

Notebook UNosnovatos

Contents

| | |
|--|----|
| 1 C++ | 2 |
| 1.1 C++ plantilla | 2 |
| 1.2 Librerias | 2 |
| 1.3 Bitmask | 2 |
| 1.4 Cosas de strings | 3 |
| 2 Estructuras de Datos | 4 |
| 2.1 Disjoint Set Union | 4 |
| 2.2 Fenwick Tree | 4 |
| 2.3 Segment Tree | 4 |
| 2.4 Persistent ST | 5 |
| 2.5 Distinct Values Queries | 6 |
| 3 Programacion dinamica | 6 |
| 3.1 LIS | 6 |
| 3.2 Bin Packing | 6 |
| 3.3 Algoritmo de Kadane 2D | 7 |
| 3.4 Knuth Clasico | 7 |
| 3.5 Edit Distances | 7 |
| 3.6 Divide Conquer | 8 |
| 3.7 Knuth | 8 |
| 4 Grafos | 8 |
| 4.1 Puentes | 8 |
| 4.2 Puntos de Articulacion | 9 |
| 4.3 Puntos de articulacion y puentes (dirigidos) | 9 |
| 4.4 Algoritmo Kosajaru | 10 |
| 4.5 Tarjan | 10 |
| 4.6 Dijkstra | 10 |
| 4.7 Bellman Ford | 10 |
| 4.8 Floyd Warshall | 11 |
| 4.9 MST Kruskal | 11 |
| 4.10 MST Prim | 11 |
| 4.11 Shortest Path Faster Algorithm | 11 |
| 4.12 Camino mas corto de longitud fija | 12 |
| 5 Flujos | 12 |
| 5.1 Edmonds-Karp | 12 |
| 5.2 Dinic | 13 |

| | |
|--|----|
| 5.3 Maximum Bipartite Matching | 13 |
| 5.4 Minimum cost flow | 14 |
| 6 Matematicas | 15 |
| 6.1 Descomposicion en primos (y mas cosas) | 15 |
| 6.2 Criba Modificada | 15 |
| 6.3 Funcion Totient de Euler | 15 |
| 6.4 Exponenciacion binaria | 16 |
| 6.5 Exponenciacion matricial | 16 |
| 6.6 Fibonacci Matriz | 16 |
| 6.7 GCD y LCM | 16 |
| 6.8 Algoritmo Euclideo Extendido | 16 |
| 6.9 Inverso modular | 16 |
| 6.10 Coeficientes binomiales | 17 |
| 6.11 Logaritmo Discreto | 17 |
| 6.12 Freivalds algorithm | 17 |
| 7 Metodos numericos | 17 |
| 7.1 Ternary Search | 17 |
| 7.2 Regla de Simpson | 18 |
| 8 Strings | 18 |
| 8.1 Funcion Z | 18 |
| 8.2 Funcion Phi | 18 |
| 8.3 Kmp | 18 |
| 8.4 Aho-Corasick | 19 |
| 8.5 Hashing | 19 |
| 8.6 Manacher | 20 |
| 8.7 Minimal-Rotation | 20 |
| 8.8 Rabin-Karp | 21 |
| 8.9 Kmp-Automata | 21 |
| 8.10 Suffix Array Forma 1 | 21 |
| 8.11 Suffix Array Forma 2 | 22 |
| 8.12 Suffix Automata Forma 1 | 23 |
| 8.13 Suffix Automata Forma 2 | 23 |
| 8.14 Longest Common Subsequence | 24 |
| 8.15 Longest Common Substring | 24 |
| 8.16 Lyndon Factorization | 24 |
| 8.17 Cantidad Substring por len | 24 |
| 8.18 Cantidad Substrings | 25 |
| 8.19 Kth-Substring con repeticiones | 25 |
| 8.20 Kth-substring sin repeticiones | 25 |
| 8.21 Primera aparicion patrones | 26 |

| | |
|--|----|
| 8.22 Repetitions | 26 |
| 8.23 Substring mas largo repetido | 26 |
| 9 Geometria | 26 |
| 9.1 Puntos | 26 |
| 9.2 Lineas | 27 |
| 9.3 Vectores | 27 |
| 9.4 Poligonos | 28 |
| 9.5 Convex Hull | 28 |
| 10 Teoría y miscelánea | 29 |
| 10.1 Sumatorias | 29 |
| 10.2 Teoría de Grafos | 29 |
| 10.2.1 Teorema de Euler | 29 |
| 10.2.2 Planaridad de Grafos | 29 |
| 10.3 Teoría de Números | 29 |
| 10.3.1 Ecuaciones Diofánticas Lineales | 29 |
| 10.3.2 Pequeño Teorema de Fermat | 30 |
| 10.3.3 Teorema de Euler | 30 |
| 10.4 Teorema de Pick | 30 |
| 10.5 Combinatoria | 30 |
| 10.5.1 Permutaciones | 30 |
| 10.5.2 Combinaciones | 30 |
| 10.5.3 Permutaciones con Repetición | 30 |
| 10.5.4 Combinaciones con Repetición | 30 |
| 10.5.5 Números de Catalan | 30 |
| 10.6 DP Optimization Theory | 31 |

1 C++

1.1 C++ plantilla

```
#include <bits/stdc++.h>
using namespace std;
#define watch(x) cout<<#x<<"="<<x<<'\n'
#define sz(arr) ((int) arr.size())
#define all(v) v.begin(), v.end()
typedef long long ll;
typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
typedef vector<long long> vll;
typedef pair<ll, ll> pll;
typedef vector<pll> vll;
const int INF = 1e9;
```

```
const ll INFL = 1e18;
const int MOD = 1e9+7;
const double EPS = 1e-9;
int dirx[4] = {0,-1,1,0};
int diry[4] = {-1,0,0,1};
int dr[] = {1, 1, 0, -1, -1, -1, 0, 1};
int dc[] = {0, 1, 1, 1, 0, -1, -1, -1};
const string ABC = "abcdefghijklmnopqrstuvwxyz";
const char ln = '\n';

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout << setprecision(20) << fixed;
    // freopen("file.in", "r", stdin);
    // freopen("file.out", "w", stdout);

    return 0;
}
```

1.2 Librerías

```
// En caso de que no sirva #include <bits/stdc++.h>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <sstream>
#include <fstream>
#include <cassert>
#include <climits>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <deque>
#include <stack>
#include <list>
#include <map>
#include <set>
#include <bitset>
#include <iomanip>
#include <unordered_map>
////
#include <tuple>
#include <random>
#include <chrono>
```

1.3 Bitmask

```
// Todas son O(1)Representacion
int a = 5; // Representacion binaria: 0101
int b = 3; // Representacion binaria: 0011
// Operaciones Principales
int resultado_and = a & b; // 0001 (1 en decimal)
int resultado_or = a | b; // 0111 (7 en decimal)
int resultado_xor = a ^ b; // 0110 (6 en decimal)

int num = 42; // Representacion binaria: 00101010
bitset<8> bits(num); // Crear un objeto bitset a partir
del numero
cout << "Secuencia de bits: " << bits << "\n";
bits.count(); // Cantidad de bits activados
bits.set(3, true); // Establecer el cuarto bit en 1
bits.reset(6); // Establecer el septimo bit en 0

ll S,T;
// Operaciones con bits (/*) por 2 (redondea de forma
automatica)
S=34; // == 100010
S = S<<1; // == S*2 == 68 == 1000100
S = S>>2; // == S/4 == 17 == 10001
S = S>>1; // == S/2 == 8 == 1000

// Encender un bit
S = 34;
S = S|(1<<3); // S = 42 (101010)

// Limpiar o apagar un bit
// ~: Not operacion
S = 42;
S &= ~(1<<1); // S = 40 (101000)

// Comprobar si un bit esta encendido
S = 42;
T = S&(1<<3); // (!= 0): el tercer bit esta encendido

// Invertir el estado de un bit
S = 40;
S ^= (1<<2); // 44 (101100)

// LSB (Primero de la derecha)
S = 40;
T = ((S) & -(S)); // 8 (001000)
__builtin_ctz(T); // nos entrega el indice del LSB

// Encender todos los bits
ll n = 3; // el tamaño del set de bits
S = 0;
S = (1<<n) - 1; // 7 (111)

// n es el tamaño de la mask (Alternativa)
// ll n = 64;
// for (ll subset = 0; subset < (1<<n); ++subset){

// Enumerar todos los posibles subsets de un bitmask
int mask = 18;
for (int subset = mask; subset; subset = (mask & (subset
```

```
-1))) {
    cout << subset << "\n";
}

// otras funciones de c++
__builtin_popcount(32); // 100000 (base 2), only 1 bit is
on
__builtin_popcount(30); // 11110 (base 2), 4 bits are on
__builtin_popcountl((1l<<62)-1l); // 2^62-1 has 62 bits
on (near limit)
__builtin_ctz(32); // 100000 (base 2), 5 trailing zeroes
__builtin_ctz(30); // 11110 (base 2), 1 trailing zero
__builtin_ctzl(1l<<62); // 2^62 has 62 trailing zeroes
```

1.4 Cosas de strings

```
// Funcion para convertir un caracter a un entero
int conv(char ch) {
    return ((ch >= 'a' && ch <= 'z') ? ch-'a' : ch-'A'+26);
}

string s="abc";
cout<<s.substr(1)<<"\n";
cout<<s.substr(0,1)<<"\n";

// El primer parametro es la posicion inicial
s.insert(3, "def");
cout<<s<<"\n";
// El primer parametro es la posicion y el segundo es la
cantidad de caracteres a borrar
s.erase(3,3);
cout<<s<<"\n";
// El primer parametro es la posicion y el segundo es la
cantidad de caracteres a reemplazar
s.replace(0,2, "def");
cout<<s<<"\n";

for(char& c:s){
    c=toupper(c);
}
cout<<s<<"\n";

for(char& c:s){
    c=tolower(c);
}
cout<<s<<"\n";

// De string a entero
s="123";int n;
istringstream(s)>>n;
cout<<n<<"\n";

// De entero a string
n=456;
ostringstream os;
os<<n;
```

```
s=os.str();
cout<<s<<"\n";
```

2 Estructuras de Datos

2.1 Disjoint Set Union

```
struct dsu{
    vi p, size;
    int num_sets;
    int maxSize;

    dsu(int n){
        p.assign(n, 0);
        size.assign(n, 1);
        num_sets = n;
        for (int i = 0; i<n; i++) p[i] = i;
    }

    int find_set(int i) {return (p[i] == i) ? i : (p[i] =
        find_set(p[i]));}

    bool is_same_set(int i, int j) {return find_set(i) ==
        find_set(j);}

    void unionSet(int i, int j){
        if (!is_same_set(i, j)){
            int a = find_set(i), b = find_set(j);
            if (size[a] < size[b])
                swap(a, b);
            p[b] = a;
            size[a] += size[b];
            maxSize = max(size[a], maxSize);
            num_sets--;
        }
    }
};
```

2.2 Fenwick Tree

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

struct fenwick_tree{
    vl ft; int n;
    fenwick_tree(int n): n (n){ft.assign(n+1, 0);}
    ll rsq(int j){
        ll sum = 0;
        for(;j;j -= LSONe(j)) sum += ft[j];
        return sum;
    }
    ll rsq(int i, int j) {return rsq(j) - (i == 1 ? 0 :
        rsq(i-1));}
```

