Notebook UNosnovatos

Contents

1 C++

1.1 C++ plantilla

```cpp
#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> vl;
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 Librerias

```cpp
// 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 tamanio del set de bits
S = 0;
S = (1<<n) - 1; // 7 (111)

// n es el tamanio 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 Ordered set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;
// -------- CONSTRUCTOR -------- //
// 1. Para ordenar por MAX cambiar less<int> por greater<
    int>
// 2. Para multiset cambiar less<int> por less_equal<int>
//      Para borrar siendo multiset:
//      int idx = st.order_of_key(value);
//      st.erase(st.find_by_order(idx));
// -------- METHODS --------- //
st.find_by_order(k) // returns pointer to the k-th
    smallest element
st.order_of_key(x)  // returns how many elements are
    smaller than x
st.find_by_order(k) == st.end() // true, if element does
    not exist
```

2.2 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--;
            }
        }
};
```

2.3 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;
    }
};
```

2.4 ST iterativo

```
struct segtree{
    int n; vl v; ll nulo = 0;

    ll op(ll a, ll b) {return a + b;}

    segtree(int n) : n(n), v(2*n, nulo){}

    segtree(vl &a) : n(sz(a)), v(2*n){
        for(int i = 0; i<n; i++) v[n + i] = a[i];
        for (int i = n-1; i>=1; --i) v[i] = op(v[i<<1], v
            [i<<1|1]);
    }
```

```
    void upd(int k, ll nv){
        for (v[k += n] = nv; k > 1; k >>= 1) v[k>>1] = op
            (v[k], v[k^1]);
    }

    ll get(int l, int r){
        ll vl = nulo, vr = nulo;
        for (l += n, r += n+1; l < r; l >>= 1, r >>= 1){
            if (l&1) vl = op(vl, v[l++]);
            if (r&1) vr = op(v[--r], vr);
        }
        return op(vl, vr);
    }
};
```

2.5 Segment Tree

```
int nullValue = 0;

struct nodeST{
    nodeST *left, *right;
    int l, r; ll value, lazy, lazy1;

    nodeST(vi &v, int l, int r) : l(l), r(r){
        int m = (l+r)>>1;
        lazy = 0;
        lazy1 = 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(lazy1){
            value = lazy1 * (r-l+1);
            if (l != r){
                left->lazy1 = lazy1, right->lazy1 = lazy1
                    ;
                left->lazy = 0, right->lazy = 0;
            }
            lazy1 = 0;
            lazy = 0;
        }
        else{
            value += lazy * (r-l+1);
            if (l != r){
                if(left->lazy1) left->lazy1 += lazy;
                else left->lazy += lazy;
                if(right->lazy1) right->lazy1 += 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 upd1(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->upd1(i, j, nv);
        right->upd1(i, j, nv);
```

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

2.6 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
    // tamano 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);
    }
};
```

2.7 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.actl(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 subset 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, elminar 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 optr) {
    if (l > r)
        return;

    int mid = (l + r) >> 1;
    pair<long long, int> best = {LLONG_MAX, -1};

    for (int k = optl; k <= min(mid, optr); 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, optr);
}

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);
```

```cpp
    printf("There are %d SCCs\n", numSCC);
}
```

4.5 Tarjan

```cpp
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

```cpp
//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

```cpp
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

```cpp
//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)) )
vl primeFactors(ll N) {
    vl 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];              //Eliminarlo de 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(ll 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
        }
        return (N != 1) ? 2*ans : ans;  //Ultimo factor = N^1
    }

    //Suma de los divisores de N
    //N = a^i * b^i * ... * c^k => N = (a^(i+1) - 1) / (a-1)
        + ...
    ll sumDiv(ll N) {
        ll ans = 1;          // empezar con ans = 1
        for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <=
            N); ++i) {
            ll multiplier = p[i], total = 1;
            while (N%p[i] == 0) {
                N /= p[i];
                total += multiplier;
                multiplier *= p[i];
            }                                   // total para
            ans *= total;                       // este
                factor primo
        }
        if (N != 1) ans *= (N+1);   // N^2-1/N-1 = N+1
        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 Exponenciacion binaria

