
  for (int i = i0 + 1; i <= MAX; i += i & -i)
    for (int j = j0 + 1; j <= MAX; j += j & -j)
      ft[i][j] += v;
}
int get(int i0, int j0) {
  int r = 0;
  for (int i = i0; i; i -= i & -i)
    for (int j = j0; j; j -= j & -j) r += ft[i][j];
  return r;
}
int get_sum(int i0, int j0, int i1, int j1) {
  return get(i1, j1) - get(i1, j0) - get(i0, j1) +
         get(i0, j0);
}
```

## 2.4 Fraction

```
/**
 * Descripcion: estructura para manejar fracciones.
 * Se asume que el denominador no es cero.
 * Tiempo por operacion: O(1)
 */
struct Frac {
  int num, den;

  Frac(int num, int den) {
    int g = gcd(num, den);
    this->num = num / g;
    this->den = den / g;
  }

  Frac operator*(Frac& f) const {
    return Frac(num * f.num, den * f.den);
  }
  Frac operator/(Frac& f) const {
    return (*this) * Frac(f.den, f.num);
  }
  Frac operator+(Frac& f) const {
    return Frac(num * f.den + den * f.num, den * f.den);
  }
  Frac operator-(Frac& f) const {
    return Frac(num * f.den - den * f.num, den * f.den);
  }
  bool operator<(Frac& other) const {
    return num * other.den < other.num * den;
  }
  bool operator==(Frac& other) const {
    return num == other.num && den == other.den;
  }
  bool operator!=(Frac& other) const {
    return !(*this == other);
  }
};
```

## 2.5 Line Container

```
/*
 * Line Container (Convex Hull Trick)
 * Descripcion: Contenedor donde puedes anadir lineas en
 * forma kx+m, y hacer consultas al valor maximo en un
 * punto x. Pro-Tip: Si se busca el valor minimo en un
 * punto x, anadir las lineas con pendiente K negativa (la
 * consulta se dara de forma negativa)
 * Tiempo: O(log n)
 */

struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
  // (para doubles, usar inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) {  // floored division
    return a / b - ((a ^ b) < 0 && a % b);
  }
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, 0;
    if (x->k == y->k)
      x->p = x->m > y->m ? inf : -inf;
    else
      x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y))
      isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

## 2.6 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;

  Matrix(VVT aux) : M(aux) {}

  Matrix operator*(Matrix& other) const {
    assert(SZ(M[0]) == SZ(other.M));
    int k = SZ(other.M[0]);
    VVT C(n, vector<T>(k, 0));
    FOR(i, 0, SZ(M))
```

```cpp
    FOR(j, 0, k) FOR(l, 0, SZ(M[0])) 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;
  }

  void print() {
    cout << "--------------------" << ENDL;
    FOR(i, 0, n) {
      FOR(j, 0, m) { cout << M[i][j] << ' '; }
      cout << ENDL;
    }
    cout << "******************" << ENDL;
  }
};
```

## 2.7 Mo Queries

```cpp
/*
 * Mos Algorithm
 * Descripcion: Es usado para responder consultas en
 * intervalos (L,R) de manera offline con base a un
 * orden especial basado en bloques moviendose de una
 * consulta a la siguiente anadiendo/removiendo puntos
 * en el inicio o el final. Rango de query inclusivo a
 * la izquierda y exclusivo a la derecha.
 *
 * Tiempo: O((N + Q) sqrt(N))
 */

void add(int idx, int end) {
}  // add a[idx] (end = 0 or 1)
void del(int idx, int end) {}  // remove a[idx]
int calc(){...}                // compute current answer

vi mosAlgo(vector<pi> Q) {
  // IMPORTANT!! blk ~= N/sqrt(Q)
  int L = 0, R = 0, blk = 350;
  vi s(SZ(Q)), res = s;
#define K(x) \
  pi(x.first / blk, x.second ^ -(x.first / blk & 1))
  iota(ALL(s), 0);
  sort(ALL(s),
       [&](int s, int t) { return K(Q[s]) < K(Q[t]); });
  for (int qi : s) {
    pi q = Q[qi];
    while (L > q.first) add(--L, 0);
    while (R < q.second) add(R++, 1);
    while (L < q.first) del(L++, 0);
    while (R > q.second) del(--R, 1);
    res[qi] = calc();
  }
  return res;
}
```

## 2.8 Order Statistics Tree

```cpp
/**
 * 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: numero de elementos estrictamente
 * menores a x
 * 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>;
```

## 2.9 Segment Tree

```cpp
/**
 * Descripcion: segment tree para suma en rango y
 * actualizacion en punto. Rangos no inclusivos a la der.
 * [a,b) es el rango de la query y [s, e) el del nodo
 *
 * Uso: STree st;st.init(arr);st.upd(0, 3);st.query(0, n);
 * Tiempo de construccion: O(n)
 * Tiempo de consulta y actualizacion: O(log n)
 * Memoria total: O(n + qlog(n))
 * Status: testeado en CSES Hotel Queries
 */
#define NEUT 0
struct STree {
  int n;
  vi st;
  STree(int n) : st(4 * n + 5, NEUT), n(n) {}
  int comb(int x, int y) { return x + y; }
  void init(int k, int s, int e, vi& a) {
    if (s + 1 == e) {
      st[k] = a[s];
      return;
    }
    int m = (s + e) / 2;
    init(2 * k + 1, s, m, a);
    init(2 * k + 2, m, e, a);
    st[k] = comb(st[2 * k + 1], st[2 * k + 2]);
  }
  int query(int k, int s, int e, int a, int b) {
    if (a <= s && e <= b) return st[k];
    if (e <= a || s >= b) return NEUT;
    int m = (s + e) / 2;
    return comb(query(2 * k + 1, s, m, a, b),
                query(2 * k + 2, m, e, a, b));
  }
  void upd(int k, int s, int e, int i, int v) {
    if (e <= i || s > i) return;
    if (s + 1 == e) {
      st[k] = v;
      return;
    }
    int m = (s + e) / 2;
    upd(2 * k + 1, s, m, i, v);
    upd(2 * k + 2, m, e, i, v);
    st[k] = comb(st[2 * k + 1], st[2 * k + 2]);
  }
  int query(int a, int b) { return query(0, 0, n, a, b); }
  void upd(int i, int v) { upd(0, 0, n, i, v); }
  void init(vi& a) { init(0, 0, n, a); }
};
```

## 2.10 Segment Tree 2D

```cpp
/**
 * Descripcion: segment tree 2D para suma en un
 * rectangulo y actualizacion en punto.
 * Implementacion iterativa para usar x4 menos memoria.
 *
 * Uso: STree st(n,m);st.build(a);st.query(x0,y0,x1,y1);
 * (x0, y0) es la esquina superior izquierda (inclusiva)
 * y (x1, y1) la esquina inferior derecha (exclusiva)
 * Tiempo de construccion: O(nm)
 * Tiempo de consulta y actualizacion: O(log n log m)
 * Memoria total: O(nm)
 * Status: testeado en CSES Forest Queries II
 */
#define NEUT 0
struct STree {
  int n, m;
  vector<vi> st;
  STree(int n, int m)
      : st(2 * n, vi(2 * m, NEUT)), n(n), m(m) {}
  int comb(int x, int y) { return x + y; }
  void build(vector<vi>& a) {
    FOR(i, 0, n) FOR(j, 0, m) st[i + n][j + m] = a[i][j];
    FOR(i, 0, n)
    for (int j = m - 1; j; --j) st[i + n][j] =
        comb(st[i + n][j << 1], st[i + n][j << 1 | 1]);
    for (int i = n - 1; i; --i) FOR(j, 0, 2 * m)
    st[i][j] = comb(st[i << 1][j], st[i << 1 | 1][j]);
  }
  void upd(int x, int y, int v) {
    st[x + n][y + m] = v;
    for (int j = y + m; j > 1; j >>= 1)
      st[x + n][j >> 1] =
          comb(st[x + n][j], st[x + n][j ^ 1]);
    for (int i = x + n; i > 1; i >>= 1)
      for (int j = y + m; j; j >>= 1)
        st[i >> 1][j] = comb(st[i][j], st[i ^ 1][j]);
  }
  int query(int x0, int y0, int x1, int y1) {
    int r = 0;
    for (int i0 = x0 + n, i1 = x1 + n; i0 < i1;
         i0 >>= 1, i1 >>= 1) {
      int t[4], q = 0;
      if (i0 & 1) t[q++] = i0++;
      if (i1 & 1) t[q++] = --i1;
      FOR(k, 0, q)
      for (int j0 = y0 + m, j1 = y1 + m; j0 < j1;
           j0 >>= 1, j1 >>= 1) {
        if (j0 & 1) r = comb(r, st[t[k]][j0++]);
        if (j1 & 1) r = comb(r, st[t[k]][--j1]);
      }
    }
    return r;
  }
};
```

## 2.11 Segment Tree Lazy

```cpp
/**
 * Descripcion: lazy segment tree para suma y
 * actualizacion en rango. Rangos no inclusivos a la der.
 * [a,b) es el rango de la query y [s, e) el del nodo
 * Si los incrementos pueden ser negativos o cero,
 * utilizar arreglo extra has_lazy, o buscar otro neutral
 *
 * Uso: STree st;st.init(arr);st.upd(0, n, 3);st.query(0,
 * n); Tiempo de construccion: O(n) Tiempo de consulta y
 * actualizacion: O(log n) Status: testeado en CSES
 * Dynamic Range Sum Queries
 */
#define LAZY_NEUT 0
```

```
#define VAL_NEUT 0
struct STree {
  int n;
  vector<int> st, lazy;
  STree(int n)
      : st(4 * n + 5, VAL_NEUT),
        lazy(4 * n + 5, LAZY_NEUT),
        n(n) {}
  int comb(int x, int y) { return x + y; }
  void init(int k, int s, int e, vi& a) {
    if (s + 1 == e) {
      st[k] = a[s];
      return;
    }
    int m = (s + e) / 2;
    init(2 * k + 1, s, m, a);
    init(2 * k + 2, m, e, a);
    st[k] = comb(st[2 * k + 1], st[2 * k + 2]);
  }
  void push(int k, int s, int e) {
    if (lazy[k] == LAZY_NEUT) return;
    st[k] += (e - s) * lazy[k];
    if (s + 1 != e)
      lazy[2 * k + 1] += lazy[k],
          lazy[2 * k + 2] += lazy[k];
    lazy[k] = LAZY_NEUT;
  }
  int query(int k, int s, int e, int a, int b) {
    push(k, s, e);
    if (a <= s && e <= b) return st[k];
    if (e <= a || s >= b) return VAL_NEUT;
    int m = (s + e) / 2;
    return comb(query(2 * k + 1, s, m, a, b),
                query(2 * k + 2, m, e, a, b));
  }
  void upd(int k, int s, int e, int a, int b, int v) {
    push(k, s, e);
    if (e <= a || s >= b) return;
    if (a <= s && e <= b) {
      lazy[k] += v;
      push(k, s, e);
      return;
    }
    int m = (s + e) / 2;
    upd(2 * k + 1, s, m, a, b, v);
    upd(2 * k + 2, m, e, a, b, v);
    st[k] = comb(st[2 * k + 1], st[2 * k + 2]);
  }
  int query(int a, int b) { return query(0, 0, n, a, b); }
  void upd(int a, int b, int v) { upd(0, 0, n, a, b, v); }
  void init(vi& a) { init(0, 0, n, a); }
};
```

## 2.12 Segment Tree Persistent

```
/**
 * Descripcion: segment tree persistente para consulta de
 * minimos en un rango. Un segment tree es persistente si
 * mantiene datos viejos disponible tras actualizaciones.
 * init y upd retornan la nueva raiz, rt es la ultima raiz
 * Para mejorar velocidad, usar arreglos.
 *
 * Uso: STree st;st.init(arr);
 * vi roots;roots.pb(st.upd(0,3));st.query(roots[1], 0,
 * n);
 * Tiempo: log(n) Status: testeado en OmegaUp Campo
 * Inestable
 */