```
void upd(int i, ll v){
    for (; i <= n; i += LSONe(i)) ft[i] += v;
}

};
```

2.3 Segment Tree

```
int nullValue = 0;

struct nodeST{
    nodeST *left, *right;
    int l, r; ll value, lazy, lazyl;

    nodeST(vi &v, int l, int r) : l(l), r(r){
        int m = (l+r)>>1;
        lazy = 0;
        lazyl = 0;
        if (l!=r){
            left = new nodeST(v, l, m);
            right = new nodeST(v, m+1, r);
            value = opt(left->value, right->value);
        }
        else{
            value = v[l];
        }
    }

    ll opt(ll leftValue, ll rightValue){
        return leftValue + rightValue;
    }

    void propagate(){
        if(lazyl){
            value = lazyl * (r-l+1);
            if (l != r){
                left->lazyl = lazyl, right->lazyl = lazyl;
                left->lazy = 0, right->lazy = 0;
            }
            lazyl = 0;
            lazy = 0;
        }
        else{
            value += lazy * (r-l+1);
            if (l != r){
                if(left->lazyl) left->lazyl += lazy;
                else left->lazy += lazy;
                if(right->lazyl) right->lazyl += lazy;
                else right->lazy += lazy;
            }
            lazy = 0;
        }
    }

    ll get(int i, int j){
        propagate();
```

```

        if (l>=i && r<=j) return value;
        if (l>j || r<i) return nullValue;

        return opt(left->get(i, j), right->get(i, j));
    }

    void upd(int i, int j, int nv){
        propagate();
        if (l>j || r<i) return;
        if (l>=i && r<=j){
            lazy += nv;
            propagate();
            // value = nv;
            return;
        }

        left->upd(i, j, nv);
        right->upd(i, j, nv);

        value = opt(left->value, right->value);
    }

    void upd(int k, int nv){
        if (l>k || r<k) return;
        if (l>=k && r<=k){
            value = nv;
            return;
        }

        left->upd(k, nv);
        right->upd(k, nv);

        value = opt(left->value, right->value);
    }

    void updl(int i, int j, int nv){
        propagate();
        if (l>j || r<i) return;
        if (l>=i && r<=j){
            lazy = 0;
            lazy1 = nv;
            propagate();
            return;
        }

        left->updl(i, j, nv);
        right->updl(i, j, nv);

        value = opt(left->value, right->value);
    }
};

```

2.4 Persistent ST

```

const ll nullVal = 0;
ll oper(ll n1, ll n2){

```

```

    return n1 + n2;
}

struct Vertex {
    Vertex *l, *r;
    ll val;

    Vertex(ll num) : l(nullptr), r(nullptr), val(num) {}
    Vertex(Vertex *l, Vertex *r) : l(l), r(r), val(
        nullVal) {
        if (l) val = oper(val, l->val);
        if (r) val = oper(val, r->val);
    }
};

struct perST{
    ll n;
    // rts es donde guardamos las roots nuevas creadas
    vector<Vertex*> rts;

    // Creacion de la root inicial y asignacion de
    // tamaño de la base de PerST
    perST(vl& a): n(a.size()) {
        rts.pb(build(a, 0, n - 1));
    }

    // build del ST (funciona igual que uno normal solo
    // que con punteros)
    Vertex* build(vl& a, ll tl, ll tr) {
        if (tl == tr)
            return new Vertex(a[tl]);
        ll tm = (tl + tr) >> 1;
        return new Vertex(build(a, tl, tm), build(a, tm
            +1, tr));
    }

    // get del ST (funciona igual que uno normal)
    // el valor de tl y tr sirven para saber en que rango
    // nos encontramos
    ll get(Vertex* v, ll tl, ll tr, ll l, ll r) {
        if (l > r)
            return nullVal;
        if (l == tl && tr == r)
            return v->val;
        ll tm = (tl + tr) >> 1;
        return oper(get(v->l, tl, tm, l, min(r, tm)),
            get(v->r, tm+1, tr, max(l, tm+1), r));
    }

    // el upd del perST recorre el arbol reciclando nodos
    // que
    // quedan igual y creando nuevos para los cuales
    // cambia.
    // Retorna el vertice root del nuevo ST
    Vertex* upd(Vertex* v, ll tl, ll tr, ll pos, ll
        newVal) {
        if (tl == tr)

```

```

        return new Vertex(newVal);
    ll tm = (tl + tr) >> 1;
    if (pos <= tm)
        return new Vertex(upd(v->l, tl, tm, pos,
                               newVal), v->r);
    else
        return new Vertex(v->l, upd(v->r, tm+1, tr,
                                     pos, newVal));
}

// simplificaciones de upd y get
// el valor de k es igual a la version en la cual
// trabajaremos
Vertex* upd(ll k, ll pos, ll newVal){
    return upd(rts[k], 0, n - 1, pos, newVal);
}

ll get(ll k, ll a, ll b){
    return get(rts[k], 0, n - 1, a, b);
}
};

```

2.5 Distinct Values Queries

```

// insertar Persistent ST de sumas
int main() {
    ll n, k; cin >> n >> k;
    vl vals(n, 0);
    forx(i, n) cin >> vals[i];

    // creacion del perST
    vl basSt(n, 0);
    perST vers(basSt);

    // Cada ST estara guardando si el i-esimo elemento es
    // una
    // ultima ocurrencia y la idea es crear una nueva
    // version
    // por cada actualizacion de este dato
    map<ll, ll> lastOcur;
    for(int i = 1; i <= n; i++){
        if (!lastOcur[vals[i - 1]]){
            vers.rts.pb(vers.upd(i - 1, i - 1, 1));
            lastOcur[vals[i - 1]] = i;
        } else {
            vers.rts.pb(vers.upd(i - 1, i - 1, 1));
            vers.rts[i] = vers.upd(i, lastOcur[vals[i - 1]] - 1, 0);
            lastOcur[vals[i - 1]] = i;
        }
    }

    // Para hacer la consulta de la cantidad de
    // distintos en un rango basta con hacer una

```

```

// tipica consulta pero en la version de b
while(k--){
    ll a, b; cin >> a >> b;
    a--; b--;
    cout << vers.get(b + 1, a, b) << ln;
}
}

```

3 Programacion dinamica

3.1 LIS

```

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n; cin >> n;
    vl vals(n);
    for (int i = 0; i < n; i++) cin >> vals[i];

    vl copia(vals);
    sort(copia.begin(), copia.end());

    map<ll, ll> dicc;
    for (int i = 0; i < n; i++) if (!dicc.count(copia[i])) dicc[copia[i]] = i;

    vl baseSt(n, 0);
    nodeSt st(baseSt, 0, n - 1);
    ll maxi = 0;
    for (ll pVal:vals) {
        ll op = st.get(0, dicc[pVal] - 1) + 1;
        maxi = max(maxi, op);
        st.act1(dicc[pVal], op);
    }
    cout << maxi << ln;
}

```

3.2 Bin Packing

```

int main() {
    ll n, capacidad;
    cin >> n >> capacidad;
    vl pesos(n, 0);
    forx(i, n) cin >> pesos[i];

    vector<pll> dp((1 << n));
    dp[0] = {1, 0};
    // dp[X] = {#numero de paquetes, peso de min paquete}

    // La idea es probar todos los subet y en cada uno
    // preguntarnos

```

```
// quien es mejor para subirse de ultimo buscando
// minimizar
// primero el numero de paquetes
for (int subset = 1; subset < (1 << n); subset++) {
    dp[subset] = {21, 0};

    for (int iPer = 0; iPer < n; iPer++) {
        if ((subset >> iPer) & 1) {
            pll ant = dp[subset ^ (1 << iPer)];
            ll k = ant.ff;
            ll w = ant.ss;

            if (w + pesos[iPer] > capacidad) {
                k++;
                w = min(pesos[iPer], w);
            } else {
                w += pesos[iPer];
            }

            dp[subset] = min(dp[subset], {k, w});
        }
    }

    cout << dp[(1 << n) - 1].ff << ln;
}
```

3.3 Algoritmo de Kadane 2D

```
int main() {
    ll fil,col;cin>>fil>>col;
    vector<vl> grid(fil,vl(col,0));

    // Algoritmo de Kadane/DP para suma maxima de una matriz
    // 2D en o(n^3)
    for(int i=0;i<fil;i++){
        for(int e=0;e<col;e++){
            ll num;cin>>num;
            if (e>0) grid[i][e]=num+grid[i][e-1];
            else grid[i][e]=num;
        }
    }

    ll maxGlobal = LONG_LONG_MIN;
    for(int l=0;l<col;l++){
        for(int r=l;r<col;r++){
            ll maxLoc=0;
            for(int row=0;row<fil;row++){
                if (l>0) maxLoc+=grid[row][r]-grid[row][l-1];
                else maxLoc+=grid[row][r];
                if (maxLoc<0) maxLoc=0;
                maxGlobal= max(maxGlobal,maxLoc);
            }
        }
    }
}
```

```
}
}
```

3.4 Knuth Clasico

```
const int N = 1010;
const int INF = (int) 1e9;
int v[N], dp[N][N], sum[N], best[N][N];

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    int n;
    while(cin >> n) {
        if(n == 0) break;
        for(int i = 0; i < n; i++) cin >> v[i];

        for(int i = 0; i < n; i++) {
            sum[i+1] = sum[i] + v[i];
        }

        for(int i = 0; i < n; i++) best[i][i] = i;

        for(int len = 2; len <= n; ++len) {
            for(int i = 0; i+len-1 < n; ++i) {
                int j = i+len-1;
                int &ref = dp[i][j];
                ref = INF;
                for(int k = best[i][j-1]; k <= best[i+1][j]; ++k) {
                    if(k < j) {
                        int cur = dp[i][k] + dp[k+1][j];
                        if(cur < ref) {
                            best[i][j] = k;
                            ref = cur;
                        }
                    }
                }
                ref += sum[j+1] - sum[i];
            }
        }

        cout << dp[0][n-1] << '\n';
    }

    return 0;
}
```

3.5 Edit Distances

```
int editDistances(string& wor1, string& wor2) {
    // O(tam1*tam2)
    // minimo de letras que debemos insertar, eliminar o
    // reemplazar
}
```

```

// de wor1 para obtener wor2
ll tam1=wor1.size();
ll tam2=wor2.size();
vector<vl> dp(tam2+1, vl(tam1+1, 0));
for(int i=0; i<=tam1; i++) dp[0][i]=i;
for(int i=0; i<=tam2; i++) dp[i][0]=i;
dp[0][0]=0;
for(int i=1; i<=tam2; i++){
    for(int j=1; j<=tam1; j++){
        ll op1 = min(dp[i-1][j], dp[i][j-1])+1;
        // el minimo entre eliminar o insertar
        ll op2 = dp[i-1][j-1]; // reemplazarlo
        if(wor1[j-1]!=wor2[i-1]) op2++;
        // si el reemplazo tiene efecto o quedo igual
        dp[i][j]=min(op1, op2);
    }
}
return dp[tam2][tam1];
}

```

3.6 Divide Conquer

```

int m, n;
vector<long long> dp_before(n), dp_cur(n);
long long C(int i, int j);
// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int opttr) {
    if (l > r)
        return;

    int mid = (l + r) >> 1;
    pair<long long, int> best = {LLONG_MAX, -1};
    for (int k = optl; k <= min(mid, opttr); k++) {
        best = min(best, {(k ? dp_before[k - 1] : 0) + C(
            k, mid), k});
    }

    dp_cur[mid] = best.first;
    int opt = best.second;