```
    ll binpow(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 Exponenciacion 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;
    }
    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

```cpp
// 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 * 1ll * a / g) % m;
    }

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

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
        cur = (cur * 1ll * a) % m;
    }

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

6.12 Freivalds algorithm

```cpp
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--) {
        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;
}
```

6.13 Pollard Rho

```cpp
//O(n^(1/4)) (?)
ll pollard_rho(ll n, ll c) {
  ll x = 2, y = 2, i = 1, k = 2, d;
  while (true) {
    x = (mul(x, x, n) + c);
    if (x >= n) x -= n;
    d = __gcd(x - y, n);
    if (d > 1) return d;
    if (++i == k) y = x, k <<= 1;
  }
  return n;
}
void factorize(ll n, vector<ll> &f) {
  if (n == 1) return;
  if (is_prime(n)) {
    f.push_back(n);
    return;
  }
  ll d = n;
  for (int i = 2; d == n; i++)
    d = pollard_rho(n, i);
  factorize(d, f);
  factorize(n/d, f);
}
```

7 Metodos numericos

7.1 Ternary Search

```cpp
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=l+(r-l)/3;
    double m2=r-(r-l)/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

```cpp
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 - l]);
    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 tamano "<<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 tamano 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]+m-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):n(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 tamano)
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="aabaaab";
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-1-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-1-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

```
typedef double lf;
const lf eps = 1e-9;
typedef double T;
struct pt {
    T x, y;
    pt operator + (pt p) { return {x+p.x, y+p.y}; }
    pt operator - (pt p) { return {x-p.x, y-p.y}; }
    pt operator * (pt p) { return {x*p.x-y*p.y, x*p.y+y*p
       .x}; }
    pt operator * (T d) { return {x*d, y*d}; }
    pt operator / (lf d) { return {x/d, y/d}; } /// only
       for floating point
    bool operator == (pt b) { return x == b.x && y == b.y
       ; }
    bool operator != (pt b) { return !(*this == b); }
```

```
    bool operator < (const pt &o) const { return y < o.y || (y == o.y && x < o.x); }
    bool operator > (const pt &o) const { return y > o.y || (y == o.y && x > o.x); }
};
int cmp (lf a, lf b) { return (a + eps < b ? -1 :(b + eps < a ? 1 : 0)); } //double comparator

T norm(pt a) { return a.x*a.x + a.y*a.y; }
lf abs(pt a) { return sqrt(norm(a)); }
lf arg(pt a) { return atan2(a.y, a.x); }
pt unit(pt a) { return a/abs(a); }

T dot(pt a, pt b) { return a.x*b.x + a.y*b.y; } // x = 90 -> cos = 0
T cross(pt a, pt b) { return a.x*b.y - a.y*b.x; } // x = 180 -> sin = 0
T orient(pt a, pt b, pt c) { return cross(b-a,c-a); }// clockwise = -
pt rot(pt p, lf a) { return {p.x*cos(a) - p.y*sin(a), p.x*sin(a) + p.y*cos(a)}; }
pt rotate_to_b(pt a, pt b, lf ang) { return rot(a-b, ang)+b; } // rotate by ang center b
pt rot90ccw(pt p) { return {-p.y, p.x}; }
pt rot90cw(pt p) { return {p.y, -p.x}; }
pt translate(pt p, pt v) { return p+v; }
pt scale(pt p, double f, pt c) { return c + (p-c)*f; } // c-center
bool are_perp(pt v, pt w) { return dot(v,w) == 0; }
int sign(T x) { return (T(0) < x) - (x < T(0)); }