#define NEUT 0
struct STree {
  vi st, L, R;
  int n, sz, rt;
  STree(int n)
      : st(1, NEUT),
        L(1, 0),
        R(1, 0),
        n(n),
        rt(0),
        sz(1) {}
  int comb(int x, int y) { return min(x, y); }
  int new_node(int v, int l = 0, int r = 0) {
    int ks = SZ(st);
    st.pb(v);
    L.pb(l);
    R.pb(r);
    return ks;
  }
  int init(int s, int e, vi &a) {
    if (s + 1 == e) return new_node(a[s]);
    int m = (s + e) / 2, l = init(s, m, a),
        r = init(m, e, a);
    return new_node(comb(st[l], st[r]), l, r);
  }
  int upd(int k, int s, int e, int p, int v) {
    int ks = new_node(st[k], L[k], R[k]);
    if (s + 1 == e) {
      st[ks] = v;
      return ks;
    }
    int m = (s + e) / 2, ps;
    if (p < m)
      ps = upd(L[ks], s, m, p, v), L[ks] = ps;
    else
      ps = upd(R[ks], m, e, p, v), R[ks] = ps;
    st[ks] = comb(st[L[ks]], st[R[ks]]);
    return ks;
  }
  int query(int k, int s, int e, int a, int b) {
    if (e <= a || b <= s) return NEUT;
    if (a <= s && e <= b) return st[k];
    int m = (s + e) / 2;
    return comb(query(L[k], s, m, a, b),
                query(R[k], m, e, a, b));
  }
  int init(vi &a) { return init(0, n, a); }
  int upd(int k, int p, int v) {
    return rt = upd(k, 0, n, p, v);
  }
  int upd(int p, int v) { return rt = upd(rt, p, v); }
  int query(int k, int a, int b) {
    return query(k, 0, n, a, b);
  };
};
```

## 2.13 Segment Tree Sparse

```
/**
 * Descripcion: segment tree esparcido. Para suma en
 * rango y actualizacion en punto.
 * Un segment tree es esparcido cuando sus nodos se van
 * creando conforme se vayan utilizando, lo que permite
 * manejar grandes rangos de indices. Es util cuando en
 * esta situacion necesitas un algoritmo online (no se
 * puede realizar compresion de coordenadas).
 * Rangos no inclusivos a la der.
 * Para mejorar velocidad, usar arreglos.
 * [a,b) es el rango de la query y [s, e) el del nodo
 *
 * Uso: STree st(1e18);st.upd(6, 3);st.query(5, 1e12);
 * Tiempo de consulta y actualizacion: O(log n)
 * Memoria total: O(qlog(n))
 * Status: testeado en CSES Salary Queries
 */
#define NEUT 0
struct STree {
  int n;
  vi st, L, R;
  STree(int n) : st(1, NEUT), L(1, -1), R(1, -1), n(n) {}
  int comb(int x, int y) { return x + y; }
  int new_node() {
    st.pb(NEUT);
    L.pb(-1);
    R.pb(-1);
    return SZ(st) - 1;
  }
  int query(int k, int s, int e, int a, int b) {
    if (k == -1 || e <= a || s >= b) return NEUT;
    if (a <= s && e <= b) return st[k];
    int m = s + (e - s) / 2;
    return comb(query(L[k], s, m, a, b),
                query(R[k], m, e, a, b));
  }
  int upd(int k, int s, int e, int i, int v) {
    if (e <= i || s > i) return k;
    if (k == -1) {
      k = new_node();
    }
    if (s + 1 == e) {
      st[k] = v;
      return k;
    }
    int m = s + (e - s) / 2,
        le = L[k] == -1 ? 0 : st[L[k]],
        ri = R[k] == -1 ? 0 : st[R[k]];
    if (i < m)
      L[k] = upd(L[k], s, m, i, v), le = st[L[k]];
    else
      R[k] = upd(R[k], m, e, i, v), ri = st[R[k]];
    st[k] = comb(le, ri);
    return k;
  }
  int query(int a, int b) { return query(0, 0, n, a, b); }
  void upd(int i, int v) { upd(0, 0, n, i, v); }
};
```

## 2.14 Sparse Table

```
/**
 * Descripcion: util para consultas min/max en rango para
 * arreglos inmutables, 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 <class T>
struct SparseTable {
  vector<vector<T>> jmp;
  void init(const vector<T>& V) {
    if (SZ(jmp)) jmp.clear();
    jmp.emplace_back(V);
    for (int pw = 1, k = 1; pw * 2 <= SZ(V);
         pw *= 2, ++k) {
      jmp.emplace_back(SZ(V) - pw * 2 + 1);
      FOR(j, 0, SZ(jmp[k]))
      jmp[k][j] = min(jmp[k - 1][j], jmp[k - 1][j + pw]);
    }
  }
  T query(int l, int r) {  // [a, b)
    int dep = 31 - __builtin_clz(r - l);
    return min(jmp[dep][l], jmp[dep][r - (1 << dep)]);
  }
};
```

## 2.15 Sub Matrix

```
/*
 * Descripcion: Calcula la suma de una submatriz
 * rapidamente, dada la esquina superior izquierda e
 * inferior derecha
 * Uso: SubMatrix<int> m(matriz);
```

```
 * m.sum(0,0,2,2); // 4 elemento arriba a la izquierda
 *
 * Tiempo: O(N^2)
 */

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];
  }
};
```

# 3 NumberTheory

## 3.1 CRT

```
/*
 * Teorema del Resto Chino (CRT)
 * Descripcion: Encuentra x dentro de un sistema lineal
 * de congruencias tal que:
 * x =_ a (mod n)
 * x =_ b (mod m)
 * Se asume que n*m < 2^62
 *
 * Tiempo: O(log n)
 */

ll crt(ll a, ll m, ll b, ll n) {
  if (n > m) swap(a, b), swap(m, n);
  ll x, y, g = euclid(m, n, x, y);
  assert((a - b) % g == 0);  // else no solution
  x = (b - a) % n * x % n / g * m + a;
  return x < 0 ? x + m * n / g : x;
}
```

## 3.2 Euclid

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

## 3.3 Linear Diophantine

```
/**
 * Problema: Dado a, b y n. Encuentra x,y que
 * satisfagan la ecuacion ax + by = n.
 * Encuentra una de las soluciones, otras validas
 * pueden obtenerse por medio de:
 * x = x0 + k*(b/g), y = y0 - k*(a/g)
 *
 * Tiempo: O(log n)
 */
pi linear_diophantine(int a, int b, int n) {
  int x0, y0, g = euclid(a, b, x0, y0);
  if (n % g) return {-1, -1};  // no solution exists
  x0 *= n / g, y0 *= n / g;
  return {x0, y0};
}
```

## 3.4 Linear Sieve

```
/**
 * Descripcion: algoritmo para precalcular los
 * numeros primos menor o iguales a N.
 * Sirve como guia para calcular funciones
 * multiplicativas
 *
 * Tiempo: O(n)
 */
void sieve(int N) {
  vi lp(N + 1), 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;
    }
  }
}
```

## 3.5 Mobius

```
/**
 * Descripcion: Calcula la funcion de Mobius
 * para todo entero menor o igual a n
 *
 * Tiempo: O(N)
 */
void mobius(int N) {
  vi mu(N), check(N);
  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];
    }
  }
}
```

## 3.6 Mod Inverse

```
/**
 * Descripcion: Precalculo de modulos
 * inversos para toda x <= MAX.
 * Se asume que LIM <= m y que m es primo
 *
 * Tiempo: O(MAX)
 */
ll inv[MAX];
void precalc_inverse(int m) {
  inv[1] = 1;
  FOR(i, 2, MAX) {
    inv[i] = m - (m / i) * inv[m % i] % m;
  }
}
/**
 * Descripcion: Precalculo de un solo
 * inverso, usa el primer metodo si m
 * es primo, y el segundo en otro caso.
 *
 * Tiempo: O(log m)
 */
#include "Euclid.h"
ll inverse(ll b) { return be(b, MOD - 2) % MOD; }
ll inverse(ll a) {
  ll x, y, d = euclid(a, MOD, x, y);
  assert(d == 1);
  return (x + MOD) % MOD;
}
```

## 3.7 Mod Multiplication

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

## 3.8 Mod Pow

```
/**
 * Descripcion: Calcula a^b mod m
 *
 * Tiempo: O(log b)
 */
ll be(ll a, ll b, ll m) {
  ll res = 1;
  a %= m;
  while (b) {
    if (b & 1) res = res * a % m;
    a = a * a % m;
    b >>= 1;
  }
  return res;
}
```

## 3.9 Sieve

```
/**
 * Descripcion: algoritmo para precalcular los
 * numeros primos menor o iguales a N.
 *
 * Tiempo: O(n log(log n))
 */
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]) continue;
    for (ll i = p * p; i <= n; i += p) is_prime[i] = 0;
    primes.push_back(p);
  }
}
```

# 4 Combinatorial

## 4.1 Catalan

```
catalan = [0 for i in range(150 + 5)]


def fcatalan(n):
    catalan[0] = 1
    catalan[1] = 1
    for i in range(2, n + 1):
        catalan[i] = 0
        for j in range(i):
            catalan[i] = catalan[i] + catalan[j] * catalan[i
                   - j - 1]


fcatalan(151)
```

## 4.2 Combinations

```
/**
 * 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 < MAX, se utiliza el hecho de que comb(n, k) = (n!) /
 * (k! * (n - k)!)
 * Tiempo: O(MAX) en el precalculo de
 * inversos modulares y O(1) por query.
 */

#include "ModInverse.h";

ll invfact[maxn];
void precalc_invfact() {
  precalc_inv();
  invfact[1] = 1;
  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 MAX 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: precalculo de combinaciones
 *
 * Tiempo: O(n^2)
 */
ll c[maxn][maxk];
void calc_comb() {
  FOR(i, 0, MAX_N) {
    c[i][0] = c[i][i] = 1;
    FOR(j, 1, i) c[i][j] = c[i - 1][j] + c[i - 1][j - 1];
  }
}
```

# 5 Numerical

## 5.1 Determinant

```
/*
 * Descripcion: Calcula la determinante de la matriz.
 * Nota: La matriz se modifica
 *
 * Tiempo: O(n^3)
 */

double det(vector<vector<double>>& a) {
  int n = SZ(a);
  double res = 1;
  FOR(i, 0, n) {
    int b = i;
    FOR(j, i + 1, n)
    if (fabs(a[j][i]) > fabs(a[b][i])) b = j;
    if (i != b) swap(a[i], a[b]), res *= -1;
    res *= a[i][i];
    if (res == 0) return 0;
    FOR(j, i + 1, n) {
      double v = a[j][i] / a[i][i];
      if (v != 0) FOR(k, i + 1, n) a[j][k] -= v * a[i][k];
    }
  }
  return res;
}

// Determinante con modulo
// Si se elimina el modulo obtendras la version de solo
// enteros
ll detMod(vector<vector<ll>>& a, int mod) {
  int n = SZ(a);
  ll ans = 1;
  FOR(i, 0, n) {
    FOR(j, i + 1, n) {
      while (a[j][i] != 0) {
        ll t = a[i][i] / a[j][i];
        if (t) FOR(k, i, n)
        a[i][k] = (a[i][k] - a[j][k] * t) % mod;
        swap(a[i], a[j]);
        ans *= -1;
      }
    }
    ans = ans * a[i][i] % mod;
    if (!ans) return 0;
  }
  return (ans + mod) % mod;
}
```

## 5.2 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;
}
```