    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, opttr);
}

int solve() {
    for (int i = 0; i < n; i++)
        dp_before[i] = C(0, i);

    for (int i = 1; i < m; i++) {
        compute(0, n - 1, 0, n - 1);
        dp_before = dp_cur;
    }
}

```

```

return dp_before[n - 1];
}

```

3.7 Knuth

```

#Condiciones
#C(b,c) <= C(a,d)
#C(a,c) + C(b,d) <= C(a,d) + C(b,c)
int solve() {
    int N;
    ... // read N and input
    int dp[N][N], opt[N][N];

    auto C = [&](int i, int j) {
        ... // Implement cost function C.
    };

    for (int i = 0; i < N; i++) {
        opt[i][i] = i;
        ... // Initialize dp[i][i] according to the
              problem
    }

    for (int i = N-2; i >= 0; i--) {
        for (int j = i+1; j < N; j++) {
            int mn = INT_MAX;
            int cost = C(i, j);
            for (int k = opt[i][j-1]; k <= min(j-1, opt[i
                +1][j]); k++) {
                if (mn >= dp[i][k] + dp[k+1][j] + cost) {
                    opt[i][j] = k;
                    mn = dp[i][k] + dp[k+1][j] + cost;
                }
            }
            dp[i][j] = mn;
        }
    }

    cout << dp[0][N-1] << endl;
}

```

4 Grafos

4.1 Puentes

```

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

void IS_BRIDGE(int u, int v, vii &puentes){
    puentes.push_back({min(u, v), max(u, v)});
}

```



```

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

void find_bridges(vector<vi> &adj, vii &puentes, int n) {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(adj, puentes, i);
    }
}

```

4.2 Puntos de Articulacion

```

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

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)

```

```

        IS_CUTPOINT(v);
    }
}

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

```

4.3 Puntos de articulacion y puentes (dirigidos)

```

//Puntos de articulacion: son vertices que desconectan el
//grafo
//Puentes: son aristas que desconectan el grafo
//Usar para grafos dirigidos
//O(V+E)
vi dfs_num, dfs_low, dfs_parent, articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;
vector<vii> adj;
void articulationPointAndBridge(int u) {
    dfs_num[u] = dfsNumberCounter++;
    dfs_low[u] = dfs_num[u]; // dfs_low[u] <= dfs_num[u]
    for (auto &[v, w] : adj[u]) {
        if (dfs_num[v] == -1) { // una arista de arbol
            dfs_parent[v] = u;
            if (u == dfsRoot) ++rootChildren; // vaso
            // especial, raiz
            articulationPointAndBridge(v);
            if (dfs_low[v] >= dfs_num[u]) // para puntos
            // de articulacion
                articulation_vertex[u] = 1;
            if (dfs_low[v] > dfs_num[u]) // para puentes
                printf(" (%d, %d) is a bridge\n", u, v);
            dfs_low[u] = min(dfs_low[u], dfs_low[v]); //
        }
        else if (v != dfs_parent[u]) // si es ciclo no
        // trivial
            dfs_low[u] = min(dfs_low[u], dfs_num[v]); //
            // entonces actualizar
    }
}

int main() {
    dfs_num.assign(V, -1); dfs_low.assign(V, 0);
    dfs_parent.assign(V, -1); articulation_vertex.assign(
    V, 0);
    dfsNumberCounter = 0;
    adj.resize(V);
    printf("Bridges:\n");

```

```

for (int u = 0; u < V; ++u)
    if (dfs_num[u] == -1) {
        dfsRoot = u; rootChildren = 0;
        articulationPointAndBridge(u);
        articulation_vertex[dfsRoot] = (rootChildren
                                         > 1); // caso especial
    }
printf("Articulation Points:\n");
for (int u = 0; u < V; ++u)
    if (articulation_vertex[u])
        printf(" Vertex %d\n", u);
}

```

4.4 Algoritmo Kosajaru

```

//Encontrar las componentes fuertemente conexas en un
//grafo dirigido
//Componente fuertemente conexa: es un grupo de nodos en
//el que hay
//un camino dirigido desde cualquier nodo hasta cualquier
//otro nodo dentro del grupo.
void Kosaraju(int u, int pass) {
    dfs_num[u] = 1;
    vii &neighbor = (pass == 1) ? AL[u] : AL_T[u];
    for (auto &[v, w] : neighbor)
        if (dfs_num[v] == UNVISITED)
            Kosaraju(v, pass);
    S.push_back(u);
}

int main() {
    S.clear();
    dfs_num.assign(N, UNVISITED);
    for (int u = 0; u < N; ++u)
        if (dfs_num[u] == UNVISITED)
            Kosaraju(u, 1);
    numSCC = 0;
    dfs_num.assign(N, UNVISITED);
    for (int i = N-1; i >= 0; --i)
        if (dfs_num[S[i]] == UNVISITED)
            ++numSCC, Kosaraju(S[i], 2);
    printf("There are %d SCCs\n", numSCC);
}

```

4.5 Tarjan

```

vi low, num, comp, g[nax];
int scc, timer;
stack<int> st;
void tjn(int u) {
    low[u] = num[u] = timer++; st.push(u); int v;

```

```

for(int v: g[u]) {
    if(num[v]==-1) tjn(v);
    if(comp[v]==-1) low[u] = min(low[u], low[v]);
}
if(low[u]==num[u]) {
    do{ v = st.top(); st.pop(); comp[v]=scc;
    }while(u != v);
    ++scc;
}
}

void callt(int n) {
    timer = scc = 0;
    num = low = comp = vector<int>(n, -1);
    for(int i = 0; i < n; i++) if(num[i]==-1) tjn(i);
}

```

4.6 Dijkstra

```

//Camino mas cortos
//NO USAR CON PESOS NEGATIVOS, usar Bellman Ford o SPFA (
//mas rapido)
// O ((V+E)*log V)
vi dijkstra(vector<vii> &adj, int s, int V) {
    vi dist(V+1, INT_MAX); dist[s] = 0;
    priority_queue<ii, vii, greater<ii>> > pq; pq.push(ii(0, s));
    while(!pq.empty()) {
        ii front = pq.top(); pq.pop();
        int d = front.first, u = front.second;
        if (d > dist[u]) continue;
        for (int j = 0; j < (int)adj[u].size(); j++) {
            ii v = adj[u][j];
            if (dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                pq.push(ii(dist[v.first], v.first));
            }
        }
    }
    return dist;
}

```

4.7 Bellman Ford

```

vi bellman_ford(vector<vii> &adj, int s, int n) {
    vi dist(n, INF); dist[s] = 0;
    for (int i = 0; i < n-1; i++) {
        bool modified = false;
        for (int u = 0; u < n; u++)
            if (dist[u] != INF)
                for (auto &[v, w] : adj[u]) {
                    if (dist[v] >= dist[u] + w) continue;

```

```

        dist[v] = dist[u] + w;
        modified = true;
    }
    if (!modified) break;
}
bool negativeCicle = false;
for (int u = 0; u < n; u++)
    if (dist[u] != INF)
        for (auto &[v, w] : adj[u]){
            if (dist[v] > dist[u] + w) negativeCicle = true;
        }
return dist;
}

```

4.8 Floyd Warshall

```

//Camino minimo entre todos los pares de vertices
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n; cin >> n;
    vector<vi> adjMat(n+1, vi(n+1));
    //Condicion previa: adjMat[i][j] contiene peso de la
    //arista (i, j)
    //o INF si no existe esa arista
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (adjMat[i][k] < INF && adjMat[k][j] <
                    INF)
                    adjMat[i][j] = min(adjMat[i][j],
                        adjMat[i][k] + adjMat[k][j]);
            }
        }
    }
}

```

4.9 MST Kruskal

```

//Arbol de minima expansion
//O(E*log V)
int main() {
    int n, m;
    cin >> n >> m;
    vector<pair<int, ii>> adj; //Los pares son: {peso, {
    //vertice, vecino}}
    for (int i = 0; i < m; i++){
        int x, y, w; cin >> x >> y >> w;
        adj.push_back(make_pair(w, ii(x, y)));
    }
}

```

```

    }
    sort(adj.begin(), adj.end());
    int mst_costo = 0, tomados = 0;
    dsu UF(n);
    for (int i = 0; i < m && tomados < n-1; i++){
        pair<int, ii> front = adj[i];
        if (!UF.is_same_set(front.second.first, front.
            second.second)){
            tomados++;
            mst_costo += front.first;
            UF.unionSet(front.second.first, front.second.
                second);
        }
    }
    cout << mst_costo;
}

```

4.10 MST Prim

```

vector<vii> adj;
vi tomado;
priority_queue<ii> pq;
void process(int u){
    tomado[u] = 1;
    for (auto &[v, w] : adj[u]){
        if (!tomado[v]) pq.emplace(-w, -v);
    }
}
int prim(int v, int n){
    tomado.assign(n, 0);
    process(0);
    int mst_costo = 0, tomados = 0;
    while (!pq.empty()){
        auto [w, u] = pq.top(); pq.pop();
        w = -w; u = -u;
        if (tomado[u]) continue;
        mst_costo += w;
        process(u);
        tomados++;
        if (tomados == n-1) break;
    }
    return mst_costo;
}

```

4.11 Shortest Path Faster Algorithm

```

//Algoritmo mas rapido de ruta minima
//O(V*E) peor caso, O(E) en promedio.
bool spfa(vector<vii> &adj, vector<int> &d, int s, int n)
{
    d.assign(n, INF);
}

```

```

vector<int> cnt(n, 0);
vector<bool> inqueue(n, false);
queue<int> q;

d[s] = 0;
q.push(s);
inqueue[s] = true;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    inqueue[v] = false;

    for (auto edge : adj[v]) {
        int to = edge.first;
        int len = edge.second;

        if (d[v] + len < d[to]) {
            d[to] = d[v] + len;
            if (!inqueue[to]) {
                q.push(to);
                inqueue[to] = true;
                cnt[to]++;
                if (cnt[to] > n)
                    return false; //ciclo negativo
            }
        }
    }
}
return true;
}

```

4.12 Camino mas corto de longitud fija

```

/*
Modificar operacion * de matrix de esta forma:
En la exponenciacion binaria inicializar matrix ans = b
*/
matrix operator * (const matrix &b){
    matrix ans(this->r, b.c, vector<vl>(this->r, vl(b.c,
        INFL)));

    for (int i = 0; i<this->r; i++) {
        for (int k = 0; k<b.r; k++){
            for (int j = 0; j<b.c; j++){
                ans.m[i][j] = min(ans.m[i][j], m[i][k] +
                    b.m[k][j]);
            }
        }
    }
    return ans;
}

int main() {
    int n, m, k; cin >> n >> m >> k;

```

```

vector<vl> adj(n, vl(n, INFL));
for (int i = 0; i<m; i++){
    ll a, b, c; cin >> a >> b >> c; a--; b--;
    adj[a][b] = min(adj[a][b], c);
}

matrix graph(n, n, adj);
graph = pow(graph, k-1);

cout << (graph.m[0][n-1]==INFL ? -1 : graph.m[0][n-1]) << "\n";

return 0;
}

```

5 Flujos

5.1 Edmonds-Karp

```

//O(V * E^2)
ll bfs(vector<vi> &adj, vector<vl> &capacity, int s, int
t, vi& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pll> q;
    q.push({s, INFL});

    while (!q.empty()) {
        int cur = q.front().first;
        ll flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1LL && capacity[cur][
                next]) {
                parent[next] = cur;
                ll new_flow = min(flow, capacity[cur][
                    next]);
                if (next == t)
                    return new_flow;
                q.push({next, new_flow});
            }
        }
    }

    return 0;
}

ll maxflow(vector<vi> &adj, vector<vl> &capacity, int s,
int t, int n) {
    ll flow = 0;
    vi parent(n);
    ll new_flow;

```