bool in_angle(pt a, pt b, pt c, pt x) { // x inside angle abc (center in a)
    assert(orient(a,b,c) != 0);
    if (orient(a,b,c) < 0) swap(b,c);
    return orient(a,b,x) >= 0 && orient(a,c,x) <= 0;
}
//angle bwn 2 vectors
lf angle(pt a, pt b) { return acos(max(-1.0, min(1.0, dot(a,b)/abs(a)/abs(b)))); }
lf angle(pt a, pt b) { return atan2(cross(a, b), dot(a, b)); }
/// returns vector to transform points
pt get_linear_transformation(pt p, pt q, pt r, pt fp, pt fq) {
    pt pq = q-p, num{cross(pq, fq-fp), dot(pq, fq-fp)};
    return fp + pt{cross(r-p, num), dot(r-p, num)} / norm(pq);
}
bool half(pt p) { /// true if is in (0, 180] (line is x axis)
    assert(p.x != 0 || p.y != 0); /// the argument of (0,0) is undefined
    return p.y > 0 || (p.y == 0 && p.x < 0);
}
bool half_from(pt p, pt v = {1, 0}) { //line is v (above v is true)
    return cross(v,p) < 0 || (cross(v,p) == 0 && dot(v,p) < 0);
}
bool polar_cmp(const pt &a, const pt &b) {//polar sort
    return make_tuple(half(a), 0) < make_tuple(half(b), cross(a,b));
//   return make_tuple(half(a), 0, sq(a)) < make_tuple(half(b), cross(a, b), sq(b)); // further ones appear later
}
```

9.2 Lineas

```
struct line {
    pt v; T c; // v:direction c: pos in y axis
    line(pt v, T c) : v(v), c(c) {}
    line(T a, T b, T c) : v({b,-a}), c(c) {} // ax + by = c
    line(pt p, pt q) : v(q-p), c(cross(v,p)) {}
    T side(pt p) { return cross(v,p)-c; }
    lf dist(pt p) { return abs(side(p)) / abs(v); }
    lf sq_dist(pt p) { return side(p)*side(p) / (lf)norm(v); }
    line perp_through(pt p) { return {p, p + rot90ccw(v)}; } // line perp to v passing through p
    bool cmp_proj(pt p, pt q) { return dot(v,p) < dot(v,q); } // order for points over the line
    line translate(pt t) { return {v, c + cross(v,t)}; }
    line shift_left(double d) { return {v, c + d*abs(v)}; }
    pt proj(pt p) { return p - rot90ccw(v)*side(p)/norm(v); } // pt proyected on the line
    pt refl(pt p) { return p - rot90ccw(v)*2*side(p)/norm(v); } // pt reflected on the other side of the line
};

bool inter_ll(line l1, line l2, pt &out) {
    T d = cross(l1.v, l2.v);
    if (d == 0) return false;
    out = (l2.v*l1.c - l1.v*l2.c) / d; // floating points
    return true;
}
//bisector divides the angle in 2 equal angles
//interior line goes on the same direction as l1 and l2
line bisector(line l1, line l2, bool interior) {
    assert(cross(l1.v, l2.v) != 0); /// l1 and l2 cannot be parallel!
    lf sign = interior ? 1 : -1;
    return {l2.v/abs(l2.v) + l1.v/abs(l1.v) * sign,
            l2.c/abs(l2.v) + l1.c/abs(l1.v) * sign};
}
```

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 negatico:
    // <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

