Notebook

Para el regional elegimos nombre

6 de noviembre de 2025

Índice

1. Template

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

#define forr(i, a, b) for (int i = int(a); i < int(b); i++)
#define forn(i, n) forr(i,0,n)
#define dforr(i, a, b) for (int i = int(b)-1; i >= int(a); i--)
#define dforn(i, n) dforr(i,0,n)
#define all(v) begin(v),end(v)
#define sz(v) (int(size(v)))
#define pb push_back
#define fst first
#define snd second
#define mp make_pair
#define endl '\n'
#define dprint(v) cerr << __LINE__ << ": " #v " = " << v << endl

using ll = long long;
using pii = pair<int,int>;

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

1.1. run.sh

```sh
clear
make -s $1 &&
for CASE in ./cases/$1/*; do
    ./$1 < $CASE
done
```

1.2. comp.sh

```sh
clear
make -s $1 2>&1 | head -$2
```

1.3. Makefile

```
CXXFLAGS = -std=gnu++2a -O2 -g -Wall -Wextra -Wshadow -Wconversion\
-fsanitize=address -fsanitize=undefined
```

2. Estructuras de datos

2.1. Sparse Table

```
#define oper min
Elem st[K][1<<K]; // K tal que (1<<K) > n
void st_init(vector<Elem>& a) {
    int n = sz(a); // assert(K >= 31-__builtin_clz(2*n));
    forn(i,n) st[0][i] = a[i];
    forr(k,1,K) forn(i,n-(1<<k)+1)
        st[k][i] = oper(st[k-1][i], st[k-1][i+(1<<(k-1))]);
}
Elem st_query(int l, int r) { // assert(l<r);
    int k = 31-__builtin_clz(r-l);
    return oper(st[k][l], st[k][r-(1<<k)]);
}
// si la operacion no es idempotente
Elem st_query(int l, int r) {
    int k = 31-__builtin_clz(r-l);
    Elem res = st[k][l];
    for (l+=(1<<k), k--; l<r; k--) {
        if (l+(1<<k)<=r) {
            res = oper(res, st[k][l]);
            l += (1<<k);
        }
    }
    return res;
}
```

2.2. Segment Tree

```
// Dado un array y una operacion asociativa con neutro, get(i,j)
    opera en [i,j)
#define oper(x, y) max(x, y)
const int neutro=0;
struct RMQ{
    int sz;
    tipo t[4*MAXN];
    tipo &operator[](int p){return t[sz+p];}
    void init(int n){ // O(nlgn)
        sz = 1 << (32-__builtin_clz(n));
        forn(i, 2*sz) t[i]=neutro;
    }
    void updall(){dforn(i, sz) t[i]=oper(t[2*i], t[2*i+1]);} //
        O(N)
    tipo get(int i, int j){return get(i,j,1,0,sz);}
    tipo get(int i, int j, int n, int a, int b){ // O(lgn)
        if(j<=a || i>=b) return neutro;
        if(i<=a && b<=j) return t[n];
        int c=(a+b)/2;
        return oper(get(i, j, 2*n, a, c), get(i, j, 2*n+1, c, b));
    }
    void set(int p, tipo val){ // O(lgn)
        for(p+=sz; p>0 && t[p]!=val;){
            t[p]=val;
            p/=2;
            val=oper(t[p*2], t[p*2+1]);
        }
    }
}rmq;
// Usage:
cin >> n; rmq.init(n); forn(i, n) cin >> rmq[i]; rmq.updall();
```

2.3. Segment Tree Lazy

```
//Dado un arreglo y una operacion asociativa con neutro, get(i, j)
    opera sobre el rango [i, j).
typedef int Elem;//Elem de los elementos del arreglo
typedef int Alt;//Elem de la alteracion
#define oper(x,y) x+y
#define oper2(k,a,b) k*(b-a)//Aplicar actualizacion sobre [a, b)
const Elem neutro=0; const Alt neutro2=-1;
struct RMQ{
    int sz;
    Elem t[4*MAXN];
    Alt dirty[4*MAXN];//las alteraciones pueden ser distintas a
        Elem
    Elem &operator[](int p){return t[sz+p];}
    void init(int n){//O(nlgn)
        sz = 1 << (32-__builtin_clz(n));
        forn(i, 2*sz) t[i]=neutro;
        forn(i, 2*sz) dirty[i]=neutro2;
    }
    void push(int n, int a, int b){//propaga el dirty a sus hijos
        if(dirty[n]!=0){
            t[n]+=oper2(dirty[n], a, b);//altera el nodo
            if(n<sz){//cambiar segun el problema
                dirty[2*n]  = dirty[n];
                dirty[2*n+1] = dirty[n];
            }
            dirty[n]=0;
        }
    }
    Elem get(int i, int j, int n, int a, int b){//O(lgn)
        if(j<=a || i>=b) return neutro;
        push(n, a, b);
        if(i<=a && b<=j) return t[n];
        int c=(a+b)/2;
        return oper(get(i, j, 2*n, a, c), get(i, j, 2*n+1, c, b));
    }
    Elem get(int i, int j){return get(i,j,1,0,sz);}
    //altera los valores en [i, j) con una alteracion de val
    void alterar(Alt val,int i,int j,int n,int a,int b){//O(lgn)
        push(n, a, b);
        if(j<=a || i>=b) return;
        if(i<=a && b<=j){
            dirty[n]+=val;
            push(n, a, b);
            return;
        }
        int c=(a+b)/2;
        alterar(val, i, j, 2*n, a, c);
        alterar(val, i, j, 2*n+1, c, b);
        t[n]=oper(t[2*n], t[2*n+1]);
    }
    void alterar(Alt val, int i, int j){alterar(val,i,j,1,0,sz);}
}rmq;
```

2.4. Segment Tree 2D

```
int n,m;
int a[MAXN][MAXN],st[4*MAXN][4*MAXN];
void build(){
    forr(i,0,n)forr(j,0,m)st[i+n][j+m]=a[i][j];
    forr(i,0,n)for(int j=m-1;j;--j)
        st[i+n][j]=op(st[i+n][j<<1],st[i+n][j<<1|1]);
    for(int i=n-1;i;--i)forr(j,0,2*m)
        st[i][j]=op(st[i<<1][j],st[i<<1|1][j]);
}
void upd(int x, int y, int v){
    st[x+n][y+m]=v;
    for(int
        j=y+m;j>1;j>>=1)st[x+n][j>>1]=op(st[x+n][j],st[x+n][j^1]);
    for(int i=x+n;i>1;i>>=1)for(int j=y+m;j;j>>=1)
        st[i>>1][j]=op(st[i][j],st[i^1][j]);
}
int query(int x0, int x1, int y0, int y1){
    int r=NEUT;
    for(int i0=x0+n,i1=x1+n;i0<i1;i0>>=1,i1>>=1){
        int t[4],q=0;
        if(i0&1)t[q++]=i0++;
        if(i1&1)t[q++]=--i1;
        forr(k,0,q)for(int j0=y0+m,j1=y1+m;j0<j1;j0>>=1,j1>>=1){
            if(j0&1)r=op(r,st[t[k]][j0++]);
            if(j1&1)r=op(r,st[t[k]][--j1]);
        }
    }
    return r;
}
```