```

while ((new_flow = bfs(adj, capacity, s, t, parent)))
{
    flow += new_flow;
    int cur = t;
    while (cur != s) {
        int prev = parent[cur];
        capacity[prev][cur] -= new_flow;
        capacity[cur][prev] += new_flow;
        cur = prev;
    }
}
return flow;
}

```

5.2 Dinic

```

//O(V^2 * E)
//En redes unitarias: O(E * sqrt(V))
struct FlowEdge {
    int v, u;
    ll cap, flow = 0;
    FlowEdge(int v, int u, ll cap) : v(v), u(u), cap(cap)
    {}
};

struct Dinic {
    const ll flow_inf = INFL;
    vector<FlowEdge> edges;
    vector<vi> adj;
    int n, m = 0;
    int s, t;
    vi level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, ll cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)

```

```

                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    ll dfs(int v, ll pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size();
             cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap -
                edges[id].flow < 1)
                continue;
            ll tr = dfs(u, min(pushed, edges[id].cap -
                edges[id].flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    ll flow() {
        ll f = 0;
        while (true) {
            fill(all(level), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(all(ptr), 0);
            while (ll pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};

```

5.3 Maximum Bipartite Matching

```

int main() {

```

```

//n: numero de grupo 1, m: numero de grupo 2, k:
//posibles conexiones
int n, m, k; cin >> n >> m >> k;
Dinic graph(n+m+2, 0, n+m+1);
//nodo inicial ficticio "0" que se dirige a todos los
//del grupo 1
for (int i = 1; i<=n; i++) graph.add_edge(0, i, 1LL);
//nodo final ficticio "n+m+1" al que se dirigen todos
//los del grupo 2
for (int i = 1; i<=m; i++) graph.add_edge(n+i, n+m+1,
1LL);

//anadiendo las posibles conexiones al grafo
for (int i = 0; i<k; i++){
    int a, b; cin >> a >> b;
    graph.add_edge(a, n+b, 1LL);
}

//numero de emparejamientos realizados
cout << graph.flow() << ln;

//emparejamientos realizados
for (FlowEdge edge : graph.edges){
    if (edge.v != 0 && edge.u != n+m+1 && edge.flow >
0){
        cout << edge.v << " " << edge.u - n << ln;
    }
}

return 0;
}

```

5.4 Minimum cost flow

```

struct Edge{
    ll from, to, capacity, cost;
    Edge(ll from, ll to, ll capacity, ll cost) : from(
from), to(to), capacity(capacity), cost(cost) {}
};

vector<vl> adj, cost, capacity;

void shortest_paths(int n, int v0, vl &d, vector<ll> &p)
{
    d.assign(n, INFL);
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<ll> q;
    q.push(v0);
    p.assign(n, -1);

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

```

```

    inq[u] = false;
    for (int v : adj[u]) {
        if (capacity[u][v] > 0 && d[v] > d[u] + cost[
u][v]) {
            d[v] = d[u] + cost[u][v];
            p[v] = u;
            if (!inq[v]) {
                inq[v] = true;
                q.push(v);
            }
        }
    }
}

ll min_cost_flow(int N, vector<Edge> &edges, ll K, int s,
int t) {
    adj.assign(N, vl());
    cost.assign(N, vl(N, 0));
    capacity.assign(N, vl(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }

    ll flow = 0;
    ll cost = 0;
    vl d, p;
    while (flow < K) {
        shortest_paths(N, s, d, p);
        if (d[t] == INFL)
            break;

        // find max flow on that path
        ll f = K - flow;
        int cur = t;
        while (cur != s) {
            f = min(f, capacity[p[cur]][cur]);
            cur = p[cur];
        }

        // apply flow
        flow += f;
        cost += f * d[t];
        cur = t;
        while (cur != s) {
            capacity[p[cur]][cur] -= f;
            capacity[cur][p[cur]] += f;
            cur = p[cur];
        }
    }

    if (flow < K) return -1;
    else return cost;
}

```

6 Matematicas

6.1 Descomposicion en primos (y mas cosas)

```

ll _sieve_size;
bitset<10000010> bs;
vl p;
void sieve(ll upperbound) {
    _sieve_size = upperbound+1;
    bs.set();
    bs[0] = bs[1] = 0;
    for (ll i = 2; i < _sieve_size; ++i) if (bs[i]) {
        for (ll j = i*i; j < _sieve_size; j += i) bs[j] = 0;
        p.push_back(i);
    }
}
// O( sqrt(N) / log(sqrt(N)) )
vector<int> primeFactors(long long N) {
    vector<int> factors;
    for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i)
        while (N%p[i] == 0) { //Hallado un primo
            para N
            N /= p[i]; //Dividir N
            factors.push_back(p[i]);
        }
    if (N != 1) factors.push_back(N); //El N restante es primo
    return factors;
}
int main() {
    sieve(10000000);
}

//Variantes del algoritmo
//Contar el numero de divisores de N
int numDiv(long long N) {
    int ans = 1; //Empezar con ans = 1
    for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
        int power = 0; //Contar la potencia
        while (N%p[i] == 0) { N /= p[i]; ++power; }
        ans *= power+1; //Seguir la formula
    }
    if (N != 1) return 2 * ans; //Ultimo factor = N^1
    else return ans;
}

//Suma de los divisores de N
//N = a^i * b^i * ... * c^k => N = (a^(i+1) - 1) / (a-1)
+ ...

```

```

long long sumDiv(long long N) {
    long long ans = 1; // empezar con ans = 1
    for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
        long long multiplier = p[i], total = 1;
        while (N%p[i] == 0) {
            N /= p[i];
            total += multiplier;
            multiplier *= p[i];
        }
        ans *= total; // total para este
        factor primo
    }
    if (N != 1) ans *= (N+1); //N^1 + N^0
    return ans;
}

```

6.2 Criba Modificada

```

//Criba modificada
/*
Si hay que determinar el numero de factores primos para
muchos (o un rango) de enteros.
La mejor solucion es el algoritmo de criba modificada O(N log log N)
*/
int numDiffPFarr[MAX_N+10] = {0}; // e.g., MAX_N = 10^7
for (int i = 2; i <= MAX_N; ++i)
    if (numDiffPFarr[i] == 0) // i is a prime number
        for (int j = i; j <= MAX_N; j += i)
            ++numDiffPFarr[j]; // j is a multiple of i

//Similar para EulerPhi
int EulerPhi[MAX_N+10];
for (int i = 1; i <= MAX_N; ++i) EulerPhi[i] = i;
for (int i = 2; i <= MAX_N; ++i)
    if (EulerPhi[i] == i) // i is a prime number
        for (int j = i; j <= MAX_N; j += i)
            EulerPhi[j] = (EulerPhi[j]/i) * (i-1);

```

6.3 Funcion Totient de Euler

```

//EulerPhi(N): contar el numero de enteros positivos < N
que son primos relativos a N.
//El vector p es el que genera la criba de eratostenes
//Phi(N) = N * productoria(1 - (1/pi))
ll EulerPhi(ll N) {
    ll ans = N; // Empezar con ans = N
    for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
        if (N%p[i] == 0) ans -= ans/p[i]; //contar factores
    }
}

```

```

    while (N%p[i] == 0) N /= p[i];    //primos unicos
}
if (N != 1) ans -= ans/N; // ultimo factor
return ans;
}

```

6.4 Exponenciación binaria

```

ll bincpow(ll b, ll n, ll m) {
    b %= m;
    ll res = 1;
    while (n > 0) {
        if (n & 1)
            res = res * b % m;
        b = b * b % m;
        n >>= 1;
    }
    return res % m;
}

```

6.5 Exponenciación matricial

```

struct matrix {
    int r, c; vector<vl> m;
    matrix(int r, int c, const vector<vl> &m) : r(r), c(c), m(m) {}

    matrix operator * (const matrix &b) {
        matrix ans(this->r, b.c, vector<vl>(this->r, vl(b.c, 0)));

        for (int i = 0; i < this->r; i++) {
            for (int k = 0; k < b.r; k++) {
                if (m[i][k] == 0) continue;
                for (int j = 0; j < b.c; j++) {
                    ans.m[i][j] += mod(m[i][k], MOD) *
                        mod(b.m[k][j], MOD);
                    ans.m[i][j] = mod(ans.m[i][j], MOD);
                }
            }
        }
        return ans;
    }
};

matrix pow(matrix &b, ll p) {
    matrix ans(b.r, b.c, vector<vl>(b.r, vl(b.c, 0)));
    for (int i = 0; i < b.r; i++) ans.m[i][i] = 1;
    while (p) {
        if (p & 1)
            ans = ans * b;
        b = b * b;
        p >>= 1;
    }
}

```

```

    p >>= 1;
}
return ans;
}

```

6.6 Fibonacci Matriz

```

/*
[1 1] p   [fib(p+1) fib(p)]
[1 0]   = [fib(p)   fib(p-1)]
*/
vector<vl> matriz = {{1, 1}, {1, 0}};
matrix m(2, 2, matriz);

ll n; cin >> n;

cout << pow(m, n).m[0][1] << "\n";

```

6.7 GCD y LCM

```

//O(log10 n) n == max(a, b)
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
int lcm(int a, int b) { return a / gcd(a, b) * b; }
//gcd(a, b, c) = gcd(a, gcd(b, c))

```

6.8 Algoritmo Euclideo Extendido

```

// O(log(min(a, b)))
ll extEuclid(ll a, ll b, ll &x, ll &y) {
    ll xx = y = 0;
    ll yy = x = 1;
    while (b) {
        ll q = a/b;
        ll t = b; b = a%b; a = t;
        t = xx; xx = x-q*xx; x = t;
        t = yy; yy = y-q*yy; y = t;
    }
    return a; //Devuelve gcd(a, b)
}

```

6.9 Inverso modular

```

ll mod(ll a, ll m) {
    return ((a%m) + m) % m;
}

ll modInverse(ll b, ll m) {
    ll x, y;

```



```

ll d = extEuclid(b, m, x, y); //obtiene b*x + m*y ==
d
if (d != 1) return -1; //indica error
// b*x + m*y == 1, ahora aplicamos (mod m) para
// obtener b*x == 1 (mod m)
return mod(x, m);
}
// Otra forma
// O(log MOD)
ll inv(ll a){
    return binpow(a, MOD-2, MOD);
}

```

6.10 Coeficientes binomiales