```
enum {IN, OUT, ON};
struct polygon {
    vector<pt> p;
    polygon(int n) : p(n) {}
    int top = -1, bottom = -1;
    void delete_repetead() {
        vector<pt> aux;
        sort(p.begin(), p.end());
        for(pt &i : p)
            if(aux.empty() || aux.back() != i)
              aux.push_back(i);
        p.swap(aux);
    }
    bool is_convex() {
        bool pos = 0, neg = 0;
        for (int i = 0, n = p.size(); i < n; i++) {
            int o = orient(p[i], p[(i+1)%n], p[(i+2)%n]);
            if (o > 0) pos = 1;
            if (o < 0) neg = 1;
        }
        return !(pos && neg);
    }
    lf area(bool s = false) { // better on clockwise
       order
        lf ans = 0;
        for (int i = 0, n = p.size(); i < n; i++)
            ans += cross(p[i], p[(i+1)%n]);
        ans /= 2;
        return s ? ans : abs(ans);
    }
    lf perimeter() {
        lf per = 0;
        for(int i = 0, n = p.size(); i < n; i++)
           per += abs(p[i] - p[(i+1)%n]);
        return per;
    }
    bool above(pt a, pt p) { return p.y >= a.y; }
    bool crosses_ray(pt a, pt p, pt q) { // pq crosses
       ray from a
        return (above(a,q)-above(a,p))*orient(a,p,q) > 0;
    }
    int in_polygon(pt a) {
        int crosses = 0;
        for(int i = 0, n = p.size(); i < n; i++) {
            if(on_segment(p[i], p[(i+1)%n], a)) return ON
                ;
            crosses += crosses_ray(a, p[i], p[(i+1)%n]);
```

```
    }
    return (crosses&1 ? IN : OUT);
}
void normalize() { /// polygon is CCW
    bottom = min_element(p.begin(), p.end()) - p.
        begin();
    vector<pt> tmp(p.begin()+bottom, p.end());
    tmp.insert(tmp.end(), p.begin(), p.begin()+bottom
        );
    p.swap(tmp);
    bottom = 0;
    top = max_element(p.begin(), p.end()) - p.begin()
        ;
}
int in_convex(pt a) {
    assert(bottom == 0 && top != -1);
    if(a < p[0] || a > p[top]) return OUT;
    T orientation = orient(p[0], p[top], a);
    if(orientation == 0) {
        if(a == p[0] || a == p[top]) return ON;
        return top == 1 || top + 1 == p.size() ? ON :
            IN;
    } else if (orientation < 0) {
        auto it = lower_bound(p.begin()+1, p.begin()+
            top, a);
        T d = orient(*prev(it), a, *it);
        return d < 0 ? IN : (d > 0 ? OUT: ON);
    } else {
        auto it = upper_bound(p.rbegin(), p.rend()-
            top-1, a);
        T d = orient(*it, a, it == p.rbegin() ? p[0]
            : *prev(it));
        return d < 0 ? IN : (d > 0 ? OUT: ON);
    }
}
polygon cut(pt a, pt b) { // cuts polygon on line ab
    line l(a, b);
    polygon new_polygon(0);
    for(int i = 0, n = p.size(); i < n; ++i) {
        pt c = p[i], d = p[(i+1)%n];
        lf abc = cross(b-a, c-a), abd = cross(b-a, d-
            a);
        if(abc >= 0) new_polygon.p.push_back(c);
        if(abc*abd < 0) {
          pt out; inter_ll(l, line(c, d), out);
          new_polygon.p.push_back(out);
        }
    }
    return new_polygon;
}
void convex_hull() {
    sort(p.begin(), p.end());
    vector<pt> ch;
    ch.reserve(p.size()+1);
    for(int it = 0; it < 2; it++) {
        int start = ch.size();
        for(auto &a : p) {
            /// if colineal are needed, use < and
                remove repeated points
            while(ch.size() >= start+2 && orient(ch[
                ch.size()-2], ch.back(), a) <= 0)
                 ch.pop_back();
            ch.push_back(a);
        }
        ch.pop_back();
        reverse(p.begin(), p.end());
    }
    if(ch.size() == 2 && ch[0] == ch[1]) ch.pop_back
        ();
    /// be careful with CH of size < 3
    p.swap(ch);
}
vector<pii> antipodal() {
    vector<pii> ans;
    int n = p.size();
    if(n == 2) ans.push_back({0, 1});
    if(n < 3) return ans;
    auto nxt = [&](int x) { return (x+1 == n ? 0 : x
        +1); };
    auto area2 = [&](pt a, pt b, pt c) { return cross
        (b-a, c-a); };
    int b0 = 0;
    while(abs(area2(p[n - 1], p[0], p[nxt(b0)])) >
        abs(area2(p[n - 1], p[0], p[b0]))) ++b0;
    for(int b = b0, a = 0; b != 0 && a <= b0; ++a) {
        ans.push_back({a, b});
        while (abs(area2(p[a], p[nxt(a)], p[nxt(b)]))
            > abs(area2(p[a], p[nxt(a)], p[b]))) {
            b = nxt(b);
            if(a != b0 || b != 0) ans.push_back({ a,
                b });
            else return ans;
        }
        if(abs(area2(p[a], p[nxt(a)], p[nxt(b)])) ==
            abs(area2(p[a], p[nxt(a)], p[b]))) {
            if(a != b0 || b != n-1) ans.push_back({ a
                , nxt(b) });
            else ans.push_back({ nxt(a), b });
        }
    }
    return ans;
}
pt centroid() {
    pt c{0, 0};
    lf scale = 6. * area(true);
    for(int i = 0, n = p.size(); i < n; ++i) {
        int j = (i+1 == n ? 0 : i+1);
        c = c + (p[i] + p[j]) * cross(p[i], p[j]);
    }
    return c / scale;
```