## 5.3 Gauss

```
/*
 * Descripcion: Dado un sistema de N ecuaciones lineales
 * con M incognitas, determinar si existe solucion,
 * infinitas soluciones o en caso de que halla al menos
 * una, encontrar cualquiera de ellas.
 * Tiempo: O(n^3)
 */

int gauss(vector<vector<double>> &a,
          vector<double> &ans) {
  int n = SZ(a), m = SZ(a[0]) - 1;
  vi where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; ++col) {
    int sel = row;
    FOR(i, row, n)
    if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
    if (abs(a[sel][col]) < EPS) continue;

    FOR(i, col, m + 1)
    swap(a[sel][i], a[row][i]);
    where[col] = row;

    FOR(i, 0, n) {
      if (i != row) {
        double c = a[i][col] / a[row][col];
        for (int j = col; j <= m; ++j)
          a[i][j] -= a[row][j] * c;
      }
    }
    ++row;
  }

  ans.assign(m, 0);
  FOR(i, 0, m) {
    if (where[i] != -1)
      ans[i] = a[where[i]][m] / a[where[i]][i];
  }
  FOR(i, 0, n) {
    double sum = 0;
    FOR(j, 0, m)
    sum += ans[j] * a[i][j];
    if (abs(sum - a[i][m]) > EPS) return 0;
  }

  FOR(i, 0, m)
  if (where[i] == -1)
    return 1e9;  /// infinitas soluciones
  return 1;
}

// Gauss con MOD
// Nota: es necesario la funcion modInverse
// Si MOD = 2, se convierte en operacion XOR y se puede
// utilizar un bitset para construir la ecuacion
// (disminuye la complejidad)
ll gaussMod(vector<vi> &a, vi &ans) {
  ll n = SZ(a), m = SZ(a[0]) - 1;
  vi where(m, -1);
  for (ll col = 0, row = 0; col < m && row < n; ++col) {
    ll sel = row;
    FOR(i, row, n)
    if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
    if (abs(a[sel][col]) <= EPS) continue;

    FOR(i, col, m + 1)
    swap(a[sel][i], a[row][i]);
    where[col] = row;

    FOR(i, 0, n) {
      if (i != row) {
        ll c = 1LL * a[i][col] * modInverse(a[row][col]) %
               MOD;
        for (ll j = col; j <= m; ++j)
          a[i][j] = (MOD + a[i][j] -
                     (1LL * a[row][j] * c) % MOD) %
                    MOD;
      }
    }
    ++row;
  }

  ans.assign(m, 0);
  FOR(i, 0, m) {
    if (where[i] != -1)
```

```
      ans[i] = 1LL * a[where[i]][m] *
               modInverse(a[where[i]][i]) % MOD;
  }
  FOR(i, 0, n) {
    ll sum = 0;
    FOR(j, 0, m)
    sum = (sum + 1LL * ans[j] * a[i][j]) % MOD;
    if (abs(sum - a[i][m]) > EPS) return 0;
  }

  FOR(i, 0, m)
  if (where[i] == -1)
    return 1e9;  /// infinitas soluciones
  return 1;
}
```

## 5.4 Simplex

```
/*
 * Descripcion: Es un algoritmo de programacion lineal
 * para resolver problemas de optimizacion. El objetivo es
 * maximizar una funcion lineal(c) sujeta a un conjunto de
 * restricciones lineales(b). Maximiza cTx sujeto a Ax <
 * b, x > 0
 * Uso: T val = LPSolver(A,b,c).solve(x);
 * Entrada: A -> MxN matriz de r
 *          b -> un vector de tamano M (restricciones)
 *          c -> un vector de tamano N (maximizar)
 *          x -> un vector donde la solucion optima es
 * almacenado Salida: el valor maximo de la solucion
 * optima. (Infinito en un caso ilimtado)
 * Tiempo:
 * O(NM*#pivotes) donde el #pivotes en el peor caso es
 * exponencial
 */

typedef double T;
typedef vector<T> vd;
typedef vector<vd> vvd;

const T eps = 1e-8, inf = 1 / .0;
#define MP make_pair
#define ltj(X) \
  if (s == -1 || MP(X[j], N[j]) < MP(X[s], N[s])) s = j

struct LPSolver {
  int m, n;
  vi N, B;
  vvd D;

  LPSolver(const vvd& A, const vd& b, const vd& c)
      : m(SZ(b)),
        n(SZ(c)),
        N(n + 1),
        B(m),
        D(m + 2, vd(n + 2)) {
    FOR(i, 0, m)
    FOR(j, 0, n)
    D[i][j] = A[i][j];
    FOR(i, 0, m) {
      B[i] = n + i;
      D[i][n] = -1;
      D[i][n + 1] = b[i];
    }
    FOR(j, 0, n) {
      N[j] = j;
      D[m][j] = -c[j];
    }
    N[n] = -1;
    D[m + 1][n] = 1;
  }

  void pivot(int r, int s) {
    T *a = D[r].data(), inv = 1 / a[s];
    FOR(i, 0, m + 2)
    if (i != r && abs(D[i][s]) > eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      FOR(j, 0, n + 2)
      b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    FOR(j, 0, n + 2)
    if (j != s) D[r][j] *= inv;
    FOR(i, 0, m + 2)
    if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool simplex(int phase) {
    int x = m + phase - 1;
    for (;;) {
      int s = -1;
      FOR(j, 0, n + 1)
      if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return true;
      int r = -1;
      FOR(i, 0, m) {
        if (D[i][s] <= eps) continue;
        if (r == -1 ||
            MP(D[i][n + 1] / D[i][s], B[i]) <
                MP(D[r][n + 1] / D[r][s], B[r]))
          r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  }

  T solve(vd& x) {
    int r = 0;
    FOR(i, 1, m)
    if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -eps) {
      pivot(r, n);
      if (!simplex(2) || D[m + 1][n + 1] < -eps)
        return -inf;
      FOR(i, 0, m)
      if (B[i] == -1) {
        int s = 0;
        FOR(j, 1, n + 1)
        ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1);
    x = vd(n);
    FOR(i, 0, m)
    if (B[i] < n) x[B[i]] = D[i][n + 1];
    return ok ? D[m][n + 1] : inf;
  }
};
```

## 5.5 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;
}
```

# 6 DP

## 6.1 Deque Tecnique

```
/**
 * 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()];
  }
}
```

## 6.2 Digit

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

## 6.3 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]));
}
```

## 6.4 LIS

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

## 6.5 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);
  }
}
```

## 6.6 TSP

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

# 7 Graphs

## 7.1 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 verdedero 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
 */

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

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

  int addVar() {
    g.emplace_back();
    g.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);
    g[x].push_back(y ^ 1), g[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 : g[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;
  }
};
```

## 7.2 Bridges And Articulation Points

```
/**
 * Descripcion: basado en la clasificacion de aristas por
 * medio del arbol de dfs, donde una arista es puente si
 * no existe un back-edge que cruce por encima, los puntos
 * de articulacion se obtienen de forma similar. Retorna
 * los puentes y un vector donde art[i] = 1 si el i-esimo
 * nodo es un punto de articulacion.
 * Tiempo: O(V + E)
 */

pair<vector<pi>, vi> findBridgesAndArticulationPoints(
    vector<vi>& g) {
  int n = SZ(g), t = 0;
  vector<pi> bridges;
  vi tin(n, -1), low(n, -1), art(n);
  auto dfs = [&](auto self, int u, int p = -1) -> void {
    tin[u] = low[u] = t++;
    int children = 0;
    for (int v : g[u])
      if (v != p) {
        if (tin[v] != -1) {
          low[u] = min(low[u], tin[v]);
          continue;
        }
        self(self, v, u);
        if (low[v] >= tin[u] && p != -1) art[u] = 1;
        if (low[v] > tin[u]) bridges.pb({u, v});
        low[u] = min(low[u], low[v]);
        children++;
      }
    if (p == -1 && children > 1) art[u] = 1;
  };
  FOR(u, 0, n) if (tin[u] == -1) dfs(dfs, u);
  return {bridges, art};
}
```

## 7.3 Maximal Cliques

```
/**
 * Descripcion: Corre un callback para todos los cliques
 * maximales en un grafo (dado como una matriz simetrica
 * de bitset; aristas propias no permitidas). El callback
 * recibe un bitset representando al clique maximal.
 *
 * Tiempo: O(3^{V/3}), mucho mas veloz para grafos
 * dispersos.
 */
/// Posible optimizacion: en el mayor nivel de recursion
/// ignora 'cands' y muevete hacia nodos en orden de grado
/// creciente, donde los grados bajan al eliminar
/// nodos (en su mayoria irrelevante dado MaximumClique)

typedef bitset<128> B;
template <class F>
void cliques(vector<B>& eds, F f, B P = ~B(), B X = {},
             B R = {}) {
  if (!P.any()) {
    if (!X.any()) f(R);
    return;
  }
  auto q = (P | X)._Find_first();
  auto cands = P & ~eds[q];
  rep(i, 0, sz(eds)) if (cands[i]) {
    R[i] = 1;
    cliques(eds, f, P & eds[i], X & eds[i], R);
    R[i] = P[i] = 0;
    X[i] = 1;
  }
}
```

## 7.4 Maximum Clique

```
/**
 * Descripcion: Rapidamente encuentra el clique maximo de
 * un grafo (dado como una matriz simetrica de bitset;
 * aristas propias no permitidas). Puede ser usado para
 * encontrar un conjunto independiente maximo, encontrando
 * un clique del grafo complemento.
 *
 * Tiempo: aproximadamente 1s para n=155 y el peor caso
 * para grafos aleatorios (p=.90), mas veloz para grafos
 * dispersos.
 */
typedef vector<bitset<200>> vb;
struct Maxclique {
  double limit = 0.025, pk = 0;
  struct Vertex {
    int i, d = 0;
  };
  typedef vector<Vertex> vv;
  vb e;
  vv V;
  vector<vi> C;
  vi qmax, q, S, old;
  void init(vv& r) {
    for (auto& v : r) v.d = 0;
    for (auto& v : r)
      for (auto j : r) v.d += e[v.i][j.i];
    sort(ALL(r),
         [](auto a, auto b) { return a.d > b.d; });
    int mxD = r[0].d;
    FOR(i, 0, sz(r)) r[i].d = min(i, mxD) + 1;
  }
  void expand(vv& R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (sz(R)) {
      if (sz(q) + R.back().d <= sz(qmax)) return;
      q.push_back(R.back().i);
      vv T;
      for (auto v : R)
        if (e[R.back().i][v.i]) T.push_back({v.i});
      if (sz(T)) {
        if (S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1,
            mnk = max(sz(qmax) - sz(q) + 1, 1);
        C[1].clear(), C[2].clear();
        for (auto v : T) {
          int k = 1;
```

```
          auto f = [&](int i) { return e[v.i][i]; };
          while (any_of(ALL(C[k]), f)) k++;
          if (k > mxk) mxk = k, C[mxk + 1].clear();
          if (k < mnk) T[j++].i = v.i;
          C[k].push_back(v.i);
        }
        if (j > 0) T[j - 1].d = 0;
        FOR(k, mnk, mxk + 1)
        for (int i : C[k]) T[j].i = i,
                           T[j++].d = k;
        expand(T, lev + 1);
      } else if (sz(q) > sz(qmax))
        qmax = q;
      q.pop_back(), R.pop_back();
    }
  }
  vi maxClique() {
    init(V), expand(V);
    return qmax;
  }
  Maxclique(vb conn)
      : e(conn), C(sz(e) + 1), S(sz(C)), old(S) {
    FOR(i, 0, sz(e)) V.push_back({i});
  }
};
```

## 7.5 SCC

```
/**
 * Descripcion: algoritmo de Tarjan para la busqueda de
 * componentes fuertemente conexos (SCC)
 * Idea: Los SCC forman subarboles en el arbol de
 * expansion de la DFS. Ademas de calcular tin(u) y low(u)
 * para cada vertice, se agrega el vertice u al final de
 * una pila z y mantenemos la informacion de que vertices
 * estan siendo explorados, mediante vi vis. Solo los
 * vertices visitados (parte del SCC actual) pueden
 * actualizar low(u). Si low(u) = tin(u), u es la raiz de
 * un SCC y sus miembros pueden obtenerse sacando de la
 * pila. Retorna un vector, donde scc[u] es el indice de
 * su SCC.
 * Tiempo: O(V + E)
 */
vi scc(vector<vi>& g) {
  int n = SZ(g), t = 0, ncomps = 0;
  vi tin(n), scc(n, -1), z;
  auto dfs = [&](auto&& self, int u) -> int {
    int low = tin[u] = ++t, x;
    z.push_back(u);
    for (auto v : g[u])
      if (scc[v] < 0)
        low = min(low, tin[v] ?: self(self, v));
    if (low == tin[u]) {
      do {
        x = z.back();
        z.pop_back();
        scc[x] = ncomps;
      } while (x != u);
      ncomps++;
    }
    return tin[u] = low;
  };
  FOR(i, 0, n) if (scc[i] == -1) dfs(dfs, i);
  return scc;
}
```