```

const int MAX_N = 100010; // MOD > MAX_N
// O(log MOD)
ll inv(ll a){
    return binpow(a, MOD-2, MOD);
}

ll fact[MAX_N];
// O(log MOD)
ll C(int n, int k){
    if (n < k) return 0;
    return ((fact[n] * inv(fact[k])) % MOD) * inv(fact[n
        -k]) % MOD;
}

int main() {
    fact[0] = 1;
    for (int i = 1; i < MAX_N; i++){
        fact[i] = (fact[i-1]*i) % MOD;
    }
    cout << C(100000, 50000) << "\n";
    return 0;
}

```

6.11 Logaritmo Discreto

```

// Returns minimum x for which a ^ x % m = b % m.
int solve(int a, int b, int m) {
    a %= m, b %= m;
    int k = 1, add = 0, g;
    while ((g = gcd(a, m)) > 1) {
        if (b == k)
            return add;
        if (b % g)
            return -1;
        b /= g, m /= g, ++add;
        k = (k * 111 * a / g) % m;
    }
}

```

```

}
int n = sqrt(m) + 1;
int an = 1;
for (int i = 0; i < n; ++i)
    an = (an * 111 * a) % m;

unordered_map<int, int> vals;
for (int q = 0, cur = b; q <= n; ++q) {
    vals[cur] = q;
    cur = (cur * 111 * a) % m;
}

for (int p = 1, cur = k; p <= n; ++p) {
    cur = (cur * 111 * an) % m;
    if (vals.count(cur)) {
        int ans = n * p - vals[cur] + add;
        return ans;
    }
}
return -1;
}

```

6.12 Freivalds algorithm

```

mt19937 rnd(chrono::steady_clock::now().time_since_epoch
().count());
// check if two n*n matrix a*b=c within complexity (
// iteration*n^2)
// probability of error 2^(-iteration)
int Freivalds(matrix &a, matrix &b, matrix &c) {
    int n = a.r, iteration = 20;
    matrix zero(n, 1), r(n, 1);
    while (iteration-- > 0) {
        for (int i = 0; i < n; i++) r.m[i][0] = rnd() % 2;
        matrix ans = (a * (b * r)) - (c * r);
        if (ans.m != zero.m) return 0;
    }
    return 1;
}

```

7 Metodos numericos

7.1 Ternary Search

```

double f(double x) {
    return x*x;
}

// O(log((r-l)/eps))
double ternary_search(double l, double r) {
    double eps=1e-9; // precision
    while (r-l > eps) {

```

```

double m1=1+(r-1)/3;
double m2=r-(r-1)/3;
if (f(m1)<f(m2)) l=m1;
else r=m2;
} return max(f(l),f(r)); // El maximo de la funcion en
    el intervalo [l,r]
}

```

7.2 Regla de Simpson

```

double f(double x){
    return x*x;
}

const int N = 1000 * 1000; // number of steps (already
    multiplied by 2)
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;
}

```

8 Strings

8.1 Funcion Z

```

// Funcion z O(s)
vi z_function(string s) {
    int n=len(s),l=0,r=0;
    vi z(n);
    for(int i=1;i<n;i++){
        if(i<r) z[i]=min(r-i, z[i-1]);
        while(i+z[i]<n && s[z[i]]==s[i+z[i]]) z[i]++;
        if(i+z[i]>r){
            l=i;
            r=i+z[i];
        }
    }
    return z;
}

```

8.2 Funcion Phi

```

// Funcion phi O(s)
vi prefix_function(string s){

```

```

int n=len(s);
vi pi(n);
for(int i=1;i<n;i++){
    int j=pi[i-1];
    while(j>0 && s[i]!=s[j]) j=pi[j-1];
    if (s[i]==s[j]) j++;
    pi[i]=j;
}
return pi;
}

int main() {
    vi pi=prefix_function(string); // Obtener phi
    //Lo siguiente es para saber cuantas veces aparece cada
        prefijo O(n)
    int n=len(s);
    vi ans(n+1);
    for(int i=0;i<n;i++) ans[pi[i]]++;
    for(int i=n-1;i>0;i--) ans[pi[i-1]]+=ans[i];
    for(int i=0;i<n;i++) ans[i]++;
    for(int i=0;i<n;i++) cout<<"El prefijo de tamaño "<<i<<"
        aparece "<<ans[i]<<" veces\n";
    return 0;
}

```

8.3 Kmp

```

// Implementar primero prefix_function
// O(t+p)
int matches=0;
void kmp(string &t, string &p){
    vi phi=prefix_function(p);
    for(int i=0,j=0;i<sz(t);i++){
        while(j>0 && t[i]!=p[j]) j=phi[j-1];
        if(t[i]==p[j]) j++;
        if(j==sz(p)){
            cout<<i-j+1<<" "; // Posicion de la ocurrencia
            matches++;
            j=phi[j-1];
        }
    }
}

// Devuelve el arreglo de matches sin implementar
    prefix_function
const int MAX=2e5+9;
int pi[MAX];
// Pasar el arreglo int d con tamaño len(t)
void kmp_vi(string& p, string& t, int *d){
    pi[0]=0;int m=len(p),n=len(t);
    for(int i=1,k=0;i<m;i++){
        while(k>0 && p[k]!=p[i]) k=pi[k-1];
        if(p[i]==p[k]) k++;
        pi[i]=k;
    }
}

```

```

    }
    for(int i=0,k=0;i<n;i++){
        while(k>0 && p[k]!=t[i])k=pi[k-1];
        if(t[i]==p[k])k++;
        d[i]=k;
        if(k==m)k=pi[k-1];
    }
}

```

8.4 Aho-Corasick

```

// Usar el aho-corasick para buscar multiples patrones en
// un texto
const static int N=1e5; // Maximo de strings
const static int alpha = 26; // Tamano del alfabeto
int trie[N][alpha], fail[N], nodes, end_word[N], cnt_word
[N], fail_out[N];
inline int conv(char ch) { // Funcion para indexar el
    alfabeto
    return ((ch >= 'a' && ch <= 'z') ? ch-'a' : ch-'A'+26);
}

// Para cada string, se agrega al trie O(s), peor caso O(
s*n) n=numero de strings
void add(string &s, int i) {
    int act=0;
    for(char c:s) {
        int x=conv(c);
        if(!trie[act][x]) trie[act][x]=++nodes;
        act=trie[act][x];
    }
    ++cnt_word[act];
    end_word[act]=i;
}

// Se crea el trie con bfs O(N*log(ALPHA))
void build() {
    queue<int> q;q.push(0);
    while(sz(q)){
        int u=q.front();q.pop();
        for(int i=0;i<alpha;++i) {
            int v=trie[u][i];
            if(!v)trie[u][i]=trie[fail[u]][i];
            else q.push(v);
            if(!u || !v)continue;
            fail[v]=trie[ fail[u] ][i];
            fail_out[v]=end_word[fail[v]]?fail[v]:fail_out[fail
                [v]];
            cnt_word[v]+=cnt_word[fail[v]];
        }
    }
}

// O(n+m) donde n=tamano del texto y m=cantidad de

```

```

    strings
vs strings;
void searchPatterns(string &t) {
    int act=0, n=len(t);
    for(int i=0;i<n;++i) {
        int x=conv(t[i]);
        act=trie[act][x];
        int temp=act;
        while(temp) {
            if(end_word[temp]) cout<<"En la posicion "<<i<<" se
                encontro la palabra "<<strings[end_word[temp
                    ]-1]<<"\n";
            temp=fail_out[temp];
        }
    }
}

// Por si solo se necesita saber si esta O(s)
void solve(int index, string s){
    int act=0;
    bool pass=false;
    for(auto c:s){
        int x=c-'a';
        while(act && !trie[act][x])act=fail[act];
        act=trie[act][x];
        pass|=end_word[act]<index;
    }
    cout<<(pass?"YES":"NO")<<"\n";
}

int main() {
    add(string, i+1); //Anadir todos los patrones
    build(); // Construir el trie
    searchPatterns(texto); // Buscar todos los patrones en el
        texto
    return 0;
}

```

8.5 Hashing

```

// se recomienda usar m = pow(2,64) porque
// m=1e9+9 no es suficiente para la multiplicacion de dos
// 64-bit integers
// Porque la probabilidad de colisiones es 1/m = 10^-9
// y si son 10^6 strings que hay que comparar con este
// entonces 1/m = 10^-3
// y comparamos unos con otros entonces 1/m = 1, si o si
// va a haber algun fallo
// Una solucion sencilla es hacer dos hash (hash1, hash2)
// con p diferentes para tener una probabilidad de
// 1/10^18
// y si comparamos unos con otros entonces 1/m = 10^-6

```

```
// Dos strings con mismo hash no necesariamente son
// iguales
// Pero si tienen distinto hash, entonces son distintos
ll compute_hash(string const& s) { // O(n)
    const int p = 31; // 51 si se usan mayusculas tambien
    // Importante que m sea un numero primo
    const int m = 1e9 + 9;
    ll hash_value=0;
    ll p_pow=1;
    for (char c:s) {
        hash_value=(hash_value+(c-'a'+1)*p_pow)%m;
        p_pow=(p_pow*p)%m;
    }
    return hash_value;
}

// O(n(m+logn)) n=cantidad de strings, m=tamano del
// string mas largo
vector<vi> group_identical_strings(vs const& s) {
    int n=s.size();
    vector<pair<ll, int>> hashes(n);
    for(int i=0;i<n;i++)
        hashes[i]={compute_hash(s[i]),i};
    sort(all(hashes));
    vector<vi> groups;
    for(int i=0;i<n;i++) {
        // Si es el primero o si el hash es distinto al
        // anterior entonces es un nuevo grupo
        if(i==0 || hashes[i].first!=hashes[i-1].first)groups.
            emplace_back();
        groups.back().push_back(hashes[i].second);
    }
    return groups;
}
```

8.6 Manacher

```
// a b c b a a b
// 1 1 5 1 1 1 1 f = 0 impar
// 0 0 0 0 0 4 0 f = 1 par (raiz, izq, der)
void manacher(string &s, int f, vi &d){ // O(s)
    int l=0, r=-1, n=len(s);
    d.assign(n,0);
    for(int i=0;i<n;++i){
        int k=(i>r?(1-f):min(d[l+r-i+f], r-i+f))+f;
        while(i+k-f<n && i-k>=0 && s[i+k-f]==s[i-k]))++k;
        d[i]=k-f;--k;
        if(i+k-f>r)l=i-k,r=i+k-f;
    }
    for(int i=0;i<n;++i)d[i]=(d[i]-1+f)*2+1-f;
}

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

```
vi manacher_odd, manacher_even;
manacher(s, 0, manacher_odd);
manacher(s, 1, manacher_even);
for(int i=0;i<len(s);++i){
    if(manacher_odd[i]==0 || manacher_odd[i]==1)continue;
    cout<<s.substr(i-manacher_odd[i]/2, manacher_odd[i])<<"
        ";
}
cout<<"\n";
for(int i=0;i<len(s);++i){
    if(manacher_even[i]==0)continue;
    cout<<s.substr(i-manacher_even[i]/2, manacher_even[i])
        <<" ";
}
cout<<"\n";
}
```