```
    }
    ll pick() {
        ll boundary = 0;
        for(int i = 0, n = p.size(); i < n; i++) {
            int j = (i+1 == n ? 0 : i+1);
            boundary += __gcd((ll)abs(p[i].x - p[j].x), (
                ll)abs(p[i].y - p[j].y));
        }
        return area() + 1 - boundary/2;
    }
    pt& operator[] (int i){ return p[i]; }
};
```

9.5 Angulos

Calcula el angulo de una linea con respecto a otra.

```
lf get_ang(pt a, pt b) {
    lf ang = acos(max(lf(-1.0), min(lf(1.0), lf(dot(a,b))
        /abs(a)/abs(b))));
    ang = ang * 180.0 / acos(-1.0);
    if (b.y < 0) ang = lf(360) - ang;
    return ang;
}

lf angle(pt a, pt b) {
    pt xo = {1, 0};
    lf ang = get_ang(xo, b) - get_ang(xo, a);
    if (ang < 0) ang += 360;
    return ang;
}

double DegToRad(double d) {
        return d * acos(-1.0) / 180.0;
}

double RadToDeg(double r) {
        return r * 180.0 / acos(-1.0);
}
```

9.6 Circulos

```
struct circle {
    pt c; T r;
};
// (x-xo)^2 + (y-yo)^2 = r^2
//circle that passes through abc
circle center(pt a, pt b, pt c) {
    b = b-a, c = c-a;
    assert(cross(b,c) != 0); /// no circumcircle if A,B,C
        aligned
    pt cen = a + rot90ccw(b*norm(c) - c*norm(b))/cross(b,
        c)/2;
    return {cen, abs(a-cen)};
}
//centers of the circles that pass through ab and has
    radius r
vector<pt> centers(pt a, pt b, T r) {
    if (abs(a-b) > 2*r + eps) return {};
    pt m = (a+b)/2;
    double f = sqrt(r*r/norm(a-m) - 1);
    pt c = rot90ccw(a-m)*f;
    return {m-c, m+c};
}
int inter_cl(circle c, line l, pair<pt, pt> &out) {
    lf h2 = c.r*c.r - l.sq_dist(c.c);
    if(h2 >= 0) { // line touches circle
        pt p = l.proj(c.c);
        pt h = l.v*sqrt(h2)/abs(l.v); // vector of len h
            parallel to line
        out = {p-h, p+h};
    }
    return 1 + sign(h2); // if 1 -> out.F == out.S
}
int inter_cc(circle c1, circle c2, pair<pt, pt> &out) {
    pt d = c2.c-c1.c;
    double d2 = norm(d);
    if(d2 == 0) { assert(c1.r != c2.r); return 0; } //
        concentric circles (identical)
    double pd = (d2 + c1.r*c1.r - c2.r*c2.r)/2; // = |
        O_1P| * d
    double h2 = c1.r*c1.r - pd*pd/d2; // = h^2
    if(h2 >= 0) {
        pt p = c1.c + d*pd/d2, h = rot90ccw(d)*sqrt(h2/d2
            );
        out = {p-h, p+h};
    }
    return 1 + sign(h2);
}
//circle-line inter = 1
int tangents(circle c1, circle c2, bool inner, vector<
    pair<pt,pt>> &out) {
    if(inner) c2.r = -c2.r; // inner tangent
    pt d = c2.c-c1.c;
    double dr = c1.r-c2.r, d2 = norm(d), h2 = d2-dr*dr;
    if(d2 == 0 || h2 < 0) { assert(h2 != 0); return 0; }
        // (identical)
    for(double s : {-1,1}) {
        pt v = (d*dr + rot90ccw(d)*sqrt(h2)*s)/d2;
        out.push_back({c1.c + v*c1.r, c2.c + v*c2.r});
    }
    return 1 + (h2 > 0); // if 1: circle are tangent
}
//circle targent passing through pt p
int tangent_through_pt(pt p, circle c, pair<pt, pt> &out)
    {
    double d = abs(p - c.c);
    if(d < c.r) return 0;
```