## 7.6 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
 * indexacion. Si despues de terminar la llamada a
 * solve(), white[v] = 0, v es parte de cada matching
 * maximo.
 * Tiempo: O(VE), mas rapido en la practica.
 */
struct MaxMatching {
  int N;
  vector<vi> adj;
  vector<pi> label;
  vi mate, first;
  vector<bool> white;

  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);  // label del vertice
    else
      match(label[p].first, label[p].second),
          match(label[p].second,
                label[p].first);  // label de la arista
  }
  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();  // vertice exterior
      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]);  // haz blanco a todo a
                                  // lo largo del camino
              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;
  }
};
```

## 7.7 Hopcroft Karp

```
/**
 * Descripcion: Algoritmo rapido para maximo
 * emparejamiento bipartito. el grafo g debe de ser una
 * lista de los vecinos de la particion izquierda y m el
 * numero de nodos en la particion derecha. Retorna
 * (Numero de emparejamientos, btoa[]) donde btoa[i] sera
 * el emparejamiento para el vertice i del lado derecho o
 * -1 si no lo tiene
 * Tiempo: O(sqrt(V)E)
 */
pair<int, vi> hopcroftKarp(vector<vi>& g, int m) {
  int res = 0;
  vi btoa(m, -1), A(SZ(g)), B(m), cur, next;
  auto dfs = [&](auto self, int a, int L) -> bool {
    if (A[a] != L) return 0;
    A[a] = -1;
    for (int b : g[a])
      if (B[b] == L + 1) {
        B[b] = 0;
        if (btoa[b] == -1 || self(self, btoa[b], L + 1))
          return btoa[b] = a, 1;
      }
    return 0;
  };
  while (1) {
    fill(ALL(A), 0);
    fill(ALL(B), 0);
    /// Encuentra los nodos restantes para BFS (i.e. con
    /// layer 0)
    cur.clear();
    for (int a : btoa)
      if (a != -1) A[a] = -1;
    FOR(a, 0, SZ(g)) if (A[a] == 0) cur.pb(a);
    /// Encunetra todas las layers usando BFS
    for (int lay = 1;; lay++) {
      bool islast = 0;
      next.clear();
      for (int a : cur)
        for (int b : g[a]) {
          if (btoa[b] == -1) {
            B[b] = lay;
            islast = 1;
          } else if (btoa[b] != a && !B[b]) {
            B[b] = lay;
            next.pb(btoa[b]);
          }
        }
      if (islast) break;
      if (next.empty()) return {res, btoa};
      for (int a : next) A[a] = lay;
      cur.swap(next);
    }
    /// Usa DFS para escanear caminos aumentantes
    FOR(a, 0, SZ(g)) res += dfs(dfs, a, 0);
```

```
    }
  }
```

## 7.8 Hungarian

```
/**
 * 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)
 */
template <typename T>
pair<T, vi> hungarian(const vector<vector<T>> &a) {
#define INF numeric_limits<T>::max()
  if (a.empty()) return {0, {}};
  int n = SZ(a) + 1, m = SZ(a[0]) + 1;
  vi p(m), ans(n - 1);
  vector<T> u(n), v(m);
  FOR(i, 1, n) {
    p[0] = i;
    int j0 = 0;  // agregar trabajador "dummy" 0
    vector<T> dist(m, INF);
    vi pre(m, -1);
    vector<bool> done(m + 1);
    do {  // dijkstra
      done[j0] = true;
      int i0 = p[j0], j1;
      T delta = INF;
      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) {  // actualizar camino alternativo
      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};
}
```

## 7.9 Kuhn

```
/**
 * Descripcion: Algoritmo simple para maximo
 * emparejamiento bipartito. el grafo g debe de ser una
 * lista de los vecinos de la particion izquierda y m el
 * numero de nodos en la particion derecha. Retorna
 * (|matching|, btoa) donde btoa[i] sera el emparejamiento
 * para el vertice i del lado derecho o -1 si no lo tiene
 *
 * Tiempo: O(VE)
 */
pair<int, vi> kuhn(vector<vi>& g, int m) {
  vi vis, btoa(m, -1);
  auto dfs = [&](auto self, int j) -> bool {
    if (btoa[j] == -1) return 1;
    vis[j] = 1;
    int di = btoa[j];
    for (int e : g[di])
      if (!vis[e] && self(self, e)) {
        btoa[e] = di;
        return 1;
      }
    return 0;
  };
  FOR(i, 0, SZ(g)) {
    vis.assign(SZ(btoa), 0);
    for (int j : g[i])
      if (dfs(dfs, j)) {
        btoa[j] = i;
        break;
      }
  }
  return {SZ(btoa) - (int)count(ALL(btoa), -1), btoa};
}
```

## 7.10 Minimum Vertex Cover

```
/**
 * Descripcion: Encuentra los vertices en el minimum
 * vertex cover de un grafo bipartito, si solo se quiere
 * la cardinalidad leer teoria.
 * Tiempo: depende el
 * algoritmo de matching que se use
 */

#include "Kuhn.h"  // o hopcroft karp

vi cover(vector<vi>& g, int n, int m) {
  auto [res, btoa] = kuhn(g, m);
  vector<bool> lfound(n, true), seen(m);
  for (int it : btoa)
    if (it != -1) lfound[it] = false;
  vi q, cover;
  FOR(i, 0, n) if (lfound[i]) q.push_back(i);
  while (!q.empty()) {
    int u = q.back();
    q.pop_back();
    lfound[u] = 1;
    for (int v : g[u])
      if (!seen[v] && btoa[v] != -1) {
        seen[v] = true;
        q.push_back(btoa[v]);
      }
  }
  FOR(i, 0, n) if (!lfound[i]) cover.push_back(i);
  FOR(i, 0, m) if (seen[i]) cover.push_back(n + i);
  assert(sz(cover) == res);
  return cover;
}
```

## 7.11 Kruskal

```
/**
 * Descripcion: tiene como principal funcion calcular la
 * suma del peso de las aristas del arbol minimo de
 * expansion (MST) de un grafo no dirigido, 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) Status: testeado
 * en ICPC LATAM 2017 - Imperial Roads
 */
#include <../Data Structure/DSU.h>

int kruskal(int n, vector<tuple<int, int, int>>& e) {
  sort(ALL(e));
  DSU dsu;
  dsu.init(n);
  int mst = 0;
  for (auto& [w, u, v] : e)
    if (dsu.get(u) != dsu.get(v)) {
      mst += w;
      dsu.join(u, v);
      if (--n == 1) break;
    }
  return mst;
}
```

## 7.12 Prim

```
/**
 * 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 inicia con 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 prim(vector<vector<pi>>& g) {
  vector<bool> taken(SZ(g), 0);
  priority_queue<pi> pq;
  auto process = [&](int u) -> void {
    taken[u] = 1;
    for (auto& [v, w] : g[u])
      if (!taken[v]) pq.push({-w, v});
  };
  process(0);
  int totalWeight = 0, takenEdges = 0;
  while (!pq.empty() && takenEdges != SZ(g) - 1) {
    auto [w, u] = pq.top();
    pq.pop();
    if (taken[u]) continue;
    totalWeight -= w;
    process(u);
    ++takenEdges;
  }
  return totalWeight;
}
```

## 7.13 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;
}
```

## 7.14 Topological Sort

```
/**
 * 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++)
    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);
}
```

## 7.15 Dinic

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

template <typename T>
struct Dinic {
#define INF numeric_limits<T>::max()
  struct Edge {
    int to, rev;
    T c, oc;
    T flow() {
      return max(oc - c, T(0));
    }  // 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, T c, T rcap = 0) {
    adj[a].push_back({b, SZ(adj[b]), c, c});
    adj[b].push_back({a, SZ(adj[a]) - 1, rcap, rcap});
  }
  T dfs(int v, int t, T 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 (T p = dfs(e.to, t, min(f, e.c))) {
          e.c -= p, adj[e.to][e.rev].c += p;
          return p;
        }
    }
    return 0;
  }
  T calc(int s, int t) {
    T 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[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
      }
      while (T p = dfs(s, t, INF)) flow += p;
    }
    while (lvl[t])
      ;
    return flow;
  }
  bool leftOfMinCut(int a) { return lvl[a] != 0; }
};
```

## 7.16 Johnson

```
/**
 * Descripcion: maximo flujo de coste minimo. Asume costos
 * negativos, pero no soporta ciclos negativos.
 * Tiempo:
 * FALTA!!!
 */

struct MCMF {
  using F = ll;
  using C = ll;  // tipo de flujo y de costo
  struct Edge {
    int to, rev;
    F flo, cap;
    C cost;
  };
  int N0;
  const ll INF = 1e18;
  vector<C> p, dist;
  vii pre;
  vector<vector<Edge>> adj;

  void init(int _N) {
    N0 = _N;
    p.resize(N0);
    dist.resize(N0);
    pre.resize(N0);
    adj.resize(N0);
  }
  void ae(int u, int v, F cap, C cost) {  // Agregar
                                          // arista
    assert(cap >= 0);
    adj[u].push_back(
        {v, (int)adj[v].size(), 0, cap, cost});
    adj[v].push_back(
        {u, (int)adj[u].size() - 1, 0, 0, -cost});
  }
  bool path(int s, int t) {
    dist.assign(N0, INF);
    using T = pair<C, int>;
    priority_queue<T, vector<T>, greater<T>> todo;
    todo.push({dist[s] = 0, s});
    while (todo.size()) {
      T x = todo.top();
      todo.pop();
      if (x.first > dist[x.second]) continue;
      for (auto e : adj[x.second]) {
        if (e.flo < e.cap &&
            (dist[e.to] >
             x.first + e.cost + p[x.second] - p[e.to])) {
          dist[e.to] =
              x.first + e.cost + p[x.second] - p[e.to];
          pre[e.to] = {x.second, e.rev};
          todo.push({dist[e.to], e.to});
        }
      }
    }
    return dist[t] != INF;
  }

  pair<F, C> calc(int s, int t, bool hasNegCost = false) {
    assert(s != t);
    if (hasNegCost) {  // Se encarga de costos negativos
      for (int k = 0; k < N0; k++)
        for (int i = 0; i < N0; i++)
          for (auto e : adj[i])  // Bellman-Ford, 0 index
            if (e.cap && (p[e.to] > p[i] + e.cost))
              p[e.to] = p[i] + e.cost;
    }

    F totFlow = 0;
    C totCost = 0;
    while (path(s, t)) {
      for (int i = 0; i < N0; i++) p[i] += dist[i];
      F df = INF;
      for (int x = t; x != s; x = pre[x].first) {
        Edge& e =
            adj[pre[x].first][adj[x][pre[x].second].rev];
        if (df > e.cap - e.flo) df = e.cap - e.flo;
      }

      totFlow += df;
      totCost += (p[t] - p[s]) * df;
      for (int x = t; x != s; x = pre[x].first) {
        Edge& e = adj[x][pre[x].second];
        e.flo -= df;
        adj[pre[x].first][e.rev].flo += df;
      }
    }  // Retorna el maximo flujo, costo minimo
    return {totFlow, totCost};
  }
};
```

## 7.17 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);
  }
};
```

## 7.18 Push Relabel

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

template <typename T>
struct PushRelabel {
  struct Edge {
    int dest, back;
    T f, c;
  };
  vector<vector<Edge>> g;
  vector<T> 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, T cap, T 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, T 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;
  }
  T calc(int s, int t) {
    int v = SZ(g);
    H[s] = v;
    ec[t] = 1;
    vi co(2 * v);
    co[0] = v - 1;
    FOR(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)
        if (cur[u] ==
            g[u].data() + SZ(g[u])) {  // discharge 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)
            FOR(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); }
};
```

## 7.19 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)
 */

void bellmanFord(vector<array<int, 3>>& e, int n) {
  vi d(n, INF);
  d[x] = 0;
  for (int i = 0; i < n; i++)
    for (auto& [u, v, w] : e) d[v] = min(d[v], d[u] + w);
  for (auto& [u, v, w] : e)
    if (d[u] != INF && d[u] + w < d[v]) {
      // neg cycle: all nodes reachable from u have -INF
      // distance. To reconstruct neg cycle save "prev" of
      // each node, go up from u until repeating a node.
      // this node and all nodes between the two
      // occurences form a neg cycle
    }
}
```