2.5. Segment Tree Persistente

```
const int LOG2N = 19; // ceil(log2(MAXN))
const int STLEN = 1<<LOG2N;

struct Mono {
    // TODO agregar data
    static Mono zero() { /* TODO */ } // neutro de la suma
```

```cpp
};
Mono operator+ (Mono a, Mono b) { /* TODO */ } // asociativo

struct N {
    N(Mono x_, N* l_, N* r_)
    : x{x_}, l{l_}, r{r_} {}
    Mono x; N* l; N* r;
};
N empty_node(Mono::zero(), &empty_node, &empty_node);

deque<N> st_alloc; // optimizacion: >30% mas rapido que 'new
    N(x,l,r)'
N* make_node(Mono x, N* l, N* r) {
    st_alloc.emplace_back(x, l, r);
    return &st_alloc.back();
}

N* u_(N* t, int l, int r, int i, Mono x) {
    if (i+1 <= l || r <= i) return t;
    if (r-l == 1) return make_node(x, nullptr, nullptr);
    int m = (l+r)/2;
    auto lt = u_(t->l, l, m, i, x);
    auto rt = u_(t->r, m, r, i, x);
    return make_node(lt->x + rt->x, lt, rt);
}

int ql, qr;
Mono q_(N* t, int l, int r) {
    if (qr <= l || r <= ql) return Mono::zero();
    if (ql <= l && r <= qr) return t->x;
    int m = (l+r)/2;
    return q_(t->l, l, m) + q_(t->r, m, r);
}

// suma en rango: t[l,r)
Mono query(N* t, int l, int r) { ql = l; qr = r; return q_(t, 0,
    STLEN); }

// asignacion en punto: t[i]=x
N* update(N* t, int i, Mono x) { return u_(t, 0, STLEN, i, x); }

/* uso:
auto t = &empty_node;
t = update(t, 0, Mono{10});
t = update(t, 5, Mono{5});
auto x = query(t, 0, 5); // devuelve Mono{10}
auto y = query(t, 0, 6); // devuelve Mono{10} + Mono{5}
auto z = query(t, 1, 6); // devuelve Mono{5}
*/
```

2.6. Fenwick Tree

```cpp
struct Fenwick { // 0-indexed, query [0, i), update [i]
    int ft[MAXN+1]; // Uso: ft.u(idx, val); cout << ft.q(idx);
    int u(int i0, int x) { for (int i=i0+1; i<=MAXN; i+=i&-i)
        ft[i]+=x; }
    ll q(int i0){ ll x=0; for (int i=i0; i>0; i-=i&-i) x+=ft[i];
        return x; } };

struct RangeFT { // 0-indexed, query [0, 1), update [l, r)
    Fenwick rate, err; // Uso: ft.u(l, r, val); cout << ft.q(l, r);
    void u(int l, int r, int x) { // range update
        rate.u(l, x); rate.u(r, -x); err.u(l, -x*l); err.u(r, x*r);
            }
    ll q(int i) { return rate.q(i) * i + err.q(i); } }; // prefix
        query
```

2.7. Treap

```cpp
// representa una lista como arbol con el orden implicito
struct node {
    int val, prio, tam;
    node *l, *r;
};
node *make(int val) {
    return new node { val, rand(), 1, nullptr, nullptr };
}
int tam(node *n) { return n ? n->tam : 0; }
void recalc(node *n) { n->tam = tam(n->l) + 1 + get_tam(n->r); }
node* merge(node* s, node* t) {
    if (s == nullptr) return t;
    if (t == nullptr) return s;
    if (s->prio > t->prio) {
        s->r = merge(s->r, t);
        recalc(s);
        return s;
    } else {
        t->l = merge(s, t->l);
        recalc(t);
        return t;
    }
}
pair<node*, node*> split(node *s, int k) {
    if (s == nullptr) return {nullptr, nullptr};
    if (tam(s->l) < k) {
        if (s->l == nullptr) return {nullptr, nullptr};
        auto [l, r] = split(s->r, k-tam(s->l)-1);
        s->r = l;
        recalc(s);
        return {s, r};
    } else {
        auto [l, r] = split(s->l, k);
        s->l = r;
        recalc(s);
        return {l, s};
    }
} // usage: node *list = nullptr; list = merge(list, make(5))
```

2.8. Union Find

```
vector<int> uf(MAXN, -1);
int uf_find(int x) { return uf[x]<0 ? x : uf[x] = uf_find(uf[x]); }
bool uf_join(int x, int y){ // True sii x e y estan en !=
    componentes
    x = uf_find(x); y = uf_find(y);
    if(x == y) return false;
    if(uf[x] > uf[y]) swap(x, y);
    uf[x] += uf[y]; uf[y] = x; return true;
}
```

2.9. Chull Trick

```
typedef ll tc;
struct Line{tc m,h;};
struct CHT { // for minimum (for maximum just change the sign of
    lines)
    vector<Line> c;
    int pos=0;
    tc in(Line a, Line b){
        tc x=b.h-a.h,y=a.m-b.m;
        return x/y+(x%y?!((x>0)^(y>0)):0); // ==ceil(x/y)
    }
    void add(tc m, tc h){ // m's should be non increasing
        Line l=(Line){m,h};
        if(sz(c)&&m==c.back().m){
            l.h=min(h,c.back().h);c.pop_back();if(pos)pos--;
        }
        while(sz(c)>1&&in(c.back(),l)<=in(c[sz(c)-2],c.back())){
            c.pop_back();if(pos)pos--;
        }
        c.pb(l);
    }
    inline bool fbin(tc x, int m){return in(c[m],c[m+1])>x;}
    tc eval(tc x){
        // O(log n) query:
        int s=0,e=c.size();
        while(e-s>1){int m=(s+e)/2;
            if(fbin(x,m-1))e=m;
            else s=m;
        }
        return c[s].m*x+c[s].h;
        // O(1) query (for ordered x's):
        while(pos>0&&fbin(x,pos-1))pos--;
        while(pos<c.size()-1&&!fbin(x,pos))pos++;
        return c[pos].m*x+c[pos].h;
    }
};
```

2.10. Chull Trick Dinámico

```
struct Entry {
    using It = set<Entry>::iterator;
    bool is_query;
    ll m, b; mutable It it, end;
    ll x;
};
bool operator< (Entry const& a, Entry const& b) {
    if (!b.is_query) return a.m < b.m;
    auto ni = next(a.it);
    if (ni == a.end) return false;
    auto const& c = *ni;
    return (c.b-a.b) > b.x * (a.m-c.m);
}
struct ChullTrick {
    using It = Entry::It;
    multiset<Entry> lines;
    bool covered(It it) {
        auto begin = lines.begin(), end = lines.end();
        auto ni = next(it);
        if (it == begin && ni == end) return false;
        if (it == begin) return ni->m==it->m && ni->b>=it->b;
        auto pi = prev(it);
        if (ni == end) return pi->m==it->m && pi->b>=it->b;
        return (it->m-pi->m)*(ni->b-pi->b) >=
            (pi->b-it->b)*(pi->m-ni->m);
    }
    bool add(ll m, ll b) {
        auto it = lines.insert({false, m, b});
        it->it = it; it->end = lines.end();
        if (covered(it)) { lines.erase(it); return false; }
        while (next(it) != lines.end() && covered(next(it)))
            lines.erase(next(it));
        while (it != lines.begin() && covered(prev(it)))
            lines.erase(prev(it));
        return true;
    }
    ll eval(ll x) {
        auto l = *lines.lower_bound({true, -1, -1, {}, {}, x});
        return l.m*x+l.b;
    }
};
```

3. Matemática

3.1. Criba Lineal

```
const int N = 10'000'000;
vector<int> lp(N+1);
vector<int> pr;
for (int i=2; i <= N; ++i) {
    if (lp[i] == 0) lp[i] = i, pr.push_back(i);
    for (int j = 0; i * pr[j] <= N; ++j) {
        lp[i * pr[j]] = pr[j];
        if (pr[j] == lp[i]) break;
    }
}
```