```
    pt base = c.c-p;
    double w = sqrt(norm(base) - c.r*c.r);
    pt a = {w, c.r}, b = {w, -c.r};
    pt s = p + base*a/norm(base)*w;
    pt t = p + base*b/norm(base)*w;
    out = {s, t};
    return 1 + (abs(c.c-p) == c.r);
}
```

9.7 Semiplanos

```
struct halfplane{
    double angle;
    pt p, pq;
    halfplane(){}
    halfplane(pt a, pt b): p(a), pq(b - a) {
        angle = atan2(pq.y,pq.x);
    }
    bool operator < (halfplane b)const{return angle < b.angle;}
    bool out(pt q){return cross(pq, (q-p)) < -eps;} // checks if p is inside the half plane
};

const lf inf = 1e100;
// intersection pt of the lines of 2 halfplanes
pt inter(halfplane& h1, halfplane& h2){
    if(abs(cross(unit(h1.pq), unit(h2.pq))) <= eps)return {inf, inf};
    lf alpha = cross((h2.p - h1.p), h2.pq) / cross(h1.pq, h2.pq);
    return h1.p + (h1.pq * alpha);
}

// intersection of halfplanes
vector<pt> intersect(vector<halfplane>& b){
    vector<pt> box = { {inf, inf}, {-inf, inf}, {-inf, -inf}, {inf, -inf} };
    for(int i = 0; i < 4; i++){
        b.push_back({box[i], box[(i + 1) % 4]});
    }
    sort(b.begin(), b.end());
    int n = b.size(), q = 1, h = 0;
    vector<halfplane> c(n + 10);
    for(int i = 0; i < n; i++){
        while(q < h && b[i].out(inter(c[h], c[h-1]))) h--;
        while(q < h && b[i].out(inter(c[q], c[q+1]))) q++;
        c[++h] = b[i];
        if(q < h && abs(cross(c[h].pq, c[h-1].pq)) < eps){
            if(dot(c[h].pq, 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(inter(c[h], c[h-1]))) h--;
    while(q < h-1 && c[h].out(inter(c[q], c[q+1]))) q++;
    if(h - q <= 1)return {};
    c[h+1] = c[q];
    vector<pt> s;
    for(int i = q; i < h+1; i++) s.pb(inter(c[i], c[i+1]));
    return s;
}
```

9.8 Segmentos