## 7.20 Dijkstra

```
/**
 * Descripcion: calcula el costo minimo para
 * ir desde un nodo hacia todos los demas.
 *
 * Tiempo: O(E log V)
 */
vi dijkstra(vector<vector<pi>>& g, int x) {
  int n = SZ(g);
  vi d(n);
  FOR(i, 0, n) d[i] = INF;
  d[x] = 0;
  priority_queue<pi> pq;
  pq.emplace(0, x);
  while (!pq.empty()) {
    auto [du, u] = pq.top();
    pq.pop();
    du *= -1;
```

```
    if (du > d[u]) continue;
    for (auto& [v, w] : g[u])
      if (du + w < d[v]) {
        d[v] = du + w;
        pq.emplace(-d[v], v);
      }
  }
  return d;
}
// Para tener a lo mucho V elementos en la PQ
vi dijkstra(vector<vector<pi>>& g, int st) {
  int n = SZ(g);
  vi d(n);
  set<pi> pq;
  FOR(i, 0, n) pq.emplace(d[i], i), d[i] = INF;
  while (!pq.empty()) {
    auto [du, u] = *pq.begin();
    pq.erase(pq.begin());
    for (auto& [v, w] : g[u])
      if (du + w < d[v]) {
        pq.erase(pq.find({d[v], v}));
        d[v] = du + w;
        pq.emplace(d[v], v);
      }
  }
  return d;
}
```

## 7.21 Floyd-Warshall

```
/**
 * Descripcion: algoritmo de Floyd-Warshall para calcular
 * la minima distancia entre cada par de nodos, si no se
 * requiere recalcular el camino, ignorar p. Retorna un
 * par (g, p), en donde g es una matriz modificada en la
 * que g[i][j] es el costo minimo para llegar desde el
 * nodo i al nodo j y p es el nodo anterior en dicho
 * camino, utilizado para recalcular la ruta.
 * Tiempo:
 * O(V^3)
 */

pair<vector<vi>, vi> floydWarshall(vector<vi> g) {
  int n = SZ(g);
  vector<vi> p(n, vi(n));
  FOR(i, 0, n) FOR(j, 0, n) p[i][j] = i;
  FOR(k, 0, n)
  FOR(i, 0, n)
      FOR(j, 0, n) if (g[i][k] + g[k][j] < g[i][j]) {
    p[i][j] = p[k][j];
    g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
  }
  return {g, p};
}
void printPath(vector<vi>& p, int i, int j) {
  if (i != j) printPath(p, i, p[i][j]);
  cout << j << " ";
}
```

## 7.22 Binary Lifting LCA

```
/**
 * Descripcion: siendo jmp[i][j] el ancestro 2^j del nodo
 * i, el binary liftingnos 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 jmp[i][0] para
 * todo i.
 * Tiempo: O(n log n) en construccion y O(log n)
 * por consulta Status: testeado en ICPC LATAM 2017 -
 * Imperial Roads
 */
vi g[maxn];
int jmp[maxn][maxlog], d[maxn];
void dfs(int u, int p = -1) {
  jmp[u][0] = p;
  for (auto &v : g[u])
    if (v != p) d[v] = d[u] + 1, dfs(v, u);
}
void build() {
  int n = SZ(g);
  dfs(0);
  for (int i = 1; i < maxlog; i++)
    for (int u = 0; u < n; u++)
      if (jmp[u][i - 1] != -1)
        jmp[u][i] = jmp[jmp[u][i - 1]][i - 1];
}
int LCA(int u, int v) {
  if (d[u] < d[v]) swap(u, v);
  int dist = d[u] - d[v];
  for (int i = maxlog - 1; i >= 0; i--)
    if ((dist >> i) & 1) u = jmp[u][i];
  if (u == v) return u;
  for (int i = maxlog - 1; i >= 0; i--)
    if (jmp[u][i] != jmp[v][i])
      u = jmp[u][i], v = jmp[v][i];
  return jmp[u][0];
}
int dist(int u, int v) {
  return d[u] + d[v] - 2 * d[LCA(u, v)];
}
```

## 7.23 Centroid Decomposition

```
/**
 * Descripcion: cuando se trabaja con caminos en
 * un arbol, es util descomponer a este recursivamente
 * en sub-arboles formados al eliminar su centroide, el
 * centroide de un arbol es un nodo u tal que si lo
 * eliminas, este se divide en sub-arboles con un numero
 * de nodosno mayor a la mitad del original, todos los
 * arboles tienen un centroide, y a lo mas 2. Esto provoca
 * que el arbol sea dividido en sub-arboles de distintos
 * niveles de descomposicion, por comodidad, un nodo v es
 * un centroide ancestro de otro nodo u, si v, en algun
 * nivel, fue el centroide que separo al componente de
 * u en sub-arboles. Todo camino del arbol original se
 puede
 * expresar como la concatenacion de dos caminos del tipo:
       (u, A(u)), (u, A(A(u))), (u, A(A(A(u))))..., etc.
 * Ya que en cada nivel k el numero de nodos de algun
 * componente es a lo mas |V| / 2^k, un nodo puede estar
 en
 * log |V| componentes, es decir, puede tener como maximo
 * log |V| ancestros en el arbol de centroides.
 *
 *
 * Tiempo: O(|V| log |V|)
 * Status: tested on Codeforces 321C
 */
int n;
bool tk[maxn];
int szt[maxn], fat[maxn];
vi g[maxn];
int calcsz(int u, int f) {
  szt[u] = 1;
  for (auto v : g[u])
    if (v != f && !tk[v]) szt[u] += calcsz(v, u);
  return szt[u];
}
void cdfs(int x = 0, int f = -1,
          int sz = -1) {  // O(nlogn)
  if (sz < 0) sz = calcsz(x, -1);
  for (auto v : g[x])
    if (!tk[v] && szt[v] * 2 >= sz) {
      szt[x] = 0;
      cdfs(v, f, sz);
      return;
    }
  tk[x] = true;
  fat[x] = f;
  for (auto v : g[x])
    if (!tk[v]) cdfs(v, x);
}
void decompose() {
  memset(tk, false, sizeof(tk));
  cdfs();
}
```

## 7.24 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 in[maxn], out[maxn], t = 0;
void dfs(int u, int p) {
  in[u] = ++t;
  for (auto& v : g[u])
    if (v != p) dfs(v, u);
  out[u] = t;
}
```

## 7.25 HLD

```
/*
 * Heavy-Light Decomposition
 * Descripcion: descompone un arbol en caminos pesados y
 * aristas ligeras de tal manera que un camino de
 * cualquier hoja a la raiz contiene a lo mucho log(n)
 * aristas ligeras. Raiz debe ser 0 Si el valor lo
 * contiene las aristas, asignar VALS_EDGES true
 * Tiempo:
 * O((log N)^2)
 */

// Incluir Lazy Segment Tree

template <bool VALS_EDGES>
struct HLD {
  int N, tim = 0;
  vector<vi> adj;
  vi par, siz, head, pos;
  LazySegmentTree<int> tree;
  HLD(vector<vi> adj_)
      : N(sz(adj_)),
        adj(adj_),
        par(N, -1),
        siz(N, 1),
        head(N),
        pos(N) {
    tree.init(N);
    dfsSz(0);
    dfsHld(0);
  }
  void dfsSz(int v) {
    if (par[v] != -1)
      adj[v].erase(find(ALL(adj[v]), par[v]));
    for (int& u : adj[v]) {
      par[u] = v;
      dfsSz(u);
      siz[v] += siz[u];
      if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
```

```
      }
    }
    void dfsHld(int v) {
      pos[v] = tim++;
      for (int u : adj[v]) {
        head[u] = (u == adj[v][0] ? head[v] : u);
        dfsHld(u);
      }
    }
    template <class B>
    void process(int u, int v, B op) {
      for (; head[u] != head[v]; v = par[head[v]]) {
        if (pos[head[u]] > pos[head[v]]) swap(u, v);
        op(pos[head[v]], pos[v] + 1);
      }
      if (pos[u] > pos[v]) swap(u, v);
      op(pos[u] + VALS_EDGES, pos[v] + 1);
    }
    void modifyPath(int u, int v, int val) {
      process(u, v,
              [&](int l, int r) { tree.add(l, r, val); });
    }
    int queryPath(int u, int v) {
      int res = -1e9;
      process(u, v, [&](int l, int r) {
        res = max(res, tree.query(l, r));
      });
      return res;
    }
    int querySubtree(int v) {  // modifySubtree es similar
      return tree.query(pos[v] + VALS_EDGES,
                        pos[v] + siz[v]);
    }
  };
```

# 8 Strings

## 8.1 Aho Corasick

```
/*
 * Descripcion: Construye un automata que te permite
 * buscar rapidamente multiples patrones en un texto.
 * Uso:
 * - Se inicializa con AhoCorasick ac(patrones)
 * - find(word): regresa por cada posicion de word el
 * indice de la palabra mas larga que termina ahi o -1 si
 * no existe
 * - findAll(pat, word) encuentra todas las palabras que
 * empiezan por cada posicion de word (las mas cortas
 * primero) Notas:
 * - Patrones duplicados son permitidos (afecta en tiempo)
 * - Patrones vacios no permitidos
 * - El nodo de inicio del automata esta en el indice 0
 * - Para encontrar las palabras mas grande en cada
 * posicion, invierte la entrada
 * Tiempo:
 * - Contruccion: O(26N)
 * - find(x): O(N)
 * - findAll: O(NM) o hasta O(N sqrt(N)) si no hay
 * patrones duplicados
 */

struct AhoCorasick {
  enum { alpha = 26, first = 'a' };
  struct Node {
    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++;
  }
  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);
        }
      }
    }
  }
  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;
  }
};
```

## 8.2 Dynamic Aho Corasick

```
/*
 * Descripcion: Si tenemos N cadenas en el diccionario,
 * mantenga log(N) Aho Corasick automatas. El i-esimo
 * automata contiene las primeras 2^k cadenas no incluidas
 * en el automatas anteriores. Por ejemplo, si tenemos N =
 * 19, necesitamos 3 automatas: {s[1]...s[16]},
 * {s[17]...s[18]} y {s[19]}. Para responder a la
 * consulta, podemos atravesar los automatas 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
 * automatas 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 {
  vector<vector<string>> dict;
  vector<AhoCorasick> ac;

 public:
  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;
    }
  };
```

## 8.3 Evil Hashing

```
/*
 * Evil Hashing
 * Descripcion: utiliza aritmetica modulo 2^64 - 1, el
 * doble de lento que mod 2^64 y con mas codigo, pero
 * funciona bien con casos malvados (como Thue-Morse,
 * donde ABBA... y BAAB... de longitud 2^10 tienen el
 * mismo hash mod 2^64) Utilizar "using H = ull;" si se
 * cree que los casos son aleatorios, o trabaja mod 10^9+7
 * si la paradoja del cumpleanos no es un problema.
 * Utiliza cosas de c++20.
 *
 * Uso:
 *  Normal.
 *     string s;
 *     HashInterval hi(s);
 *     hi.hashInterval(0, 5);
 *  Rabin-Karp.
 *     string s;
 *     getHashes(s, SZ(s));
 *     ....
 *
 * Tiempo: O(|s|)
 */

using ull = uint64_t;
struct H {
  ull x;
  H(ull x = 0) : x(x) {}
  H operator+(H o) { return x + o.x + (x + o.x < x); }
  H operator-(H o) { return *this + ~o.x; }
  H operator*(H o) {
    auto m = (__uint128_t)x * o.x;
    return H((ull)m) + (ull)(m >> 64);
  }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get() == o.get(); }
  bool operator<(H o) const { return get() < o.get(); }
};
static const H C =
    (ll)1e11 + 3;  // (order ~ 3e9; random also ok)

struct HashInterval {
  vector<H> ha, pw;
  HashInterval(string& str) : ha(sz(str) + 1), pw(ha) {
    pw[0] = 1;
    FOR(i, 0, sz(str))
    ha[i + 1] = ha[i] * C + str[i], pw[i + 1] = pw[i] * C;
  }
  H hashInterval(int a, int b) {  // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
};

vector<H> getHashes(string& str, int length) {
  if (sz(str) < length) {
    return {};
  }
  H h = 0, pw = 1;
  FOR(i, 0, length) {
    h = h * C + str[i];
    pw = pw * C;
  }
  vector<H> ret = {h};
  FOR(i, length, sz(str)) {
    ret.push_back(h = h * C + str[i] -
                      pw * str[i - length]);
  }
  return ret;
}