3.2. Phollard's Rho

```
ll mulmod(ll a, ll b, ll m) { return ll(__int128(a) * b % m); }

ll expmod(ll b, ll e, ll m) { // O(log b)
    if (!e) return 1;
    ll q=expmod(b,e/2,m); q=mulmod(q,q,m);
    return e%2 ? mulmod(b,q,m) : q;
}

bool es_primo_prob(ll n, int a) {
    if (n == a) return true;
    ll s = 0, d = n-1;
    while (d%2 == 0) s++, d/=2;
    ll x = expmod(a,d,n);
    if ((x == 1) || (x+1 == n)) return true;
    forn(i,s-1){
        x = mulmod(x,x,n);
        if (x == 1) return false;
        if (x+1 == n) return true;
    }
    return false;
}

bool rabin(ll n) { // devuelve true sii n es primo
```

```cpp
    if (n == 1) return false;
    const int ar[] = {2,3,5,7,11,13,17,19,23};
    forn(j,9) if (!es_primo_prob(n,ar[j])) return false;
    return true;
}

ll rho(ll n) {
    if ((n & 1) == 0) return 2;
    ll x = 2, y = 2, d = 1;
    ll c = rand() % n + 1;
    while (d == 1) {
        x = (mulmod(x,x,n)+c)%n;
        y = (mulmod(y,y,n)+c)%n;
        y = (mulmod(y,y,n)+c)%n;
        d=gcd(x-y,n);
    }
    return d==n ? rho(n) : d;
}

void factRho(map<ll,ll>&prim, ll n){ //O (lg n)^3. un solo numero
    if (n == 1) return;
    if (rabin(n)) { prim[n]++; return; }
    ll factor = rho(n);
    factRho(factor, prim); factRho(n/factor, prim);
}
auto fact(ll n){
    map<ll,ll>prim;
    factRho(prim,n);
    return prim;
}
```

3.3. Divisores

```cpp
// Usar asi: divisores(fac, divs, fac.begin()); NO ESTA ORDENADO
void divisores(const map<ll,ll> &f, vector<ll> &divs, auto it, ll
    n=1){
    if (it==f.begin()) divs.clear();
    if (it==f.end()) { divs.pb(n); return; }
    ll p=it->fst, k=it->snd; ++it;
    forn(_, k+1) divisores(f,divs,it,n), n*=p;
}

ll sumDiv (ll n){ //suma de los divisores de n
  ll rta = 1;
  map<ll,ll> f=fact(n);
  for(auto it = f.begin(); it != f.end(); it++) {
    ll pot = 1, aux = 0;
    forn(i, it->snd+1) aux += pot, pot *= it->fst;
    rta*=aux;
  }
  return rta;
}
```

3.4. Inversos Modulares

```cpp
pair<ll,ll> extended_euclid(ll a, ll b) {
    if (b == 0) return {1, 0};
    auto [y, x] = extended_euclid(b, a%b);
    y -= (a/b)*x;
    if (a*x + b*y < 0) x = -x, y = -y;
    return {x, y}; // a*x + b*y = gcd(a,b)
}
```

```cpp
constexpr ll MOD = 1000000007; // tmb es comun 998'244'353
ll invmod[MAXN]; // inversos módulo MOD hasta MAXN
void invmods() { // todo entero en [2,MAXN] debe ser coprimo con
    MOD
    inv[1] = 1;
    forr(i, 2, MAXN) inv[i] = MOD - MOD/i*inv[MOD%i]%MOD;
}

// si MAXN es demasiado grande o MOD no es fijo:
// versión corta, m debe ser primo. O(log(m))
ll invmod(ll a, ll m) { return expmod(a,m-2,m); }
// versión larga, a y m deben ser coprimos. O(log(a)), en general
    más rápido
ll invmod(ll a, ll m) { return (extended_euclid(a,m).fst % m + m)
    % m; }
```

3.5. Catalan

```cpp
ll Cat(int n){
    return ((F[2*n] *FI[n+1])%M *FI[n])%M;
}
```

3.6. Lucas

```cpp
const ll MAXP = 3e3+10; //68 MB, con 1e4 int son 380 MB
ll C[MAXP][MAXP], P;    //inicializar con el primo del input < MAXP
void llenar_C(){
    forn(i, MAXP) C[i][0] = 1;
    forr(i, 1, MAXP) forr(j, 1, i+1)
        C[i][j]=addmod(C[i-1][j-1],C[i-1][j], P);
}
// Calcula nCk (mod p) con n, k arbitrariamente grandes y p primo
    <= 3000
ll lucas(ll N, ll K){ // llamar a llenar_C() antes
    ll ret = 1;
    while(N+K){
        ret = ret * C[N%P][K%P] % P;
        N /= P, K /= P;
    }
    return ret;
}
```

3.7. Stirling-Bell

```cpp
ll STR[MAXN][MAXN], Bell[MAXN];
//STR[n][k] = formas de particionar un conjunto de n elementos en
    k conjuntos
//Bell[n] = formas de particionar un conjunto de n elementos
forr(i, 1, MAXN)STR[i][1] = 1;
forr(i, 2, MAXN)STR[1][i] = 0;
forr(i, 2, MAXN)forr(j, 2, MAXN){
    STR[i][j] = (STR[i-1][j-1] + j*STR[i-1][j]%MOD) %MOD;
}
forn(i, MAXN){
    Bell[i] = 0;
    forn(j, MAXN){
        Bell[i] = (Bell[i] + STR[i][j]) %MOD;
    }
}
```

3.8. DP Factoriales

```cpp
ll F[MAXN], INV[MAXN], FI[MAXN];
// ...
F[0] = 1; forr(i, 1, MAXN) F[i] = F[i-1]*i %M;
INV[1] = 1; forr(i, 2, MAXN) INV[i] = M - (ll)(M/i)*INV[M%i]%M;
FI[0] = 1; forr(i, 1, MAXN) FI[i] = FI[i-1]*INV[i] %M;
```

3.9. Estructura de Fracción

```cpp
tipo mcd(tipo a, tipo b){return a?mcd(b%a, a):b;}
struct frac{
    tipo p,q;
    frac(tipo p=0, tipo q=1):p(p),q(q) {norm();}
    void norm(){
        tipo a = mcd(p,q);
        if(a) p/=a, q/=a;
        else q=1;
        if (q<0) q=-q, p=-p;}
    frac operator+(const frac& o){
        tipo a = mcd(q,o.q);
        return frac(p*(o.q/a)+o.p*(q/a), q*(o.q/a));}
    frac operator-(const frac& o){
        tipo a = mcd(q,o.q);
        return frac(p*(o.q/a)-o.p*(q/a), q*(o.q/a));}
    frac operator*(frac o){
        tipo a = mcd(q,o.p), b = mcd(o.q,p);
        return frac((p/b)*(o.p/a), (q/a)*(o.q/b));}
    frac operator/(frac o){
        tipo a = mcd(q,o.q), b = mcd(o.p,p);
        return frac((p/b)*(o.q/a),(q/a)*(o.p/b));}
    bool operator<(const frac &o) const{return p*o.q < o.p*q;}
    bool operator==(frac o){return p==o.p&&q==o.q;}
};
```

3.10. Gauss

```cpp
double reduce(vector<vector<double>> &a){ //Devuelve determinante
    si m == n
    int m=sz(a), n=sz(a[0]), i=0, j=0; double r = 1.0;
    while(i < m and j < n){
      int h = i;
      forr(k, i+1, m) if(abs(a[k][j]) > abs(a[h][j])) h = k;
      if(abs(a[h][j]) < EPS){ j++; r=0.0; continue; }
      if(h != i){ r = -r; swap(a[i], a[h]); }
      r *= a[i][j];
      dforr(k, j, n) a[i][k] /= a[i][j];
      forr(k, 0, m) if(k != i)
         dforr(l_, j, n) a[k][l_] -= a[k][j] * a[i][l_];
      i ++; j ++;
   }
   return r;
}
```