```
bool in_disk(pt a, pt b, pt p) { // pt p inside ab disk
    return dot(a-p, b-p) <= 0;
}
bool on_segment(pt a, pt b, pt p) { // p on ab
    return orient(a,b,p) == 0 && in_disk(a,b,p);
}
// ab crossing cd
bool proper_inter(pt a, pt b, pt c, pt d, pt &out) {
    T oa = orient(c,d,a),
    ob = orient(c,d,b),
    oc = orient(a,b,c),
    od = orient(a,b,d);
    /// Proper intersection exists iff opposite signs
    if (oa*ob < 0 && oc*od < 0) {
        out = (a*ob - b*oa) / (ob-oa);
        return true;
    }
    return false;
}
// intersection bwn segments
set<pt> inter_ss(pt a, pt b, pt c, pt d) {
    pt out;
    if (proper_inter(a,b,c,d,out)) return {out}; //if cross -> 1
    set<pt> s;
    if (on_segment(c,d,a)) s.insert(a); // a in cd
    if (on_segment(c,d,b)) s.insert(b); // b in cd
    if (on_segment(a,b,c)) s.insert(c); // c in ab
    if (on_segment(a,b,d)) s.insert(d); // d in ab
    return s; // 0, 2
}
lf pt_to_seg(pt a, pt b, pt p) { // p to ab
    if(a != b) {
        line l(a,b);
        if (l.cmp_proj(a,p) && l.cmp_proj(p,b)) /// if closest to  projection = (a, p, b)
            return l.dist(p); /// output distance to line
    }
    return min(abs(p-a), abs(p-b)); /// otherwise distance to A or B
```

```
}
lf seg_to_seg(pt a, pt b, pt c, pt d) {
  pt dummy;
  if (proper_inter(a,b,c,d,dummy)) return 0; // ab
      intersects cd
  return min({pt_to_seg(a,b,c), pt_to_seg(a,b,d),
      pt_to_seg(c,d,a), pt_to_seg(c,d,b)}); // try the 4
      pts
}
```

9.9 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 cololineales 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);
        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 Teoría y miscelánea

10.1 Sumatorias

- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$
- $\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$
- $\sum_{i=1}^{n} i^5 = \frac{(n(n+1))^2(2n^2+2n-1)}{12}$
- $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$ 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. Para varios componentes la formula es: $V - E + F = 1 + C$, siendo C el número de componentes.

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).

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 Geometría

10.4.1 Teorema de Pick

Sea un poligono simple cuyos vertices tienen coordenadas enteras. Si B es el numero de puntos enteros en el borde, I el numero de puntos enteros en el interior del poligono, entonces el area A del poligono se puede calcular con la formula:

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

10.4.2 Fórmula de Herón

Si los lados del triángulo tienen longitudes a, b y c, y s es el semiperímetro (es decir, $s = \frac{a+b+c}{2}$), entonces el área A del triángulo está dada por:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

10.4.3 Relación de Existencia Triangular

Para un triángulo con lados de longitud a, b, y c, la relación de existencia triangular se expresa como:

$$b - c < a < b + c, \quad a - c < b < a + c, \quad a - b < c < a + b$$

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 $P_{\text{rep}}(n; n_1, n_2, \ldots, n_k)$ y se calcula mediante:

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

10.5.4 Combinaciones con Repetición

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

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

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 x 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 poligono 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.

10.5.6 Estrellas y barras

Número de soluciones de la ecuación $x_1 + x_2 + \cdots + x_k = n$.

- Con $x_i \geq 0$: $\binom{n+k-1}{n}$
- Con $x_i \geq 1$: $\binom{n-1}{k-1}$

Número de sumas de enteros con límite inferior:

Esto se puede extender fácilmente a sumas de enteros con diferentes límites inferiores. Es decir, queremos contar el número de soluciones para la ecuación:

$$x_1 + x_2 + \cdots + x_k = n$$

con $x_i \geq a_i$.

Después de sustituir $x_i' := x_i - a_i$ recibimos la ecuación modificada:

$$(x_1' + a_i) + (x_2' + a_i) + \cdots + (x_k' + a_k) = n$$

$$\Leftrightarrow \quad x_1' + x_2' + \cdots + x_k' = n - a_1 - a_2 - \cdots - a_k$$

con $x_i' \geq 0$. Así que hemos reducido el problema al caso más simple con $x_i' \geq 0$ y nuevamente podemos aplicar el teorema de estrellas y barras.

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