H hashString(string& s) {
  H h{};
  for (char c : s) {
    h = h * C + c;
  };
  return h;
}
```

## 8.4 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]);
      }
    }
  }

  ll value(int l, int r) {  // hash [i, j)
    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]);
    }
  }
}
```

## 8.5 Hashing Dynamic

```
/*
 * Hashing Dinamico
 * Descripcion: Convierte strings en hashes para
 * compararlos eficientemente
 * - Util para comparar strings o un substring de este
 * - Tambien puede cambiar un caracter del string
 * eficientemente
 * Uso:
 * - hash.get(inicio, fin); [inicio, fin)
 *   comparar dos string hash.get(l, f) == hash.get(l, f)
 * - set(posicion, caracter) indexado en 0
 *   Cambia el caracter de una posicion en el string
 * Aplicaciones:
 * - Checar si substrings de un string son palindromos
 *
 * Complejidad:
 * - Construccion O(n log(n))
 * - Query y update O(log(n))
 */

#include <bits/stdc++.h>
// Pura gente del coach moy
using namespace std;

typedef long long ll;
typedef pair<ll, ll> ii;

const ll MOD = 998244353;
const ii BASE = {1e9 + 7, 1e9 + 9};

ii operator+(const ii a, const ii b) {
  return {(a.first + b.first) % MOD,
          (a.second + b.second) % MOD};
}

ii operator+(const ii a, const ll b) {
  return {(a.first + b) % MOD, (a.second + b) % MOD};
}

ii operator-(const ii a, const ii b) {
  return {(MOD + a.first - b.first) % MOD,
          (MOD + a.second - b.second) % MOD};
}

ii operator*(const ii a, const ii b) {
  return {(a.first * b.first) % MOD,
          (a.second * b.second) % MOD};
}

ii operator*(const ii a, const ll b) {
  return {(a.first * b) % MOD, (a.second * b) % MOD};
}

inline ll modpow(ll x, ll p) {
  if (!p) return 1;
  return (modpow(x * x % MOD, p >> 1) * (p % 2 ? x : 1)) %
         MOD;
}

inline ll modinv(ll x) { return modpow(x, MOD - 2); }

struct Hash_Bit {
  int N;
  string S;
  vector<ii> fen, pp, ipp;
```

```
  Hash_Bit(string S_) {
    S = S_;
    N = S.size();
    fen.resize(N + 1);
    pp.resize(N);
    ipp.resize(N);
    pp[0] = ipp[0] = {1, 1};
    const ii ibase = {modinv(BASE.first),
                      modinv(BASE.second)};

    for (int i = 1; i < N; i++) {
      pp[i] = pp[i - 1] * BASE;
      ipp[i] = ipp[i - 1] * ibase;
    }

    for (int i = 0; i < N; i++) {
      update(i, S[i]);
    }
  }

  void update(int i, ll x) {
    ii p = pp[i] * x;
    for (++i; i <= N; i += i & -i) {
      fen[i] = fen[i] + p;
    }
  }

  ii query(int i) {
    ii ret = {0, 0};
    for (; i; i -= i & -i) {
      ret = (ret + fen[i]);
    }
    return ret;
  }

  void set(int idx, char c) {
    int d = (MOD + c - S[idx]) % MOD;
    S[idx] = c;
    update(idx, d);
  }

  ii get(int start, int end) {
    return (query(end) - query(start)) * ipp[start];
  }
};

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

  int sz = s.size();
  Hash_Bit hash(s);

  return 0;
}
```

## 8.6 KMP

```
/*
 * Descripcion: Calcula la funcion phi para un string S
 * donde phi[i] es la longitud del prefijo propio de S mas
 * largo de la subcadena S[0..i] el cual tambien es sufijo
 * 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;
}
```

## 8.7 KMPAutomaton

```
/*
 * Descripcion: Construye un automata del KMP donde el
 * estado es el valor actual de la funcion phi
 * Uso:
 * aut[state][nextCharacter]
 * Tiempo: 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]);
    }
  }
}
```

## 8.8 Manacher

```
/*
 * Descripcion: para cada posicion de un string, calcula:
 * p[0][i] = la mitad de la longitud del palindromo par
 * mas grande alrededor de la posicion i
 * p[1][i] = la longitud del palindromo impar mas grande
 * alrededor de la posicion i (la mitad redondeado hacia
 * abajo)
 * Tiempo: O(N)
 */

array<vi, 2> manacher(const string& s) {
  int n = SZ(s);
  array<vi, 2> p = {vi(n + 1), vi(n)};
  FOR(z, 0, 2)
  for (int i = 0, l = 0, r = 0; i < n; i++) {
    int t = r - i + !z;
    if (i < r) p[z][i] = min(t, p[z][l + t]);
    int L = i - p[z][i], R = i + p[z][i] - !z;
    while (L >= 1 && R + 1 < n && s[L - 1] == s[R + 1])
      p[z][i]++, L--, R++;
    if (R > r) l = L, r = R;
  }
  return p;
}
```

## 8.9 Palindromic Tree

```
/*
 * Descripcion: Calcula un EerTree el cual es un arbol que
 * permite un rapido acceso a todos los palindromos
 * contenidos en un string como encontrar el palindromo
 * mas largo en un substring
 * Uso: Considerar a s como el
 * string original y p el palindromo asociado al nodo u
 * - et[u].l: la longitud de p
 * - et[u].cnt: es el numero de veces que aparece p como
 * el sufijo palindromo mas largo de un prefijo s[0:i].
 * Despues de llamar computeFrecuency() es el numero de
 * veces en el que p aparece en s
 * - et[u].link: es el nodo que correspondiente al
 * palindromo propio mas largo de p
 * Tiempo: O(N)
 */

constexpr short alpha = 26;
constexpr char offset = 'a';
struct state {
  int l, link, cnt;
  array<int, alpha> go;
  state() {
    l = cnt = 0;
    go.fill(0);
  }
};
struct PalindromicTree {
  int n = 2;
  int last = 1;
  vector<state> et;
  string s;
  PalindromicTree() {
    et.resize(2);
    et[0].link = et[1].link = 1;
    et[1].l = -1;
  }
  int palSuff(int x) {
    while (s[SZ(s) - 2 - et[x].l] != s.back())
      x = et[x].link;
    return x;
  }
  int add(char ch) {
    s.pb(ch);
    last = palSuff(last);
    bool new_pal = !et[last].go[ch - offset];
    if (new_pal) {
      et.pb(state());
      et[last].go[ch - offset] = n++;
      et.back().link =
          et[palSuff(et[last].link)].go[ch - offset];
      et.back().l = et[last].l + 2;
      if (et.back().l == 1) et.back().link = 0;
    }
    last = et[last].go[ch - offset];
    // Do something with last, maybe if new_pal
    et[last].cnt++;
    if (et[last].l == SZ(s)) last = et[last].link;
    return new_pal;
  }
  void computeFrequency() {
    for (int i = n - 1; i > 1; i--)
      et[et[i].link].cnt += et[i].cnt;
  }
};
```

## 8.10 Suffix Array

```cpp
/*
 * Descripcion: construye un arreglo ordenado de todos los
 * sufijos de un string
 * Uso:
 * - SA[i]: es el indice de inicio del sufijo el cual es
 * el i-nesimo elemento del suffix array (el arreglo es de
 * tamano n+1 y SA[0] = n)
 * - LCP: es un arreglo que contiene el prefijo comun mas
 * largo entre los strings vecinos del suffix array LCP[i]
 * = LCP(sa[i], sa[i-1]), LCP[0] = 0
 * Tiempo: O(n log n)
 */

struct SuffixArray {
  vi SA, LCP;
  string S;
  int n;
  SuffixArray(string &s, int lim = 256)
      : S(s), n(SZ(s) + 1) {
    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] >= j) y[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)
   */
  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;
  }
};
```

## 8.11 Suffix Automaton

```cpp
/*
 * Descripcion: Construye un automata finito que reconoce
 * todos los sufijos de una cadena. len corresponde a la
 * longitud maxima de un sufijo con el estado actual, link
 * corresponde al estado del siguiente sufijo mas largo
 * del estado actual
 * Tiempo: O(n log sum)
 */

struct SuffixAutomaton {
  struct state {
    int len, link;
    map<char, int> next;
  };
  vector<state> st;
  int sz, last;
  SuffixAutomaton(int MAX = 1e5) {
    st.resize(MAX);
    last = st[0].len = 0;
    sz = 1;
    st[0].link = -1;
  }
  void extend(char c) {
    int k = sz++, p;
    st[k].len = st[last].len + 1;
    for (p = last; p != -1 && !st[p].next.count(c);
         p = st[p].link)
      st[p].next[c] = k;
    if (p == -1)
      st[k].link = 0;
    else {
      int q = st[p].next[c];
      if (st[p].len + 1 == st[q].len)
        st[k].link = q;
      else {
        int w = sz++;
        st[w].len = st[p].len + 1;
        st[w].next = st[q].next;
        st[w].link = st[q].link;
        for (; p != -1 && st[p].next[c] == q;
             p = st[p].link)
          st[p].next[c] = w;
        st[q].link = st[k].link = w;
      }
    }
    last = k;
  }
};
```

## 8.12 Suffix Tree

```cpp
/**
 * Descripcion: Algoritmo de Ukkonen la construccion de
 * online arbol de sufijo. Cada nodo contiene indices
 * [l,r) en la cadena y una lista de nodos hijos. Los
 * sufijos estan dados por recorridos de este arbol
 * uniendo subcadenas [l,r). La raiz es 0 (tiene l = -1, r
 * = 0) los hijos inexistentes son -1 Para obtener un
 * arbol completo, agregue un simbolo ficticio al final
 *
 * Tiempo: O(26N)
 */

struct SuffixTree {
  enum { N = 200010, ALPHA = 26 };  // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a;  // v = cur node, q = cur position
  int t[N][ALPHA], l[N], r[N], p[N], s[N], v = 0, q = 0,
                                           m = 2;

  void ukkadd(int i, int c) {
  suff:
    if (r[v] <= q) {
      if (t[v][c] == -1) {
        t[v][c] = m;
        l[m] = i;
```

```
        p[m++] = v;
        v = s[v];
        q = r[v];
        goto suff;
      }
      v = t[v][c];
      q = l[v];
    }
    if (q == -1 || c == toi(a[q]))
      q++;
    else {
      l[m + 1] = i;
      p[m + 1] = m;
      l[m] = l[v];
      r[m] = q;
      p[m] = p[v];
      t[m][c] = m + 1;
      t[m][toi(a[q])] = v;
      l[v] = q;
      p[v] = m;
      t[p[m]][toi(a[l[m]])] = m;
      v = s[p[m]];
      q = l[m];
      while (q < r[m]) {
        v = t[v][toi(a[q])];
        q += r[v] - l[v];
      }
      if (q == r[m])
        s[m] = v;
      else
        s[m] = m + 2;
      q = r[v] - (q - r[m]);
      m += 2;
      goto suff;
    }
  }

  SuffixTree(string a) : a(a) {
    fill(r, r + N, SZ(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1], t[1] + ALPHA, 0);
    s[0] = 1;
    l[0] = l[1] = -1;
    r[0] = r[1] = p[0] = p[1] = 0;
    FOR(i, 0, SZ(a)) ukkadd(i, toi(a[i]));
  }

  // example: find longest common substring (uses ALPHA =
  // 28)
  pii best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0,
        len = node ? olen + (r[node] - l[node]) : 0;
    FOR(c, 0, ALPHA)
    if (t[node][c] != -1) mask |=
        lcs(t[node][c], i1, i2, len);
    if (mask == 3) best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t +
                  (char)('z' + 2));
    st.lcs(0, SZ(s), SZ(s) + 1 + SZ(t), 0);
    return st.best;
  }
};
```