3.11. FFT

```
// MAXN must be power of 2 !!, MOD-1 needs to be a multiple of
    MAXN !!
typedef ll tf;
typedef vector<tf> poly;
//const tf MOD = 2305843009255636993, RT = 5;
const tf MOD = 998244353, RT = 3;
// const tf MOD2 = 897581057, RT2 = 3; // Chinese Remainder Theorem
/* FFT */ struct CD {
   double r, i;
   CD(double r_ = 0, double i_ = 0) : r(r_), i(i_) {}
   void operator/=(const int c) { r/=c, i/=c; }
};
CD operator*(const CD& a, const CD& b){
   return CD(a.r*b.r-a.i*b.i, a.r*b.i+a.i*b.r);}
CD operator+(const CD& a, const CD& b) { return CD(a.r+b.r,
    a.i+b.i); }
CD operator-(const CD& a, const CD& b) { return CD(a.r-b.r,
    a.i-b.i); }
/* NTT */ struct CD { tf x; CD(tf x_) : x(x_) {} CD() {}  };
CD operator+(const CD& a, const CD& b) { return CD(addmod(a.x,
    b.x)); }//ETC
vector<tf> rts(MAXN+9,-1);
CD root(int n, bool inv){
   tf r = rts[n]<0 ? rts[n] = expmod(RT,(MOD-1)/n) : rts[n];
   return CD(inv ? expmod(r, MOD-2) : r);
}
/* AMBOS */ CD cp1[MAXN+9], cp2[MAXN+9];
int R[MAXN+9];
void dft(CD* a, int n, bool inv){
   double pi = acos(-1.0);
   forn(i, n) if(R[i] < i) swap(a[R[i]], a[i]);
   for(int m = 2; m <= n; m *= 2){
      /* FFT */ double z = 2*pi/m * (inv?-1:1);
      /* FFT */ CD wi = CD(cos(z), sin(z));
      /* NTT */ CD wi = root(m, inv);
      for(int j = 0; j < n; j += m){
         CD w(1);
         for(int k = j, k2 = j+m/2; k2 < j+m; k++, k2++){
            CD u = a[k]; CD v = a[k2]*w; a[k] = u+v; a[k2] =
                u-v; w = w*wi;
         }
      }
   }
   /* FFT */ if(inv) forn(i, n) a[i] /= n;
   /* NTT */ if(inv){
      CD z(expmod(n, MOD-2));
      forn(i, n) a[i] = a[i]*z;
   }
}
poly multiply(poly& p1, poly& p2){
   int n = sz(p1)+sz(p2)+1;
   int m = 1, cnt = 0;
   while(m <= n) m *= 2, cnt ++;
   forn(i, m) { R[i] = 0; forn(j, cnt) R[i] =
       (R[i]<<1)|((i>>j)&1); }
   forn(i, m) cp1[i] = 0, cp2[i] = 0;
   forn(i, sz(p1)) cp1[i] = p1[i];
   forn(i, sz(p2)) cp2[i] = p2[i];
   dft(cp1, m, false); dft(cp2, m, false);
   // fast eval: forn(i, sz(p1)) p1(expmod(RT, (MOD-1)/m*i)) ==
       cp1[i].x
   forn(i, m) cp1[i] = cp1[i]*cp2[i];
   dft(cp1, m, true);
   poly res;
   n -= 2;
```

```
    /* FFT */ forn(i, n) res.pb((tf)floor(cp1[i].r+0.5));
    /* NTT */ forn(i, n)res.pb(cp1[i].x);
    return res;
}
```

4. Geometria

4.1. Punto

```
using T = double;
bool iszero(T u) { return abs(u)<=EPS; }
struct Pt {
    T x, y;
    T z; // only for 3d
    Pt() {}
    Pt(T _x, T _y) : x(_x), y(_y) {}
    Pt(T _x, T _y, T _z) : x(_x), y(_y), z(_z) {} // for 3d
    T norm2(){ return *this**this; }
    T norm(){ return sqrt(norm2()); }
    Pt operator+(Pt o){ return Pt(x+o.x,y+o.y); }
    Pt operator-(Pt o){ return Pt(x-o.x,y-o.y); }
    Pt operator*(T u){ return Pt(x*u,y*u); }
    Pt operator/(T u) {
        if (iszero(u)) return Pt(INF,INF);
        return Pt(x/u,y/u);
    }
    T operator*(Pt o){ return x*o.x+y*o.y; }
    Pt operator^(Pt p){ // only for 3D
        return Pt(y*p.z-z*p.y, z*p.x-x*p.z, x*p.y-y*p.x); }
    T operator%(Pt o){ return x*o.y-y*o.x; }
    T angle(Pt o){ return atan2(*this%o, *this*o); }
//  T angle(Pt o){ // accurate around 90 degrees
//      if (*this%o>0) return acos(*this*o);
//      return 2*M_PI-acos(*this*o); }
    Pt unit(){ return *this/norm(); }
    bool left(Pt p, Pt q){ // is it to the left of directed line
        pq?
        return ((q-p)%(*this-p))>EPS; }
    bool operator<(Pt p)const{ // for convex hull
        return x<p.x-EPS||(iszero(x-p.x)&&y<p.y-EPS); }
    bool collinear(Pt p, Pt q){
        return iszero((p-*this)%(q-*this)); }
    bool dir(Pt p, Pt q){ // does it have the same direction of pq?
        return this->collinear(p, q)&&(q-p)*(*this-p)>EPS; }
    Pt rot(Pt r){ return Pt(*this%r,*this*r); }
    Pt rot(T a){ return rot(Pt(sin(a),cos(a))); }
};
Pt ccw90(1,0);
Pt cw90(-1,0);
```

4.2. Linea

```
using T = double;
int sgn2(T x){return x<0?-1:1;}
struct Ln {
    Pt p,pq;
    Ln(Pt p, Pt q):p(p),pq(q-p){}
    Ln(){}
    bool has(Pt r){return dist(r)<=EPS;}
    bool seghas(Pt r){return has(r)&&(r-p)*(r-(p+pq))<=EPS;}
//  bool operator /(Ln l){return
     (pq.unit()^l.pq.unit()).norm()<=EPS;} // 3D
    bool operator/(Ln l){return abs(pq.unit()^l.pq.unit())<=EPS;}
        // 2D
    bool operator==(Ln l){return *this/l&&has(l.p);}
    Pt operator^(Ln l){ // intersection
        if(*this/l)return Pt(INF,INF);
        T a=-pq.y, b=pq.x, c=p.x*a+p.y*b;
        T la=-l.pq.y, lb=l.pq.x, lc=l.p.x*la+l.p.y*lb;
        T det = a * lb - b * la;
        Pt r((lb*c-b*lc)/det, (a*lc-c*la)/det);
        return r;
//      Pt r=l.p+l.pq*(((p-l.p)^pq)/(l.pq^pq));
//      if(!has(r)){return Pt(NAN,NAN,NAN);} // check only for 3D
    }
    T angle(Ln l){return pq.angle(l.pq);}
    int side(Pt r){return has(r)?0:sgn2(pq^(r-p));} // 2D
    Pt proj(Pt r){return p+pq*((r-p)*pq/pq.norm2());}
    Pt segclosest(Pt r) {
       T l2 = pq.norm2();
```

```
        if(l2==0.) return p;
        T t =((r-p)*pq)/l2;
        return p+(pq*min(1,max(0,t)));
    }
    Pt ref(Pt r){return proj(r)*2-r;}
    T dist(Pt r){return (r-proj(r)).norm();}
//  T dist(Ln l){ // only 3D
//      if(*this/l)return dist(l.p);
//      return abs((l.p-p)*(pq^l.pq))/(pq^l.pq).norm();
//  }
    Ln rot(auto a){return Ln(p,p+pq.rot(a));} // 2D
};
Ln bisector(Ln l, Ln m){ // angle bisector
    Pt p=l^m;
    return Ln(p,p+l.pq.unit()+m.pq.unit());
}
Ln bisector(Pt p, Pt q){ // segment bisector (2D)
    return Ln((p+q)*.5,p).rot(ccw90);
}
```