8.7 Minimal-Rotation

```
// Encuentra la rotacion lexicograficamente menor de un
// string O(n)
int minimal_rotation(string& t) {
    int i=0, j=1, k=0, n=len(t), x, y;
    while(i<n && j<n && k<n) {
        x=i+k; y=j+k;
        if(x>=n)x-=n;
        if(y>=n)y-=n;
        if(t[x]==t[y])++k;
        else if(t[x]>t[y]){
            i=j+1>i+k+1?j+1:i+k+1;
            swap(i, j);
            k=0;
        }else{
            j=i+1>j+k+1?i+1:j+k+1;
            k=0;
        }
    }
    return i;
}

// Son lo mismo
string min_cyclic_string(string s) {
    s+=s;
    int n=len(s), i=0, ans=0;
    while(i<n/2){
        ans=i;
        int j=i+1, k=i;
        while(j<n && s[k]<=s[j]){
            if(s[k]<s[j])k=i;
            else k++;
            j++;
        }
        while(i<=k)
            i+=j-k;
```

```

    }
    return s.substr(ans, n/2);
}

```

8.8 Rabin-Karp

```

// O(s+t)
// Dado un patron s y un texto t, devuelve un vector con
// las posiciones de las ocurrencias de s en t
vi rabin_karp(string const& s, string const& t) {
    // Ojo con p y m
    const int p=31;
    const int m=1e9+9;
    int S=s.size(), T=t.size();
    vl p_pow(max(S, T));
    p_pow[0]=1;
    // Precalculo de potencias de p
    for(int i=1; i<sz(p_pow); i++) p_pow[i]=(p_pow[i-1]*p)%m;
    vl h(T+1, 0);
    // Precalculo de hashes de prefijos de t
    for(int i=0; i<T; i++) h[i+1]=(h[i]+(t[i]-'a'+1)*p_pow[i])%m;
    ll h_s=0;
    // Hash de s
    for(int i=0; i<S; i++) h_s=(h_s+(s[i]-'a'+1)*p_pow[i])%m;
    vi occurrences;
    for(int i=0; i+S-1<T; i++) {
        ll cur_h=(h[i+S]-h[i])%m;
        if(cur_h==h_s*p_pow[i]%m) occurrences.push_back(i);
    }
    return occurrences;
}

```

8.9 Kmp-Automata

```

const int N = 1e5; // Tamano del automata
const int ALPHA = 255; // Tamano del alfabeto ASCII
int automata[N][ALPHA]; // Tabla de transicion del automata

// O(s*ALPHA)
void kmp_automata(string& s) {
    automata[0][s[0]] = 1;
    for(int i = 1, j = 0; i <= len(s); ++i) {
        // Copiar la fila anterior
        for(int k = 0; k < ALPHA; ++k) automata[i][k] = automata[j][k];
        // Actualizar la entrada correspondiente al caracter actual
        if(i<len(s)) {
            automata[i][s[i]] = i+1;
            j=automata[j][s[i]];
        }
    }
}

```

8.10 Suffix Array Forma 1

```

// O(nlogn)
vi sort_cyclic_shifts(string const& s) {
    int n=len(s);
    const int alphabet=256;
    vi p(n), c(n), cnt(max(alphabet,n), 0);
    for(int i=0; i<n; i++) cnt[s[i]]++;
    for(int i=1; i<alphabet; i++) cnt[i] += cnt[i-1];
    for(int i=0; i<n; i++) p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes=1;
    for(int i=1; i<n; i++) {
        if(s[p[i]] != s[p[i-1]]) classes++;
        c[p[i]] = classes-1;
    }
    vi pn(n), cn(n);
    for(int h=0; (1<<h)<n; ++h) {
        for(int i=0; i<n; i++) {
            pn[i] = p[i] - (1<<h);
            if(pn[i]<0) pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin()+classes, 0);
        for(int i=0; i<n; i++) cnt[c[pn[i]]]++;
        for(int i=1; i<classes; i++) cnt[i] += cnt[i-1];
        for(int i=n-1; i>=0; i--) p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes=1;
        for(int i = 1; i < n; i++) {
            ii cur={c[p[i]], c[(p[i]+(1<<h))%n]};
            ii prev={c[p[i-1]], c[(p[i-1]+(1<<h))%n]};
            if(cur!=prev) ++classes;
            cn[p[i]] = classes-1;
        }
        c.swap(cn);
    }
    return p;
}

// O(nlogn)
vi suffix_array(string s) {
    s+="$";
    vi sorted_shifts=sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}

// O(n)
// Longest common prefix
vi lcp_construction(string const& s, vi const& p) {

```

```

int n=len(s);
vi rank(n,0);
for(int i=0;i<n;i++) rank[p[i]]=i;
int k=0;
vi lcp(n-1,0);
for(int i=0;i<n;i++){
    if(rank[i]==n-1){
        k=0;continue;
    }
    int j=p[rank[i]+1];
    while(i+k<n && j+k<n && s[i+k]==s[j+k])k++;
    lcp[rank[i]] = k;
    if(k)k--;
}
return lcp;
}

int main() {
string s;cin>>s;int n=len(s);
vi sa=suffix_array(s);
cout<<"Desde el index, el suffix array\n";
for(int i=0;i<n;i++) cout<<sa[i]<<" ";
cout<<"\nVa comparando de 2 en 2 y muestra el lcp:\n";
vi lcp=lcp_construction(s,sa);
for(int i=0;i<n-1;i++) cout<<lcp[i]<<" ";
}

```

8.11 Suffix Array Forma 2

```

// Construccion O(nlogn)
// Usar cuando queremos ver patron por patron, es mejor
// que el aho-corasick
struct SuffixArray{
    char MIN_CHAR='$',';
    int ALPHA=256;
    int n;
    string s;
    vi pos, rnk, lcp;
    SuffixArray(const string &s):(len(_s) + 1), s(_s),
        pos(n), rnk(n), lcp(n-1){
        s+=MIN_CHAR;
        buildSA();
        buildLCP();
    }

    void buildSA(){
        vi cnt(max(ALPHA, n));
        for(int i=0;i<n;i++) cnt[s[i]]++;
        for(int i=1;i<ALPHA;i++) cnt[i]+=cnt[i-1];
        for(int i=n-1;i>=0;i--) pos[--cnt[s[i]]]=i;
        for(int i=1;i<n;i++) rnk[pos[i]]=rnk[pos[i-1]]+(s[pos[i]]!=s[pos[i-1]]);
        for(int k=0;(1<<k)<n;k++) {
            vi npos(n), nrnk(n), ncnt(n);

```

```

            for(int i=0;i<n;i++) pos[i]=(pos[i]-(1<<k)+n)%n;
            for(int i=0;i<n;i++) ncnt[rnk[i]]++;
            for(int i=1;i<n;i++) ncnt[i]+=ncnt[i-1];
            for(int i=n-1;i>=0;i--) npos[--ncnt[rnk[pos[i]]]]=pos[i];
            for(int i=1;i<n;i++){
                ii cur={rnk[npos[i]], rnk[(npos[i]+(1<<k))%n]};
                ii pre={rnk[npos[i-1]], rnk[(npos[i-1]+(1<<k))%n]};
                nrnk[npos[i]]=nrnk[npos[i-1]]+(cur!=pre);
            }
            pos=npos;rnk=nrnk;
        }
    }

    void buildLCP(){
        for(int i=0,k=0;i<n-1;i++,k=max(k-1,0)){
            int j=pos[rnk[i]-1];
            while(s[i+k]==s[j+k])k++;
            lcp[rnk[i]-1]=k;
        }
    }

    // O(logn+t)
    // Encuentra cuantas veces aparece t en s
    int cntMatching(const string &t){
        int m=len(t);
        if(m>n) return 0;
        int lo,hi,lb,ub;
        lo=0,hi=n-1;
        while(lo<hi){
            int mid=(lo+hi)/2;
            if(s.substr(pos[mid],m)>=t)hi=mid;
            else lo=mid+1;
        }
        lb=lo;lo=0,hi=n-1;
        while(lo<hi){
            int mid=(lo+hi+1)/2;
            if(s.substr(pos[mid],m)<=t)lo=mid;
            else hi=mid-1;
        }
        ub=lo;
        return s.substr(pos[lb], m)==t?ub-lb+1:0;
    }
};

int main() {
string s;cin>>s;
int n;cin>>n;
SuffixArray sa(s);
for(int i=0;i<n;i++){
    string t;cin>>t;
    cout<<sa.cntMatching(t)<<"\n";
}
}

```

8.12 Suffix Automata Forma 1

```
// La creacion del automata es O(n)
struct state {
    int len, link;
    map<char, int> next;
};

const int N=100000;
state st[N*2];
int sz, last;

void sa_init() {
    st[0].len=0;
    st[0].link=-1;
    sz++;
    last=0;
}

void sa_extend(char c) {
    int act=sz++;
    st[act].len=st[last].len+1;
    int p=last;
    while(p!=-1 && !st[p].next.count(c)) {
        st[p].next[c]=act;
        p=st[p].link;
    }
    if(p==-1) {
        st[act].link=0;
    } else {
        int q=st[p].next[c];
        if(st[p].len+1==st[q].len) {
            st[act].link=q;
        } else {
            int clone=sz++;
            st[clone].len=st[p].len+1;
            st[clone].next=st[q].next;
            st[clone].link=st[q].link;
            while(p!=-1 && st[p].next[c]==q) {
                st[p].next[c]=clone;
                p=st[p].link;
            }
            st[q].link=st[act].link=clone;
        }
    }
    last=act;
}
```

8.13 Suffix Automata Forma 2

```
// O(n) construccion, O(n) memoria
struct SuffixAutomaton {
    int last;
```

```
vi len, link, firstPos;
vl cnt;
vector<array<int, 2>> order;
vector<array<int, ALPHA>> nxt;
SuffixAutomaton(): last(0), len(1), link(1, -1), firstPos(1)
    , cnt(1), nxt(1) {}
SuffixAutomaton(const string &s): SuffixAutomaton() {
    for (char c: s)
        extend(c);
}

int getIndex(char c) {
    return c-MIN_CHAR;
}

void extend(char c) {
    int act=sz(len), i=getIndex(c), p=last;
    len.push_back(len[last]+1);
    link.emplace_back();
    cnt.push_back(1);
    firstPos.emplace_back(len[last]+1);
    order.push_back({len[act], act});
    nxt.emplace_back();
    while(p != -1 && !nxt[p][i]) {
        nxt[p][i]=act;
        p=link[p];
    }
    if(p != -1) {
        int q=nxt[p][i];
        if(len[p]+1==len[q]) {
            link[act]=q;
        } else {
            int clone=sz(len);
            len.push_back(len[p]+1);
            link.push_back(link[q]);
            firstPos.push_back(firstPos[q]);
            cnt.push_back(0);
            order.push_back({len[clone], clone});
            nxt.push_back(nxt[q]);
            while(p != -1 && nxt[p][i]==q) {
                nxt[p][i]=clone;
                p=link[p];
            }
            link[q]=link[act]=clone;
        }
    }
    last=act;
}

int main() {
    SuffixAutomaton sa(string);
    return 0;
}
```