## 8.13 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;
  }
};
```

## 8.14 ZAlgorithm

```
/*
 * 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;
}
```

# 9 Geometry

## 9.1 Circle

```
// Retorna el punto central del circulo que pasa por A,B,C
// Si se busca el radio solo sacar la distancia entre el
// centro y cualquier punto A,B,C
Point circumCenter(Point a, Point b, Point c) {
  b = b - a, c = c - a;
  assert(b.cross(c) !=
         0);  // no existe circunferencia colinear
  return a +
         (b * sq(c) - c * sq(b)).perp() / b.cross(c) / 2;
}

// Retorna el punto que se encuentra en el circulo dado el
// angulo
Point circlePoint(Point c, double r, double ang) {
  return Point{c.x + cos(ang) * r, c.y + sin(ang) * r};
}

// Retorna el numero de intersecciones de la linea l con
// el circulo (o,r) y los pone en out. Si solo hay una
// interseccion el par de out es igual
int circleLine(Point o, double r, Line l,
               pair<Point, Point> &out) {
  double h2 = r * r - l.sqDist(o);
  if (h2 >= 0) {
    Point p = l.proj(o);
    Point h = l.v * sqrt(h2) / l.v.norm();
    out = {p - h, p + h};
  }
  return 1 + sgn(h2);
}

// Retorna las intersecciones entre dos circulos. Funciona
// igual que la interseccion con una linea
int circleCircle(Point o1, double r1, Point o2, double r2,
                 pair<Point, Point> &out) {
  Point d = o2 - o1;
  double d2 = d.sq();
  if (d2 == 0) {
    assert(r1 != r2);  // los circulos son iguales
    return 0;
  }
  double pd = (d2 + r1 * r1 - r2 * r2) / 2;
  double h2 = r1 * r1 - pd * pd / d2;
  if (h2 >= 0) {
    Point p = o1 + d * pd / d2,
          h = d.perp() * sqrt(h2 / d2);
    out = {p - h, p + h};
  }
  return 1 + sgn(h2);
}

// Retorna un booleano indicando si los dos circulos
// intersectan o no
bool circleCircle(Point o1, double r1, Point o2,
                  double r2) {
  double dx = o1.x - o2.x, dy = o1.y - o2.y, rs = r1 + r2;
  return dx * dx + dy * dy <= rs * rs;
}

// Retorna el area de la interseccion de un circulo con
// un poligono ccw
// Tiempo O(n)
#define arg(p, q) atan2(p.cross(q), p.dot(q))
double circlePoly(Point c, double r, vector<Point> ps) {
  auto tri =
      [&](Point p,
          Point q) {  // area de interseccion con cpq
        auto r2 = r * r / 2;
        Point d = q - p;
        auto a = d.dot(p) / d.sq(),
             b = (p.sq() - r * r) / d.sq();
        auto det = a * a - b;
        if (det <= 0) return arg(p, q) * r2;
        auto s = max(0., -a - sqrt(det)),
             t = min(1., -a + sqrt(det));
        if (t < 0 || 1 <= s) return arg(p, q) * r2;
        Point u = p + d * s, v = p + d * t;
        return arg(p, u) * r2 + u.cross(v) / 2 +
               arg(v, q) * r2;
      };
  auto sum = 0.0;
  FOR(i, 0, SZ(ps))
  sum += tri(ps[i] - c, ps[(i + 1) % SZ(ps)] - c);
  return sum;
}

// Retorna el numero de tangentes de tipo especifico
// (inner, outer)
// * Si hay 2 tangentes. Out se llena con 2 pares de
// puntos:
//   los pares de puntos de tangencia de cada circulo
//   (P1,P2)
// * Si solo hay 1 tangente, los circulo son tangentes en
// algun
//   punto P, out contiene P 4 veces y la linea tangente
//   puede ser encontrada como line(o1,p).perp(p)
// * Si hay 0 tangentes, no hace nada
// * Si los circulos son identicos, aborta
int tangents(Point o1, double r1, Point o2, double r2,
             bool inner,
             vector<pair<Point, Point>> &out) {
  if (inner) r2 = -r2;
  Point d = o2 - o1;
  double dr = r1 - r2, d2 = d.sq(), h2 = d2 - dr * dr;
  if (d2 == 0 || h2 < 0) {
    assert(h2 != 0);
    return 0;
  }
  for (double sign : {-1, 1}) {
    Point v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;
    out.push_back({o1 + v * r1, o2 + v * r2});
  }
  return 1 + (h2 > 0);
}
```

## 9.2 Closest Pair

```
/*
 * Descripcion: Dado un arreglo de N puntos en el plano,
 * encontrar el par de puntos con la menor distancia entre
 * ellos Utilizar con long long de preferencia
 * Tiempo: O(n
 * log n)
 */

typedef Point P;
pair<Point, Point> closest(vector<Point> &v) {
  set<Point> S;
  sort(ALL(v),
       [](Point a, Point b) { return a.y < b.y; });
  pair<ll, pair<Point, Point>> ret{LLONG_MAX,
                                   {P{0, 0}, P{0, 0}}};
  int j = 0;
  for (Point p : v) {
    Point d{1 + (ll)sqrt(ret.first), 0};
    while (v[j].y <= p.y - d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d),
         hi = S.upper_bound(p + d);
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).sq(), {*lo, p}});
    S.insert(p);
  }
  return ret.second;
}
```

## 9.3 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)
 */

vector<Point> convexHull(vector<Point> pts) {
  if (SZ(pts) <= 1) return pts;
  sort(ALL(pts));
  vector<Point> h(SZ(pts) + 1);
  int s = 0, t = 0;
  for (int it = 2; it--; s = --t, reverse(ALL(pts)))
    for (Point p : pts) {
      while (t >= s + 2 &&
             h[t - 2].cross(h[t - 1], p) <= 0)
        t--;  // quitar = si se incluye colineares
      h[t++] = p;
    }
  return {h.begin(),
          h.begin() + t - (t == 2 && h[0] == h[1])};
}
```

## 9.4 Half Plane

```
/*
 * Descripcion: Dado un conjunto de semiplanos calcula la
 * interseccion de estos representandolos en un poligono
 * convexo. Donde cada punto dentro del poligono esta
 * dentro de todos los semiplanos
 * - Cada semiplano apunta en su region izquierda
 * - Se asume que no hay semiplanos paralelos
 *
 * Tiempo: O(N Log N)
 */

typedef double T;
const double EPS = 1e-9;
const double INF = 1e9;

struct HalfPlane {
  Point p, pq;
  T angle;
  HalfPlane() {}
  HalfPlane(Point _p, Point _q)
      : p(_p), pq(_q - _p), angle(atan2(pq.y, pq.x)) {}
  bool operator<(HalfPlane b) const {
    return angle < b.angle;
  }
  bool out(Point q) { return pq.cross(q - p) < EPS; }
  Point intersect(HalfPlane l) {
    if (abs(pq.cross(l.pq)) < EPS) return Point{INF, INF};
    return l.p +
           l.pq * ((p - l.p).cross(pq) / l.pq.cross(pq));
  }
};

vector<Point> intersect(vector<HalfPlane> b) {
  vector<Point> bx = {Point{INF, INF}, Point{-INF, INF},
                      Point{-INF, -INF},
                      Point{INF, -INF}};
  for (int i = 0; i < 4; ++i)
    b.emplace_back(bx[i], bx[(i + 1) % 4]);
  sort(b.begin(), b.end());
  int n = b.size(), q = 1, h = 0;
  vector<HalfPlane> c(b.size() + 10);
  for (int i = 0; i < n; ++i) {
    while (q < h && b[i].out(c[h].intersect(c[h - 1])))
      h--;
    while (q < h && b[i].out(c[q].intersect(c[q + 1])))
      q++;
    c[++h] = b[i];
```

```
    if (q < h && abs(c[h].pq.cross(c[h - 1].pq)) < EPS) {
      if (c[h].pq.dot(c[h - 1].pq) <= 0) return {};
      h--;
      if (b[i].out(c[h].p)) c[h] = b[i];
    }
  }
  while (q < h - 1 && c[q].out(c[h].intersect(c[h - 1])))
    h--;
  while (q < h - 1 && c[h].out(c[q].intersect(c[q + 1])))
    q++;
  if (h - q <= 1) return {};
  c[h + 1] = c[q];
  vector<Point> s;
  for (int i = q; i < h + 1; ++i)
    s.push_back(c[i].intersect(c[i + 1]));
  return s;
}
```

## 9.5 Line

```
typedef ll T;
struct Line {
  Point v;
  T c;

  // De vector direccional v y offset c
  Line(Point v, T c) : v(v), c(c) {}

  // De la ecuacion ax+by=c
  Line(T a, T b, T c) : v({b, -a}), c(c) {}

  // De punto P a punto Q
  Line(Point p, Point q) : v(q - p), c(v.cross(p)) {}

  // 0 si se encuentra en la linea, > 0 arriba, < 0 abajo
  T side(Point p) { return v.cross(p) - c; }

  double dist(Point p) { return abs(side(p)) / v.norm(); }
  double sqDist(Point p) {
    return side(p) * side(p) / (double)v.sq();
  }  // si se trabaja con enteros
  Line perp(Point p) { return {p, p + v.perp()}; }
  Line translate(Point t) { return {v, c + v.cross(t)}; }
  Line shiftLeft(double dist) {
    return {v, c + dist * v.norm()};
  }

  Point proj(Point p) {
    return p - v.perp() * side(p) / v.sq();
  }  // Punto en linea mas cercano a P
  Point refl(Point p) {
    return p - v.perp() * 2 * side(p) / v.sq();
  }

  // Sirve para comparar si un punto A esta antes de B en
  // una linea
  bool cmpProj(Point p, Point q) {
    return v.dot(p) < v.dot(q);
  }
};

bool areParallel(Line l1, Line l2) {
  return (l1.v.cross(l2.v) == 0);
}
bool areIntersect(Line l1, Line l2, Point& p) {
  T d = l1.v.cross(l2.v);
  if (d == 0)
    return false;  // cambiar a epsilon si es double
  p = (l2.v * l1.c - l1.v * l2.c) / d;  // requiere double
  return true;
}

// Un angulo bisector de dos lineas es una linea que forma
// angulos iguales con l1 y l2
Line bisector(Line l1, Line l2, bool interior) {
  assert(l1.v.cross(l2.v) !=
         0);  // l1 y l2 no pueden ser paralelas
  double sign = interior ? 1 : -1;
  return {l2.v / l2.v.norm() + l1.v / l1.v.norm() * sign,
          l2.c / l2.v.norm() + l1.c / l1.v.norm() * sign};
}
```