4.3. Poligono

```
using T = double;
struct Pol {
    int n;vector<Pt> p;
    Pol(){}
    Pol(vector<Pt> _p){p=_p;n=p.size();}
    T area() {
        ll a = 0;
        forr (i, 1, sz(p)-1) {
            a += (p[i]-p[0])^(p[i+1]-p[0]);
        }
        return abs(a)/2;
    }
    bool has(Pt q){ // O(n), winding number
        forr(i,0,n)if(Ln(p[i],p[(i+1)%n]).seghas(q))return true;
        int cnt=0;
        forr(i,0,n){
            int j=(i+1)%n;
            int k=sgn((q-p[j])^(p[i]-p[j]));
            int u=sgn(p[i].y-q.y),v=sgn(p[j].y-q.y);
            if(k>0&&u<0&&v>=0)cnt++;
            if(k<0&&v<0&&u>=0)cnt--;
        }
        return cnt!=0;
    }
    void normalize(){ // (call before haslog, remove collinear
        first)
        if(n>=3&&p[2].left(p[0],p[1]))reverse(p.begin(),p.end());
        int pi=min_element(p.begin(),p.end())-p.begin();
        vector<Pt> s(n);
        forr(i,0,n)s[i]=p[(pi+i)%n];
        p.swap(s);
    }
    bool haslog(Pt q){ // O(log(n)) only CONVEX. Call normalize
        first
        if(q.left(p[0],p[1])||q.left(p.back(),p[0]))return false;
        int a=1,b=p.size()-1; // returns true if point on boundary
        while(b-a>1){       // (change sign of EPS in left
            int c=(a+b)/2;     // to return false in such case)
            if(!q.left(p[0],p[c]))a=c;
            else b=c;
        }
        return !q.left(p[a],p[a+1]);
    }
    bool isconvex(){//O(N), delete collinear points!
        if(n<3) return false;
        bool isLeft=p[0].left(p[1], p[2]);
        forr(i, 1, n)
            if(p[i].left(p[(i+1)%n], p[(i+2)%n])!=isLeft)
                return false;
        return true;
    }
    Pt farthest(Pt v){ // O(log(n)) only CONVEX
        if(n<10){
            int k=0;
            forr(i,1,n)if(v*(p[i]-p[k])>EPS)k=i;
            return p[k];
        }
        if(n==sz(p))p.pb(p[0]);
        Pt a=p[1]-p[0];
        int s=0,e=n,ua=v*a>EPS;
        if(!ua&&v*(p[n-1]-p[0])<=EPS)return p[0];
        while(1){
            int m=(s+e)/2;Pt c=p[m+1]-p[m];
            int uc=v*c>EPS;
            if(!uc&&v*(p[m-1]-p[m])<=EPS)return p[m];
            if(ua&&(!uc||v*(p[s]-p[m])>EPS))e=m;
            else if(ua||uc||v*(p[s]-p[m])>=-EPS)s=m,a=c,ua=uc;
            else e=m;
            assert(e>s+1);
        }
    }
    Pol cut(Ln l){ // cut CONVEX polygon by line l
        vector<Pt> q; // returns part at left of l.pq
        forr(i,0,n){
            int d0=sgn(l.pq^(p[i]-l.p)),d1=sgn(l.pq^(p[(i+1)%n]-l.p));
            if(d0>=0)q.pb(p[i]);
            Ln m(p[i],p[(i+1)%n]);
            if(d0*d1<0&&!(l/m))q.pb(l^m);
        }
        return Pol(q);
    }
    T intercircle(circle c){ // area of intersection with circle
        T r=0.;
        forr(i,0,n){
            int j=(i+1)%n;T w=c.intertriangle(p[i],p[j]);
            if((p[j]-c.o)^(p[i]-c.o)>EPS)r+=w;
            else r-=w;
        }
        return abs(r);
    }
    T callipers(){ // square distance of most distant points
        T r=0;   // prereq: convex, ccw, NO COLLINEAR POINTS
        for(int i=0,j=n<2?0:1;i<j;++i){
            for(;;j=(j+1)%n){
                r=max(r,(p[i]-p[j]).norm2());
                if(((p[(i+1)%n]-p[i])^(p[(j+1)%n]-p[j]))<=EPS)break;
            }
        }
        return r;
    }
};
```

4.4. Circulo

```
using T = double;
struct Circle {
    Pt o;T r;
    Circle(Pt o, T r):o(o),r(r){}
    Circle(Pt x, Pt y, Pt z){o=bisector(x,y)^bisector(x,z);r=(o-x).norm();}
    bool has(Pt p){return (o-p).norm()<=r+EPS;}
    vector<Pt> operator^(Circle c){ // ccw
        vector<Pt> s;
        T d=(o-c.o).norm();
        if(d>r+c.r+EPS||d+min(r,c.r)+EPS<max(r,c.r))return s;
        T x=(d*d-c.r*c.r+r*r)/(2*d);
        T y=sqrt(r*r-x*x);
        Pt v=(c.o-o)/d;
        s.pb(o+v*x-v.rot(ccw90)*y);
        if(y>EPS)s.pb(o+v*x+v.rot(ccw90)*y);
        return s;
    }
    vector<Pt> operator^(Ln l){
        vector<Pt> s;
        Pt p=l.proj(o);
        T d=(p-o).norm();
        if(d-EPS>r)return s;
        if(abs(d-r)<=EPS){s.pb(p);return s;}
        d=sqrt(r*r-d*d);
        s.pb(p+l.pq.unit()*d);
        s.pb(p-l.pq.unit()*d);
        return s;
    }
    vector<Pt> tang(Pt p){
        T d=sqrt((p-o).norm2()-r*r);
        return *this^Circle(p,d);
    }
    bool in(Circle c){ // non strict
        T d=(o-c.o).norm();
        return d+r<=c.r+EPS;
    }
    T intertriangle(Pt a, Pt b){ // area of intersection with oab
        if(abs((o-a)%(o-b))<=EPS)return 0.;
        vector<Pt> q={a},w=*this^Ln(a,b);
        if(w.size()==2)for(auto p:w)if((a-p)*(b-p)<-EPS)q.pb(p);
        q.pb(b);
        if(q.size()==4&&(q[0]-q[1])*(q[2]-q[1])>EPS)swap(q[1],q[2]);
        T s=0;
        fore(i,0,q.size()-1){
            if(!has(q[i])||!has(q[i+1]))s+=r*r*(q[i]-o).angle(q[i+1]-o)/2;
            else s+=abs((q[i]-o)%(q[i+1]-o)/2);
        }
        return s;
    }
};
```

4.5. Convex Hull

```
// CCW order
// Includes collinear points (change sign of EPS in left to exclude)
vector<Pt> chull(vector<Pt> p){
    if(sz(p)<3)return p;
    vector<Pt> r;
    sort(p.begin(),p.end()); // first x, then y
    forr(i,0,p.size()){ // lower hull
        while(r.size()>=2&&r.back().left(r[r.size()-2],p[i]))r.pop_back();
        r.pb(p[i]);
    }
    r.pop_back();
    int k=r.size();
    for(int i=p.size()-1;i>=0;--i){ // upper hull
        while(r.size()>=k+2&&r.back().left(r[r.size()-2],p[i]))r.pop_back();
        r.pb(p[i]);
    }
    r.pop_back();
    return r;
}
```

4.6. Orden Radial

```
struct Radial {
    Pt o;
    Radial(Pt _o) : o(_o) {}
    int cuad(Pt p) {
        if (p.x>0 && p.y>=0) return 1;
        if (p.x<=0 && p.y>0) return 2;
        if (p.x<0 && p.y<=0) return 3;
        if (p.x>=0 && p.y<0) return 4;
        assert(p.x == 0 && p.y == 0);
        return 0; // origen < todos
    }
    bool comp(Pt p, Pt q) {
        int c1 = cuad(p), c2 = cuad(q);
        if (c1 == c2) return p%q>EPS;
        return c1 < c2;
    }
    bool operator()(const Pt &p, const Pt &q) const {
        return comp(p-o,q-o);
    }
};
```