8.14 Longest Common Subsequence

```

const int nMax = 1005;
int dp[nMax][nMax];
// Longest Common Subsequence O(n*m) (devuelve el tamaño)
int lcs(const string &s, const string &t){
    int n=len(s), m=len(t);
    for(int i=1; i<=n; i++){
        for(int j=1; j<=m; j++){
            dp[i][j]=max(dp[i-1][j], dp[i][j-1]);
            if(s[i-1]==t[j-1]) dp[i][j]=max(dp[i][j], dp[i-1][j-1] + 1);
        }
    }
    return dp[n][m];
}

// Devuelve la subsecuencia O(s*t)
string lcs_str(const string &s, const string &t){
    int n=len(s), m=len(t);
    for(int i=1; i<=n; ++i){
        for(int j=1; j<=m; ++j){
            if(s[i-1]==t[j-1]) dp[i][j]=dp[i-1][j-1]+1;
            else dp[i][j]=max(dp[i-1][j], dp[i][j-1]);
        }
    }
    int i=n, j=m;
    string res="";
    while(i>0 && j>0){
        if(s[i-1]==t[j-1]){
            res=s[i-1]+res; i--; j--;
        } else if(dp[i-1][j]>dp[i][j-1]) i--;
        else j--;
    }
    return res;
}

```

8.15 Longest Common Substring

```

// Implementar primero suffix-automata-forma-1
// Retorna la subcadena comun mas larga entre S y T O(S*T)
string lcs(string S, string T){
    sa_init();
    for(int i=0; i<sz(S); i++) sa_extend(S[i]);
    int v=0, l=0, best=0, bestpos=0;
    for(int i=0; i<sz(T); i++){
        while(v && !st[v].next.count(T[i])){
            v=st[v].link;
            l=st[v].len;
        }
        if(st[v].next.count(T[i])){
            v=st[v].next[T[i]];

```

```

            l++;
        }
        if(l>best){
            best=l;
            bestpos=i;
        }
    }
    return T.substr(bestpos-best+1, best);
}

```

8.16 Lyndon Factorization

```

// La factorizacion de Lyndon de un string es una lista
// de strings no vacios
// tal que el string original es la concatenacion de los
// strings de la lista
// en orden lexicografico. Ademas, cada string de la
// lista es un string de
// Lyndon, es decir, es un string que es
// lexicograficamente menor que todos
// sus sufijos no triviales. Por ejemplo "ab"<"ba".
// Tambien los strings estan
// ordenados de mayor a menor.

// El algoritmo de Duval encuentra la factorizacion de
// Lyndon de un string en O(n)
vs duval(string const& s) {
    int n=len(s), i=0;
    vs factorization;
    while(i<n){
        int j=i+1, k=i;
        while(j<n && s[k]<=s[j]){
            if(s[k]<s[j]) k=i;
            else k++;
            j++;
        }
        while(i<=k){
            factorization.push_back(s.substr(i, j-k));
            i+=j-k;
        }
    }
    return factorization;
}

int main() {
    string s="aabaaaab";
    vs factorization=duval(s);
    for(string& factor:factorization) cout<<factor<<"\n";
}

```

8.17 Cantidad Substring por len


```
// Implementar primero suffix-array-forma-2 y meter la
// funcion dentro
// O(n)
void numeroSubstringsPorTamano() {
    vl ps(n+1);
    for(int i=1; i<n; i++) {
        int l=lcp[i-1]+1;
        int r=n-l-pos[i];
        ps[l]++;
        ps[r+1]--;
    }
    for(int i=1; i<n; i++) {
        ps[i]+=ps[i-1];
    }
    for(int i=1; i<n; i++) {
        cout<<ps[i]<<" ";
    }
}
```

8.18 Cantidad Substrings

```
// Implementar primero suffix-array-forma-1
int different_substrings(string s) { //O(nlogn)
    vi sa=suffix_array(s);
    vi lcp=lcp_construction(s, sa);
    int n=len(s);
    int act=n*(n+1); act/=2;
    for(int i=0; i<n-1; i++) act-=lcp[i];
    return act;
}

// Otra forma con hashing O(n^2)
int count_unique_substrings(string const& s) {
    int n = s.size();
    // Ojo con p y m
    const int p=31;
    const int m=1e9+9;
    ll p_pow[n], h[n+1];
    p_pow[0]=1;
    // Precalculo de potencias de p
    for(int i=1; i<n; i++) p_pow[i]=(p_pow[i-1]*p)%m;
    // Precalculo de hashes de prefijos de s
    for(int i=0; i<n; i++) h[i+1]=(h[i]+(s[i]-'a'+1)*p_pow[i])%m;
    int cnt=0;
    for(int l=1; l<=n; l++) {
        unordered_set<ll> hs;
        for(int i=0; i<=n-l; i++) {
            ll cur_h=(h[i+l]+m-h[i])%m;
            cur_h=(cur_h*p_pow[n-i-1])%m;
            hs.insert(cur_h);
        }
        cnt+=hs.size();
    }
}
```

```
    }
    return cnt;
}
```

8.19 Kth-Substring con repeticiones

```
// Implementar primero suffix-automata-forma-2 y meter la
// funcion dentro
// El k-esimo substring lexicografico con repeticiones O(
// n+m)
void kthSubstr(ll k) {
    sort(order.rbegin(), order.rend());
    for(auto [_, u]:order) {
        cnt[link[u]]+=cnt[u];
    }
    vl dp(last+1);
    function<void(int)>dfs=[&](int u) {
        dp[u]=cnt[u];
        for(int i=0; i<26; i++) {
            if(!nxt[u][i]) continue;
            int v=nxt[u][i];
            if(!dp[v]) dfs(v);
            dp[u]+=dp[v];
        }
    };
    dfs(0);
    int u=0;
    while(k>0) {
        for(int i=0; i<26; i++) {
            if(!nxt[u][i]) continue;
            int v=nxt[u][i];
            if(k>dp[v]) {
                k-=dp[v];
            } else {
                cout<<(char)('a'+i);
                k-=cnt[v];
                u=v;
                break;
            }
        }
    }
}
```

8.20 Kth-substring sin repeticiones

```
// Implementar primero suffix-array-forma-2 y meter la
// funcion dentro
// El k-esimo substring lexicografico sin repeticiones O(
// n)
string kthSubstr(ll k) {
    for(int i=1; i<n; i++) {
        int nxt=n-l-pos[i]-lcp[i-1];
    }
}
```

```

    if(k>nxt){
        k-=nxt;
    }else{
        return s.substr(pos[i], k + lcp[i-1]);
    }
}
}

```

8.21 Primera aparicion patrones

```

// Implementar primero suffix-automata-forma-2 y meter la
// funcion dentro
// La primera aparicion de t en s O(t)
int firstMatching(const string &t) {
    int act=0;
    for(char c:t){
        int cc=c-'a';
        if(!nxt[act][cc])return -1;
        act=nxt[act][cc];
    }
    return firstPos[act]-sz(t)+1;
}

```

8.22 Repetitions

```

// implementar primero z_function
// El algoritmo encuentra todas las repeticiones de un
// string O(nlogn)
int get_z(vi const& z, int i) {
    if (0<=i && i<sz(z))return z[i];
    else return 0;
}

vii repetitions;
void convert_to_repetitions(int shift, bool left, int
    cntr, int l, int k1, int k2){
    for(int l1=max(1,l-k2);l1<=min(l,k1);l1++) {
        if(left && l1==l)break;
        int l2=l-l1;
        int pos=shift+(left?cntr-l1:cntr-l-l1+1);
        repetitions.emplace_back(pos,pos+2*l-1);
    }
}

void find_repetitions(string s, int shift=0){
    int n=len(s);
    if(n==1)return;
    int nu=n/2;
    int nv=n-nu;
    string u=s.substr(0,nu);
    string v=s.substr(nu);
    string ru(u.rbegin(), u.rend());

```

```

    string rv(v.rbegin(), v.rend());
    find_repetitions(u, shift);
    find_repetitions(v, shift+nu);
    vi z1=z_function(ru);
    vi z2=z_function(v+'#'+u);
    vi z3=z_function(ru+'#'+rv);
    vi z4=z_function(v);
    for (int cntr=0;cntr<n;cntr++) {
        int l, k1, k2;
        if(cntr<nu) {
            l=nu-cntr;
            k1=get_z(z1, nu-cntr);
            k2=get_z(z2, nv+1+cntr);
        }else{
            l=cntr-nu+1;
            k1=get_z(z3,nu+1+nv-1-(cntr-nu));
            k2=get_z(z4,(cntr-nu)+1);
        }
        if(k1+k2>=l)convert_to_repetitions(shift, cntr<nu,
            cntr, l, k1, k2);
    }
}

int main() {
    find_repetitions(string);
    for(auto& rep:repetitions)cout<<rep.first<<" "<<rep.
        second<<"\n";
}

```

8.23 Substring mas largo repetido

```

// Implementar primero suffix-array-forma-1
string longest_repeated_substring(string& s){ //O(nlogn)
    // Si se tienen que sacar varios, entonces son todos
    // los que sean iguales al maximo
    vi sa=suffix_array(s);
    vi lcp=lcp_construction(s,sa);
    int n=len(s);
    int max_len=0, start=0;
    for(int i=0;i<n-1;i++){
        if(lcp[i]>max_len){
            max_len=lcp[i];
            start=sa[i];
        }
    }
    return s.substr(start,max_len);
}

```

9 Geometria

9.1 Puntos

```

// Punto entero
struct point{
    ll x,y;
    point(ll x,ll y): x(x),y(y){}
};

// Punto flotante
struct point{
    double x,y;
    point(double _x,double _y): x(_x),y(_y){}
    bool operator==(point other) const{
        return (fabs(x-other.x)<EPS) && (fabs(y-other.y)<
            EPS);
    };
};

// Distancia entre dos puntos
double dist(point p1, point p2){
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-
        p2.y));
}

// Rotacion de un punto
point rotate(point p, double theta){
    // rotar por theta grados respecto al origen (0,0
    double rad = theta*(M_PI/180);
    return point(p.x*cos(rad)-p.y*sin(rad),p.x*sin(rad)+p
        .y*cos(rad));
}

```

9.2 Lineas

```

// Linea de flotantes de la forma ax+by+c=0
struct line{double a,b,c;};

// Creacion de linea con dos puntos
// b=1 para lineas no verticales y b =0 para verticales
void pointsToLine(point p1,point p2,line& l){
    if (fabs(p1.x-p2.x)<EPS){
        l.a=1.0; l.b=0.0; l.c=-p1.x;
    }else{
        l.a= -double(p1.y-p2.y)/(p1.x-p2.x);
        l.b= 1.0;
        l.c= -double(l.a*p1.x)-p1.y;
    }
}

// Comprobacion de lineas paralelas
bool areParallel(line l1,line l2){
    return (fabs(l1.a-l2.a)<EPS) && (fabs(l1.b-l2.b)<EPS)
;
}

// Comprobacion de lineas iguales
bool areSame(line l1,line l2){
    return areParallel(l1,l2) && (fabs(l1.c-l2.c)<EPS);
}

```

```

}

// Disntacia de un punto a una linea
double distPointToLineEq(line l, point p){
    return fabs(l.a*p.x + l.b*p.y + l.c)/sqrt(l.a*l.a+l.b
        *l.b);
}

bool areIntersect(line l1, line l2, point& p){
    if (areParallel(l1,l2)) return false;
    // resolver sistema 2x2
    p.x = (l2.b*l1.c - l1.b*l2.c)/(l2.a*l1.b - l1.a*l2.b)
;

    // CS: comprobar linea vertical -> div por cero
    if (fabs(l1.b)>EPS) p.y = -(l1.a*p.x + l1.c);
    else p.y = -(l2.a*p.x + l2.c);
    return true;
}