## 9.6 Point

```
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;

int sgn(T x) { return (T(0) < x) - (x < T(0)); }

struct Point {
  T x, y;

  // Operaciones Punto - Punto
  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*(Point b) const {
    return {x * b.x - y * b.y, x * b.y + y * b.x};
  }

  // Operaciones Punto - Numero
  Point operator*(T d) const { return {x * d, y * d}; }
  Point operator/(T d) const {
    return {x / d, y / d};
  }  // Solo para punto flotante

  // Operaciones de comparacion para punto flotante
  bool operator<(Point p) const {
    return x < p.x - EPS ||
           (abs(x - p.x) <= EPS && y < p.y - EPS);
  }
  bool operator==(Point p) const {
    return abs(x - p.x) <= EPS && abs(y - p.y) <= EPS;
  }
  bool operator!=(Point p) const { return !(*this == p); }

  // Operaciones de comparacion para enteros
  bool operator<(Point p) const {
    return tie(x, y) < tie(p.x, p.y);
  }
  bool operator==(Point p) const {
    return tie(x, y) == tie(p.x, p.y);
  }

  T sq() { return x * x + y * y; }
  double norm() { return sqrt(sq()); }
  Point unit() { return *this / norm(); }

  // Operaciones generales:
  Point translate(Point v) { return *this + v; }
  Point scale(Point c, double factor) {
    return c + (*this - c) * factor;
  }
  Point rotate(double ang) {
    return {x * cos(ang) - y * sin(ang),
            x * sin(ang) + y * cos(ang)};
  }
  Point rot_around(double ang, Point c) {
    return c + (*this - c).rotate(ang);
  }
  Point perp() { return {-y, x}; }

  T dot(Point p) { return x * p.x + y * p.y; }
  T cross(Point p) const { return x * p.y - y * p.x; }
  T cross(Point a, Point b) const {
    return (a - *this).cross(b - *this);
  }
  double angle() const { return atan2(y, x); }

  friend ostream& operator<<(ostream& os, Point p) {
    return os << "(" << p.x << "," << p.y << ")";
  }
};

// Vector: p2-p1
double dist(Point p1, Point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y);
}
bool isPerp(Point v, Point w) { return v.dot(w) == 0; }
//-1 -> left / 0 -> collinear / +1 -> right
T orient(Point a, Point b, Point c) {
  return a.cross(b, c);
}
bool cw(Point a, Point b, Point c) {
  return orient(a, b, c) < EPS;
}
bool ccw(Point a, Point b, Point c) {
  return orient(a, b, c) > -EPS;
}

// ANGULOS
// Para c++17
double angle(Point v, Point w) {
  return acos(
      clamp(v.dot(w) / v.norm() / w.norm(), -1.0, 1.0));
}
// C++14 o menor
double angle(Point v, Point w) {
  double cosTheta = v.dot(w) / v.norm() / w.norm();
  return acos(max(-1.0, min(1.0, cosTheta)));
}
// angulo aob
double angle(Point o, Point a, Point b) {
  return angle(a - o, b - o);
}
double orientedAngle(Point o, Point a, Point b) {
  if (ccw(o, a, b))
    return angle(a - o, b - o);
  else
    return 2 * PI - angle(a - o, b - o);
}
bool inAngle(Point o, Point a, Point b, Point p) {
  assert(orient(o, a, b) != 0);
  if (cw(o, a, b)) swap(b, c);
  return ccw(o, a, p) && cw(o, c, p);
}
```

## 9.7 Point3D

```
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
  retrurn(q - p) % (r - p) |
      n;  /// equivalent to 2D cross on PI (of ortogonal
          /// proj)
}
```

## 9.8 Polar Sort

```
/*
 * Descripcion: ordena los puntos segun el angulo.
 * Comienza a partir de la izquierda en contra de las
 * manecillas
 */

int half(Point p) {
  return p.y > 0 || (p.y == 0 && p.x < 0);
}

// Pro-tip: si los puntos se encuentran en la misma
// direccion son considerados iguales, entonces se
// ordenaran arbitrariamente. Si se busca un desempate, se
// puede usar la magnitud sq(v)
void polarSort(vector<Point> &v) {
  sort(ALL(v), [](Point v, Point w) {
    return make_tuple(half(v), 0) <
           make_tuple(half(w), v.cross(w));
  });
}

void polarSortAround(Point o, vector<Point> &v) {
  sort(ALL(v), [](Point v, Point w) {
    return make_tuple(half(v - o), 0) <
           make_tuple(half(w - o), (v - o).cross(w - o));
  });
}

// Si se quiere modificar que el primer angulo del polar
// sort sea el vector v utilizar esta implementacion
Point v = {/* el que sea menos {0,0} */};
bool half(Point p) {
  return v.cross(p) < 0 ||
         (v.cross(p) == 0 && v.dot(p) < 0);
}
```

## 9.9 Polygon

```
// Retorna el area de un triangulo
double areaTriangle(Point a, Point b, Point c) {
  return abs((b - a).cross(c - a)) / 2.0;
}

// Retorna si el punto esta dentro del triangulo
bool pointInTriangle(Point a, Point b, Point c, Point p) {
  T s1 = abs(a.cross(b, c));
  T s2 = abs(p.cross(a, b)) + abs(p.cross(b, c)) +
         abs(p.cross(c, a));
  return s1 == s2;
}

// Retorna el area del poligono
double areaPolygon(vector<Point> p) {
  double area = 0.0;
  int n = SZ(p);
  FOR(i, 0, n) { area += p[i].cross(p[(i + 1) % n]); }
  return abs(area) / 2.0;
}

// Retorna si el poligono es convexo
bool isConvex(vector<Point> p) {
  bool hasPos = false, hasNeg = false;
  for (int i = 0, n = SZ(p); i < n; i++) {
    int o = orient(p[i], p[(i + 1) % n], p[(i + 2) % n]);
    if (o > 0) hasPos = true;
    if (o < 0) hasNeg = true;
  }
  return !(hasPos && hasNeg);
}

// Retorna 1/0/-1 si el punto p esta dentro/sobre/fuera de
// cualquier poligono P concavo/convexo
//
 * Tiempo: O(n)
int inPolygon(vector<Point> poly, Point p) {
  int n = SZ(poly), ans = 0;
  FOR(i, 0, n) {
    Point p1 = poly[i], p2 = poly[(i + 1) % n];
    if (p1.y > p2.y) swap(p1, p2);
    if (onSegment(p1, p2, p)) return 0;
    ans ^= (p1.y <= p.y && p.y < p2.y &&
            p.cross(p1, p2) > 0);
  }
  return ans ? -1 : 1;
}

// Retorna el centroide del poligono
Point polygonCenter(vector<Point>& v) {
  Point res{0, 0};
  double A = 0;
  for (int i = 0, j = SZ(v) - 1; i < SZ(v); j = i++) {
    res = res + (v[i] + v[j]) * v[j].cross(v[i]);
    A += v[j].cross(v[i]);
  }
  return res / A / 3;
}

// Determina si un punto P se encuentra dentro de un
// poligono convexo ordenado en ccw y sin puntos
// colineares (Convex hull) Tiempo O(log n)
bool inPolygonCH(vector<Point>& l, Point p,
                 bool strict = true) {
  int a = 1, b = SZ(l) - 1, r = !strict;
  if (SZ(l) < 3) return r && onSegment(l[0], l.back(), p);
  if (orient(l[0], l[a], l[b]) > 0) swap(a, b);
  if (orient(l[0], l[a], p) >= r ||
      orient(l[0], l[b], p) <= -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (orient(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r;
}

// Retorna los dos puntos con mayor distancia en un
// poligono convexo ordenado en ccw y sin puntos
// colineares (Convex hull) Tiempo O(n)
array<Point, 2> hullDiameter(vector<Point> S) {
  int n = SZ(S), j = n < 2 ? 0 : 1;
  pair<ll, array<Point, 2>> res({0, {S[0], S[0]}});
  FOR(i, 0, j) {
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).sq(), {S[i], S[j]}});
      if ((S[(j + 1) % n] - S[j])
              .cross(S[i + 1] - S[i]) >= 0)
        break;
    }
  }
  return res.second;
}

// Retorna el poligono que se encuentra a la izquierda de
// la linea que va de s a e despues del corte
vector<Point> polygonCut(vector<Point>& poly, Point s,
                         Point e) {
  vector<Point> res;
  FOR(i, 0, SZ(poly)) {
    Point cur = poly[i],
          prev = i ? poly[i - 1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0)) {
      Point p;
      areIntersect(Line(s, e), Line(cur, prev), p);
      res.push_back(p);
    }
    if (side) res.push_back(cur);
  }
  return res;
}
```

## 9.10 Segment

```
// Retorna si el Punto P se encuentra dentro del circulo
// entre A y B
bool inDisk(Point a, Point b, Point p) {
  return (a - p).dot(b - p) <= 0;
}

// Retorna si el punto P se encuentra en el segmento de
// puntos S a E
bool onSegment(Point a, Point b, Point p) {
  return a.cross(b, p) == 0 && inDisk(a, b, p);
}
```

```
// SEGMENTO - SEGMENTO INTERSECCION
bool properInter(Point a, Point b, Point c, Point d,
                 Point& p) {
  double oa = orient(c, d, a), ob = orient(c, d, b),
         oc = orient(a, b, c), od = orient(a, b, d);
  if (oa * ob < 0 && oc * od < 0) {
    p = (a * ob - b * oa) / (ob - oa);
    return true;
  }
  return false;
}
// Si existe un punto de interseccion unico entre los
// segmentos de linea que van de A a B y de C a D, se
// devuelve. Si no existe ningun punto de interseccion, se
// devuelve un vector vacio. Si existen infinitos, se
// devuelve un vector con 2 elementos, que contiene los
// puntos finales del segmento de linea comun.
vector<Point> segInter(Point a, Point b, Point c,
                       Point d) {
  Point p;
  if (properInter(a, b, c, d, p)) return {p};
  set<Point> s;
  if (onSegment(c, d, a)) s.insert(a);
  if (onSegment(c, d, b)) s.insert(b);
  if (onSegment(a, b, c)) s.insert(c);
  if (onSegment(a, b, d)) s.insert(d);
  return {ALL(s)};
}

// SEGMENTO - PUNTO DISTANCIA
double segPoint(Point a, Point b, Point p) {
  if (a != b) {
    Line l(a, b);
    if (l.cmpProj(a, p) && l.cmpProj(p, b))
      return l.dist(p);
  }
  return min((p - a).norm(), (p - b).norm());
}

// SEGMENTO - SEGMENTO DISTANCIA
double segSeg(Point a, Point b, Point c, Point d) {
  Point dummy;
  if (properInter(a, b, c, d, dummy)) return 0;
  return min({segPoint(a, b, c), segPoint(a, b, d),
              segPoint(c, d, a), segPoint(c, d, b)});
}
```

# 10 Extras

## 10.1 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 >> j) & 1)
#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 MX_MSK = 1 << n;
for (int i = 0; i < MX_MSK; i++) {
}
```

## 10.2 Busquedas

```
/**
 * Descripcion: encuentra un valor entre un rango de
 * numeros Busqueda Binaria: divide el intervalo en 2
 * hasta encontrar el valor minimo correcto Busqueda
 * ternaria: divide el intervalo en 3 para buscar el
 * minimo/maximo de una funcion
 *
 * Tiempo: O(log n)
 */

int binary_search(int l, int r) {
  while (r - l > 1) {
    int m = (l + r) / 2;
    if (f(m)) {
      r = m;
    } else {
      l = m;
    }
  }
  return l;
}

double ternary_search(double l, double r) {
  while (r - l > EPS) {
    double m1 = l + (r - l) / 3;
    double m2 = r - (r - l) / 3;
    double f1 = f(m1);
    double f2 = f(m2);
    if (f1 < f2)  // Maximo de f(x)
      l = m1;
    else
      r = m2;
  }
  return f(l);
}
```

## 10.3 Dates

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

## 10.4 Hash Pair

```
/**
 * 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;
```

## 10.5 Random

```
/*
 * Descripcion: Basado en Mersenne Twister genera un
 * numero pseudoaleatorio de 32 bits mucho mas rapido que
 * la funcion rand() de c++.
 * Si necesita hasta 64 bits usar mt19937_64 en su lugar
 */

mt19937 rng(chrono::steady_clock::now()
                .time_since_epoch()
                .count());

// Aplicaciones

// Mezclar un arreglo
shuffle(ALL(a), rng);

// Generar un numero entero en el intervalo [l,r]
uniform_int_distribution<int>(l, r)(rng)
```

## 10.6 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 hexadecimales
cout << hex << 100 << " " << 1000 << " " << 10000 << dec
     << endl;

// Redondea el valor dado al entero mas cercano
```

```
round(5.5);

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

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

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

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

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

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

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

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

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

// 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));
```

## 10.7 int128

```
__int128 read() {
  __int128 x = 0, f = 1;
  char ch = getchar();
  while (ch < '0' || ch > '9') {
    if (ch == '-') f = -1;
    ch = getchar();
  }
  while (ch >= '0' && ch <= '9') {
    x = x * 10 + ch - '0';
    ch = getchar();
  }
  return x * f;
}

void print(__int128 x) {
  if (x < 0) {
    putchar('-');
    x = -x;
  }
  if (x > 9) print(x / 10);
  putchar(x % 10 + '0');
}
```