4.7. Par de puntos más cercano

```
#define dist(a, b) ((a-b).norm_sq())
bool sortx(pt a, pt b) {
    return mp(a.x,a.y)<mp(b.x,b.y); }
bool sorty(pt a, pt b) {
    return mp(a.y,a.x)<mp(b.y,b.x); }
ll closest(vector<pt> &ps, int l, int r) {
    if (l == r-1) return INF;
    if (l == r-2) {
        if (sorty(ps[l+1], ps[l]))
            swap(ps[l+1], ps[l]);
        return dist(ps[l], ps[l+1]);
    }
    int m = (l+r)/2; ll xm = ps[m].x;
    ll min_dist = min(closest(ps, l, m),closest(ps, m, r));
    vector<pt> left(&ps[l], &ps[m]), right(&ps[m], &ps[r]);
    merge(all(left), all(right), &ps[l], sorty);
    ll delta = ll(sqrt(min_dist));
    vector<pt> strip;
    forr (i, l, r) if (ps[i].x>=xm-delta&&ps[i].x<=xm+delta)
        strip.pb(ps[i]);
    forn (i, sz(strip)) forr (j, 1, 8) {
        if (i+j >= sz(strip)) break;
        min_dist = min(min_dist, dist(strip[i], strip[i+j]));
    }
    return min_dist;
}
ll closest(vector<pt> &ps) { // devuelve dist^2
    sort(all(ps), sortx);
    return closest(ps, 0, sz(ps));
}
```

4.8. Arbol KD

```
// given a set of points, answer queries of nearest point in
    O(log(n))
bool onx(pt a, pt b){return a.x<b.x;}
bool ony(pt a, pt b){return a.y<b.y;}
struct Node {
    pt pp;
    ll x0=INF, x1=-INF, y0=INF, y1=-INF;
    Node *first=0, *second=0;
    ll distance(pt p){
        ll x=min(max(x0,p.x),x1);
        ll y=min(max(y0,p.y),y1);
        return (pt(x,y)-p).norm2();
    }
    Node(vector<pt>&& vp):pp(vp[0]){
        for(pt p:vp){
            x0=min(x0,p.x); x1=max(x1,p.x);
            y0=min(y0,p.y); y1=max(y1,p.y);
        }
        if(sz(vp)>1){
            sort(all(vp),x1-x0>=y1-y0?onx:ony);
            int m=sz(vp)/2;
            first=new Node({vp.begin(),vp.begin()+m});
            second=new Node({vp.begin()+m,vp.end()});
        }
    }
};
struct KDTree {
    Node* root;
    KDTree(const vector<pt>& vp):root(new Node({all(vp)})) {}
    pair<ll,pt> search(pt p, Node *node){
        if(!node->first){
            //avoid query point as answer
            //if(p==node->pp) {INF,pt()};
            return {(p-node->pp).norm2(),node->pp};
        }
        Node *f=node->first, *s=node->second;
        ll bf=f->distance(p), bs=s->distance(p);
        if(bf>bs)swap(bf,bs),swap(f,s);
        auto best=search(p,f);
        if(bs<best.fst) best=min(best,search(p,s));
        return best;
    }
    pair<ll,pt> nearest(pt p){return search(p,root);}
};
```

4.9. Suma de Minkowski

```
vector<Pt> minkowski_sum(vector<Pt> &p, vector<Pt> &q){
    int n=sz(p),m=sz(q),x=0,y=0;
    forr(i,0,n) if(p[i]<p[x]) x=i;
    forr(i,0,m) if(q[i]<q[y]) y=i;
    vector<Pt> ans={p[x]+q[y]};
    forr(it,1,n+m){
        Pt a=p[(x+1)%n]+q[y];
        Pt b=p[x]+q[(y+1)%m];
        if(b.left(ans.back(),a)) ans.pb(b), y=(y+1)%m;
        else ans.pb(a), x=(x+1)%n;
    }
    return ans;
}
vector<Pt> do_minkowski(vector<Pt> &p, vector<Pt> &q) {
```

```cpp
    normalize(p); normalize(q);
    vector<Pt> sum = minkowski_sum(p, q);
    return chull(sum); // no normalizado
}
// escalar poligono
vector<Pt> operator*(vector<Pt> &p, td u) {
    vector<Pt> r; forn (i, sz(p)) r.pb(p[i]*u);
    return r;
}
```

5. Strings

5.1. Hashing

```cpp
struct StrHash { // Hash polinomial con exponentes decrecientes.
    static constexpr ll ms[] = {1'000'000'007, 1'000'000'403};
    static constexpr ll b = 500'000'000;
    vector<ll> hs[2], bs[2];
    StrHash(string const& s) {
        int n = sz(s);
        forn(k, 2) {
            hs[k].resize(n+1), bs[k].resize(n+1, 1);
            forn(i, n) {
                hs[k][i+1] = (hs[k][i] * b + s[i]) % ms[k];
                bs[k][i+1] = bs[k][i] * b     % ms[k];
            }
        }
    }
    ll get(int idx, int len) const { // Hashes en `s[idx,
        idx+len)`.
        ll h[2];
        forn(k, 2) {
            h[k] = hs[k][idx+len] - hs[k][idx] * bs[k][len] % ms[k];
            if (h[k] < 0) h[k] += ms[k];
        }
        return (h[0] << 32) | h[1];
    }
};
```

5.2. Suffix Array

```cpp
#define RB(x) ((x) < n ? r[x] : 0)
void csort(vector<int>& sa, vector<int>& r, int k) {
    int n = sz(sa);
    vector<int> f(max(255, n)), t(n);
    forn(i, n) ++f[RB(i+k)];
    int sum = 0;
    forn(i, max(255, n)) f[i] = (sum += f[i]) - f[i];
    forn(i, n) t[f[RB(sa[i]+k)]++] = sa[i];
    sa = t;
}
vector<int> compute_sa(string& s){ // O(n*log2(n))
    int n = sz(s) + 1, rank;
    vector<int> sa(n), r(n), t(n);
    iota(all(sa), 0);
    forn(i, n) r[i] = s[i];
    for (int k = 1; k < n; k *= 2) {
        csort(sa, r, k), csort(sa, r, 0);
        t[sa[0]] = rank = 0;
        forr(i, 1, n) {
            if(r[sa[i]] != r[sa[i-1]] || RB(sa[i]+k) !=
                RB(sa[i-1]+k)) ++rank;
            t[sa[i]] = rank;
        }
        r = t;
        if (r[sa[n-1]] == n-1) break;
    }
    return sa; // sa[i] = i-th suffix of s in lexicographical order
}
vector<int> compute_lcp(string& s, vector<int>& sa){
    int n = sz(s) + 1, L = 0;
    vector<int> lcp(n), plcp(n), phi(n);
    phi[sa[0]] = -1;
    forr(i, 1, n) phi[sa[i]] = sa[i-1];
    forn(i,n) {
        if (phi[i] < 0) { plcp[i] = 0; continue; }
        while(s[i+L] == s[phi[i]+L]) ++L;
        plcp[i] = L;
        L = max(L - 1, 0);
    }
    forn(i, n) lcp[i] = plcp[sa[i]];
    return lcp; // lcp[i] = longest common prefix between sa[i-1]
        and sa[i]
}
```