```

9.3 Vectores

```

// Creacion de un vector
struct vec{
    double x,y;
    vec(double x,double y): x(x),y(y){}
};

// Puntos a vector
vec toVec(point a,point b){
    return vec(b.x-a.x , b.y-a.y);
}

// Escalar un vector
vec scale(vec v, double s){
    // s no negativo:
    // <1 mas corto
    // 1 igual
    // >1 mas largo
    return vec(v.x*s,v.y*s);
}

// Trasladar p segun v
point traslate(point p, vec v){
    return point(p.x+v.x , p.y+v.y);
}

// Producto Punto
double dot(vec a,vec b){
    return (a.x*b.x + a.y*b.y);
}

// Cuadrado de la norma
double norm_sq(vec v){
    return v.x*v.x + v.y*v.y;
}

```

```

// Angulo formado por aob
double angle(point a, point o, point b){
    vec oa = toVec(o,a);
    vec ob = toVec(o,b);
    return acos(dot(oa,ob)/sqrt(norm_sq(oa)*norm_sq(ob)))
;
}

// Producto cruz
double cross(vec a, vec b){
    return (a.x*b.y)-(a.y*b.x);
}

// Lado respecto una linea pq
bool ccw(point p,point q,point r){
    // Devuelve verdadero si el punto r esta a la
    // izquierda de la linea pq
    return cross(toVec(p,q),toVec(p,r))>0;
}

// Colinear
bool collinear(point p, point q, point r){
    return fabs(cross(toVec(p,q), toVec(p,r)))<EPS;
}

```

9.4 Poligonos

```

// Crear un poligono
// la idea es crearlo con algun orden ya sea horario o
// anti-horario
// y debe cerrarse
vector<point> Poligono;

// Perimetro de un poligono
double perimeter(const vector<point>& P){
    double result =0.0;
    for (int i =0;i<(int)P.size()-1;i++)result+= dist(P[i],P[i+1]);
    return result;
}

// Area de un poligono
double area(const vector<point>& P){
    // la mitad del determinante
    double result = 0.0, x1,y1,x2,y2;
    for (int i =0;i<(int)P.size()-1;i++){
        x1 = P[i].x;
        x2 = P[i+1].x;
        y1 = P[i].y;
        y2 = P[i+1].y;
        result += (x1*y2 - x2*y1);
    }
    return fabs(result/2.0);
}

// Comprobacion de si es Convexo un poligono

```

```

bool isConvex(const vector<point>& P){
    int sz = (int)P.size();
    if (sz<=3) return false;
    bool isLeft = ccw(P[0],P[1],P[2]);
    for (int i =1;i<sz-1;i++)
        if (ccw(P[i],P[i+1],P[(i+2)==sz ? 1:i+2])!=isLeft)
            return false;
    return true;
}

// Comprobar si un punto esta dentro de un poligono
bool inPoligono(point pt, const vector<point>& P){
    // P puede ser concavo/convexo
    if ((int)P.size()==0) return false;
    double sum =0;
    for (int i =0;i<(int)P.size()-1;i++){
        if (ccw(pt,P[i],P[i+1]))
            sum += angle(P[i],pt,P[i+1]); // izquierda/
            anti-horario
        else sum -= angle(P[i],pt,P[i+1]); // derecha/
            horario
    }
    return fabs(fabs(sum)-2*M_PI)<EPS;
}

```

9.5 Convex Hull

```

struct pt{
    double x,y;
    int type;
    pt(double x,double y,int t): x(x),y(y),type(t){}
};

// Devuelve hacia donde esta un punto c, respecto una
// linea ab
int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // en la derecha
    if (v > 0) return +1; // en la izquierda
    return 0; // colinear
}

// imprime verdadero el punto c, esta a la derecha de la
// linea pb,
// tambien da true si son colineales e
// include_collinear == true
bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}

// nos dice si tres puntos son colineales
bool collinear(pt a, pt b, pt c) { return orientation(a,
    b, c) == 0; }

```

```

void convex_hull(vector<pt>& a, bool include_collinear =
false) {
    // Obtenemos el pivote como el menor punto con un
    // criterio dado
    // (menor y o si no menor x)
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt
b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    // Ordenamos los puntos en un orden horario, los
    // elementos colineales terminan
    // siendo arrastrados al final y si existe empate en
    // el angulo sera el que este mas cerca
    // del pivote
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt&
b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0
.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.
y-b.y);
    });
}

```

10 Teoría y miscelánea

10.1 Sumatorias

- $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=1}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$
- $\sum_{i=1}^n i^5 = \frac{(n(n+1))^2(2n^2+2n-1)}{12}$
- $\sum_{i=1}^n i^3 = \left(\frac{n(n+1)}{2}\right)^2$
- $\sum_{i=0}^n x^i = \frac{x^{n+1}-1}{x-1}$ para $x \neq 1$

10.2 Teoría de Grafos

10.2.1 Teorema de Euler

En un grafo conectado planar, se cumple que $V - E + F = 2$, donde V es el número de vértices, E es el número de aristas y F es el número de caras.

10.2.2 Planaridad de Grafos

Un grafo es planar si y solo si no contiene un subgrafo homeomorfo a K_5 (grafo completo con 5 vértices) ni a $K_{3,3}$ (grafo bipartito completo con 3 vértices en cada conjunto).

```

        return o < 0;
    });
    // Busca donde empiezan los colineales (estan al
    // final) e invierte su orden
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i
            --;
        reverse(a.begin()+i+1, a.end());
    }
    // Aplicacion de graham
    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.
back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }
    a = st;
}

```

10.3 Teoría de Números

10.3.1 Ecuaciones Diofánticas Lineales

Una ecuación diofántica lineal es una ecuación en la que se buscan soluciones enteras x e y que satisfagan la relación lineal $ax + by = c$, donde a , b y c son constantes dadas.

Para encontrar soluciones enteras positivas en una ecuación diofántica lineal, podemos seguir el siguiente proceso:

1. Encontrar una solución particular: Encuentra una solución particular (x_0, y_0) de la ecuación. Esto puede hacerse utilizando el algoritmo de Euclides extendido.
2. Encontrar la solución general: Una vez que tengas una solución particular, puedes obtener la solución general utilizando la fórmula:

$$x = x_0 + \frac{b}{\text{mcd}(a, b)} \cdot t$$

$$y = y_0 - \frac{a}{\text{mcd}(a, b)} \cdot t$$

donde t es un parámetro entero.

3. Restringir a soluciones positivas: Si deseas soluciones positivas, asegúrate de que las soluciones generales satisfagan $x \geq 0$ y $y \geq 0$. Puedes ajustar el valor de t para cumplir con estas restricciones.

10.3.2 Pequeño Teorema de Fermat

Si p es un número primo y a es un entero no divisible por p , entonces $a^{p-1} \equiv 1 \pmod{p}$.

10.3.3 Teorema de Euler

Para cualquier número entero positivo n y un entero a coprimo con n , se cumple que $a^{\phi(n)} \equiv 1 \pmod{n}$, donde $\phi(n)$ es la función phi de Euler, que representa la cantidad de enteros positivos menores que n y coprimos con n .

10.4 Teorema de Pick

Sea un polígono simple cuyos vertices tienen coordenadas enteras. Si B es el número de puntos enteros en el borde, I el número de puntos enteros en el interior del polígono, entonces el área A del polígono se puede calcular con la fórmula:

$$A = I + \frac{B}{2} - 1$$

10.5 Combinatoria

10.5.1 Permutaciones

El número de permutaciones de n objetos distintos tomados de a r a la vez (sin repetición) se denota como $P(n, r)$ y se calcula mediante:

$$P(n, r) = \frac{n!}{(n-r)!}$$

10.5.2 Combinaciones

El número de combinaciones de n objetos distintos tomados de a r a la vez (sin repetición) se denota como $C(n, r)$ o $\binom{n}{r}$ y se calcula mediante:

$$C(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

10.5.3 Permutaciones con Repetición

El número de permutaciones de n objetos tomando en cuenta repeticiones se denota como $P_{\text{rep}}(n; n_1, n_2, \dots, n_k)$ y se calcula mediante:

$$P_{\text{rep}}(n; n_1, n_2, \dots, n_k) = \frac{n!}{n_1! n_2! \dots n_k!}$$

10.5.4 Combinaciones con Repetición

El número de combinaciones de n objetos tomando en cuenta repeticiones se denota como $C_{\text{rep}}(n; n_1, n_2, \dots, n_k)$ y se calcula mediante:

$$C_{\text{rep}}(n; n_1, n_2, \dots, n_k) = \binom{n+k-1}{n} = \binom{n+k-1}{k-1}$$

10.5.5 Números de Catalan

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Los números de Catalan también pueden calcularse utilizando la siguiente fórmula recursiva:

$$C_0 = 1$$

$$C_{n+1} = \frac{4n+2}{n+2} C_n$$

Usos:

- $\text{Cat}(n)$ cuenta el número de árboles binarios distintos con n vértices.
- $\text{Cat}(n)$ cuenta el número de expresiones que contienen n pares de paréntesis correctamente emparejados.
- $\text{Cat}(n)$ cuenta el número de formas diferentes en que se pueden colocar $n+1$ factores entre paréntesis, por ejemplo, para $n = 3$ y $3+1 = 4$ factores: a, b, c, d , tenemos: $(ab)(cd), a(b(cd)), ((ab)c)d$ y $a((bc)d)$.
- Los números de Catalan cuentan la cantidad de caminos no cruzados en una rejilla $n \times n$ que se pueden trazar desde una esquina de un cuadrado o rectángulo a la esquina opuesta, moviéndose solo hacia arriba y hacia la derecha.
- Los números de Catalan representan el número de árboles binarios completos con $n+1$ hojas.
- $\text{Cat}(n)$ cuenta el número de formas en que se puede triangular un polígono convexo de $n+2$ lados. Otra forma de decirlo es como la cantidad de formas de dividir un polígono convexo en triángulos utilizando diagonales no cruzadas.

10.6 DP Optimization Theory

| Name | Original Recurrence | Sufficient Condition | From | To |
|-------|---|--|-----------|----------------|
| CH 1 | $dp[i] = \min_{j < i} \{dp[j] + b[j] * a[i]\}$ | $b[j] \geq b[j+1]$ Optionally $a[i] \leq a[i+1]$ | $O(n^2)$ | $O(n)$ |
| CH 2 | $dp[i][j] = \min_{k < j} \{dp[i-1][k] + b[k] * a[j]\}$ | $b[k] \geq b[k+1]$ Optionally $a[j] \leq a[j+1]$ | $O(kn^2)$ | $O(kn)$ |
| D&Q | $dp[i][j] = \min_{k < j} \{dp[i-1][k] + C[k][j]\}$ | $A[i][j] \leq A[i][j+1]$ | $O(kn^2)$ | $O(kn \log n)$ |
| Knuth | $dp[i][j] = \min_{i < k < j} \{dp[i][k] + dp[k][j]\} + C[i][j]$ | $A[i, j-1] \leq A[i, j] \leq A[i+1, j]$ | $O(n^3)$ | $O(n^2)$ |

Notes:

- $A[i][j]$ - the smallest k that gives the optimal answer, for example in $dp[i][j] = dp[i-1][k] + C[k][j]$
- $C[i][j]$ - some given cost function
- We can generalize a bit in the following way $dp[i] = \min_{j < i} \{F[j] + b[j] * a[i]\}$, where $F[j]$ is computed from $dp[j]$ in constant time