5.3. String Functions

```cpp
template<class Char=char>vector<int> pfun(basic_string<Char>const&
     w) {
    int n = sz(w), j = 0; vector<int> pi(n);
    forr(i, 1, n) {
        while (j != 0 && w[i] != w[j]) {j = pi[j - 1];}
        if (w[i] == w[j]) {++j;}
        pi[i] = j;
    } // pi[i] = lengh of longest proper suffix of w[0..i] that is
        also prefix
    return pi;
}
template<class Char=char>vector<int> zfun(const
     basic_string<Char>& w) {
    int n = sz(w), l = 0, r = 0; vector<int> z(n);
    forr(i, 1, n) {
        if (i <= r) {z[i] = min(r - i + 1, z[i - l]);}
        while (i + z[i] < n && w[z[i]] == w[i + z[i]]) {++z[i];}
        if (i + z[i] - 1 > r) {l = i, r = i + z[i] - 1;}
    } // z[i] = lengh of longest prefix of w that also begins at
        index i
    return z;
}
```

5.4. Kmp

```cpp
template<class Char=char>struct Kmp {
    using str = basic_string<Char>;
    vector<int> pi; str pat;
    Kmp(str const& _pat): pi(move(pfun(_pat))), pat(_pat) {}
    vector<int> matches(str const& txt) const {
        if (sz(pat) > sz(txt)) {return {};}
        vector<int> occs; int m = sz(pat), n = sz(txt);
        if (m == 0) {occs.push_back(0);}
        int j = 0;
        forn(i, n) {
            while (j != 0 && txt[i] != pat[j]) {j = pi[j-1];}
            if (txt[i] == pat[j]) {++j;}
            if (j == m) {occs.push_back(i - j + 1);}
        }
        return occs;
    }
};
```

5.5. Manacher

```cpp
struct Manacher {
    vector<int> p;
    Manacher(string const& s) {
        int n = sz(s), m = 2*n+1, l = -1, r = 1;
        vector<char> t(m); forn(i, n) t[2*i+1] = s[i];
        p.resize(m); forr(i, 1, m) {
            if (i < r) p[i] = min(r-i, p[l+r-i]);
            while (p[i] <= i && i < m-p[i] && t[i-p[i]] ==
                 t[i+p[i]]) ++p[i];
            if (i+p[i] > r) l = i-p[i], r = i+p[i];
        }
    } // Retorna palindromos de la forma {comienzo, largo}.
    pii at(int i) const {int k = p[i]-1; return pair{i/2-k/2, k};}
    pii odd(int i) const {return at(2*i+1);} // Mayor centrado en
        s[i].
    pii even(int i) const {return at(2*i);} // Mayor centrado en
        s[i-1,i].
};
```

5.6. Mínima Rotación Lexicográfica

```cpp
// única secuencia no-creciente de strings menores a sus rotaciones
vector<pii> lyndon(string const& s) {
    vector<pii> fs;
    int n = sz(s);
    for (int i = 0, j, k; i < n;) {
        for (k = i, j = i+1; j < n && s[k] <= s[j]; ++j)
            if (s[k] < s[j]) k = i; else ++k;
        for (int m = j-k; i <= k; i += m) fs.emplace_back(i, m);
    }
    return fs; // retorna substrings de la forma {comienzo, largo}
}

// último comienzo de la mínima rotación
int minrot(string const& s) {
    auto fs = lyndon(s+s);
    int n = sz(s), start = 0;
    for (auto f : fs) if (f.fst < n) start = f.fst; else break;
    return start;
}
```

5.7. Trie

```cpp
// trie genérico. si es muy lento, se puede modificar para que los
    hijos sean
// representados con un array del tamaño del alfabeto
template<class Char> struct Trie {
    struct Node {
        map<Char, Node*> child;
        bool term;
    };
    Node* root;
    static inline deque<Node> nodes;
    static Node* make() {
        nodes.emplace_back();
        return &nodes.back();
    }
    Trie() : root{make()} {}
    // retorna el largo del mayor prefijo de s que es prefijo de
        algún string
    // insertado en el trie
    int find(basic_string<Char> const& s) const {
        Node* curr = root;
        forn(i,sz(s)) {
            auto it = curr->child.find(s[i]);
            if (it == end(curr->child)) return i;
            curr = it->snd;
        }
        return sz(s);
    }
    // inserta s en el trie
    void insert(basic_string<Char> const& s) {
        Node* curr = root;
        forn(i,sz(s)) {
            auto it = curr->child.find(s[i]);
            if (it == end(curr->child)) curr = curr->child[s[i]] =
                make();
            else curr = it->snd;
        }
        curr->term = true;
    }
    // elimina s del trie
    void erase(basic_string<Char> const& s) {
        auto erase = [&](auto&& me, Node* curr, int i) -> bool {
            if (i == sz(s)) {
                curr->term = false;
                return sz(curr->child) == 0;
            }
            auto it = curr->child.find(s[i]);
            if (it == end(curr->child)) return false;
            if (!me(me,it->snd,i+1)) return false;
            curr->child.erase(it);
            return sz(curr->child) == 0;
        };
        erase(erase,root,0);
    }
};
```

5.8. Suffix Automaton

```cpp
/// Minimal DFA that accepts all suffixes of a string.
/// - Any path starting at `0` forms a substring.
/// - Every substring corresponds to a path starting at `0`.
/// - Each state corresponds to the set of all substrings that
    have the same
///  ending positions in the string, that is, each state `u`
    represents an
///  equivalence class according to their ending positions
    `endpos(u)`.
/// Given a state `u`, we can define the following concepts:
/// - `longest(u)`: longest substring corresponding to `u`.
/// - `len(u)`: length of `longest(u)`.
/// - `shortest(u)`: shortest substring corresponding to `u`.
/// - `minlen(u)`: length of `shortest(u)`.
/// Any state `u` corresponds to all suffixes of `longest(u)` no shorter
/// than `minlen(u)`.
/// For state `u`, `link(u)` points to the state `v` such that `longest(v)`
/// is a suffix of `longest(u)` with `len(v) == minlen(u) - 1`. These links
/// form a tree with the root in `0` and an inclusion relationship between
/// all `endpos`.
template<class Char=char>class SuffixAutomaton {
    using str = basic_string<Char>;
    void extend(Char c, int& last) {
        txt.pb(c); int p = last; last = new_state();
        len[last] = len[p] + 1, firstpos[last] = len[p];
        do {next[p][c] = last, p = link[p];} while (p >= 0 && !next[p].count(c));
        if (p == -1) {link[last] = 0;} else {
            int q = next[p][c];
            if (len[q] == len[p] + 1) {link[last] = q;} else {
                int cl = copy_state(q);
                len[cl] = len[p] + 1; link[last] = link[q] = cl;
                do {next[p][c] = cl, p = link[p];} while (p >= 0 && next[p].at(c) == q);
            }
        }
    }
    int new_state() {
        next.pb({}), link.pb(-1), len.pb(0), firstpos.pb(-1);
        return size++;
    }
    int copy_state(int state) {
        next.pb(next[state]), link.pb(link[state]);
        len.pb(len[state]), firstpos.pb(firstpos[state]);
        return size++;
    }
    void dfs(int curr=0) {
        terminal_paths_from[curr] = term[curr];
        paths_from[curr] = 1;
        fore(edge, next[curr]) {
            int other = edge.snd;
            if (!paths_from[other]) {dfs(other);}
            terminal_paths_from[curr] += terminal_paths_from[other];
            paths_from[curr] += paths_from[other];
            substrings_from[curr] += substrings_from[other];
        }
        substrings_from[curr] += terminal_paths_from[curr];
    }
    void compute(int last) {
        term.resize(size);
        for (int curr = last; curr != -1; curr = link[curr]) {term[curr] = true;}
        inv_link.resize(size);
        forr(curr, 1, size) {inv_link[link[curr]].pb(curr);}
    }
public:
    vector<bool> term; // Terminal statuses.
    vector<vector<int>> inv_link; // Inverse suffix links.
    vector<map<Char, int>> next{{}}; // Automaton transitions.
    vector<int> len{0}; // len[u] = lengh of longest(u)
    vector<int> link{-1}; // Suffix links.
    vector<int> firstpos{-1}; // First endpos element of each state.
    // Number of paths starting at each state and ending in a terminal state.
    // For `0`, this is the number of suffixes (including the empty suffix).
    vector<int> terminal_paths_from;
    // Number of paths starting at each state. For `0`, this is the number of
    // distinct substrings (including the empty substring).
    vector<ll> paths_from;
    // Number of substrings starting at each state. For `0`, this is the number
    // of substrings counting repetitions (including the empty substring
    // repeated `n+1` times, where `n` is the length of the
```

```cpp
        original string).
    vector<ll> substrings_from;
    int size = 1; // Number of states.
    str txt; // Original string.
    SuffixAutomaton(str const& _txt) {
        int last = 0;
        fore(c, _txt) {extend(c, last);}
        compute(last); terminal_paths_from.resize(size);
        paths_from.resize(size); substrings_from.resize(size); dfs();
    }
    pair<int, int> run(str const& pat) const {
        int curr = 0, read = 0; // curr = last visited state
        for (
            auto it = pat.begin();
            it != pat.end() && next[curr].count(*it);
            curr = next[curr].at(*(it++))
        ) {++read;} // read = number of traversed transitions
        return {curr, read};
    }
    bool is_suff(str const& pat) const
        {auto [state, read] = run(pat); return term[state] && read == sz(pat);}
    bool is_substr(str const& pat) const {return run(pat).snd == sz(pat);}
    int num_occs(str const& pat) const {
        auto [state, read] = run(pat);
        return read == sz(pat) ? terminal_paths_from[state] : 0;
    }
    int fst_occ(str const& pat) const {
        int m = sz(pat); auto [state, read] = run(pat);
        return read == m ? firstpos[state] + 1 - m : -1;
    }
    vector<int> all_occs(str const& pat) const {
        vector<int> occs; int m = sz(pat); auto [node, read] = run(pat);
        if (read == m) {
            stack<int> st{{node}};
            while (!st.empty()) {
                int curr = st.top(); st.pop();
                occs.pb(firstpos[curr] + 1 - m);
                fore(child, inv_link[curr]) {st.push(child);}
            }
        }
        // sort(all(occs)); occs.erase(unique(all(occs)), occs.end());
        return occs; // unsorted and nonunique by default
    }
};
```

6. Grafos

6.1. Dikjstra

```cpp
vector<pair<int,int>> g[MAXN]; // u->[(v,cost)]
ll dist[MAXN];
// complejidad O((E+V)*log(V))
void dijkstra(int x){
    memset(dist,-1,sizeof(dist));
    priority_queue<pair<ll,int> > q;
    dist[x]=0;q.push({0,x});
    while(!q.empty()){
        x=q.top().snd;ll c=-q.top().fst;q.pop();
        if(dist[x]!=c)continue;
        forn(i,g[x].size()){
            int y=g[x][i].fst; ll c=g[x][i].snd;
            if(dist[y]<0||dist[x]+c<dist[y])
                dist[y]=dist[x]+c,q.push({-dist[y],y});
        }
    }
}
```

6.2. LCA

```cpp
int n;
vector<int> g[MAXN];

vector<int> depth, etour, vtime;

// operación de la sparse table, escribir `#define oper lca_oper`
```

```cpp
int lca_oper(int u, int v) { return depth[u]<depth[v] ? u : v; };

void lca_dfs(int u) {
    vtime[u] = sz(etour), etour.push_back(u);
    for (auto v : g[u]) {
        if (vtime[v] >= 0) continue;
        depth[v] = depth[u]+1; lca_dfs(v); etour.push_back(u);
    }
}
auto lca_init(int root) {
    depth.assign(n,0), etour.clear(), vtime.assign(n,-1);
    lca_dfs(root); st_init(etour);
}

auto lca(int u, int v) {
    int l = min(vtime[u],vtime[v]);
    int r = max(vtime[u],vtime[v])+1;
    return st_query(l,r);
}
int dist(int u, int v) { return
    depth[u]+depth[v]-2*depth[lca(u,v)]; }
```

6.3. Binary Lifting

```cpp
vector<int> g[1<<K]; int n; // K such that 2^K>=n
int F[K][1<<K], D[1<<K];
void lca_dfs(int x){
    forn(i, sz(g[x])){
        int y = g[x][i]; if(y==F[0][x]) continue;
        F[0][y]=x; D[y]=D[x]+1;lca_dfs(y);
    }
}
void lca_init(){
    D[0]=0;F[0][0]=-1;
    lca_dfs(0);
    forr(k,1,K)forn(x,n)
        if(F[k-1][x]<0)F[k][x]=-1;
        else F[k][x]=F[k-1][F[k-1][x]];
}

int lca(int x, int y){
    if(D[x]<D[y])swap(x,y);
    for(int k = K-1;k>=0;--k) if(D[x]-(1<<k) >=D[y])x=F[k][x];
    if(x==y)return x;
    for(int k=K-1;k>=0;--k)if(F[k][x]!=F[k][y])x=F[k][x],y=F[k][y];
    return F[0][x];
}

int dist(int x, int y){
    return D[x] + D[y] - 2*D[lca(x,y)];
}
```

6.4. Toposort

```cpp
vector<int> g[MAXN];int n;
vector<int> tsort(){ // lexicographically smallest topological sort
    vector<int> r;priority_queue<int> q;
    vector<int> d(2*n,0);
    forn(i,n)forn(j,g[i].size())d[g[i][j]]++;
    forn(i,n)if(!d[i])q.push(-i);
    while(!q.empty()){
        int x=-q.top();q.pop();r.pb(x);
        forn(i,sz(g[x])){
            d[g[x][i]]--;
            if(!d[g[x][i]])q.push(-g[x][i]);
        }
    }
    return r; // if not DAG it will have less than n elements
}
```

6.5. Deteccion ciclos negativos

```cpp
// g[i][j]: weight of edge (i, j) or INF if there's no edge
// g[i][i]=0
ll g[MAXN][MAXN];int n;
void floyd(){ // O(n^3) . Replaces g with min distances
    forn(k,n)forn(i,n)if(g[i][k]<INF)forn(j,n)if(g[k][j]<INF)
        g[i][j]=min(g[i][j],g[i][k]+g[k][j]);
}
bool inNegCycle(int v){return g[v][v]<0;}
bool hasNegCycle(int a, int b){ // true iff there's neg cycle in
    between
    forn(i,n)if(g[a][i]<INF&&g[i][b]<INF&&g[i][i]<0)return true;
    return false;
}
```

6.6. Camino Euleriano

```cpp
// Directed version (uncomment commented code for undirected)
struct edge {
    int y;
//  list<edge>::iterator rev;
    edge(int y):y(y){}
};
list<edge> g[MAXN];
void add_edge(int a, int b){
    g[a].push_front(edge(b));//auto ia=g[a].begin();
//  g[b].push_front(edge(a));auto ib=g[b].begin();
//  ia->rev=ib;ib->rev=ia;
}
vector<int> p;
void go(int x){
    while(g[x].size()){
        int y=g[x].front().y;
        //g[y].erase(g[x].front().rev);
        g[x].pop_front();
        go(y);
    }
    p.push_back(x);
}
vector<int> get_path(int x){ // get a path that begins in x
// check that a path exists from x before calling to get_path!
    p.clear();go(x);reverse(p.begin(),p.end());
    return p;
}
```

6.7. Camino Hamiltoniano

```cpp
constexpr int MAXN = 20;
int n;
bool adj[MAXN][MAXN];

bool seen[1<<MAXN][MAXN];
bool memo[1<<MAXN][MAXN];
// true sii existe camino simple en el conjunto s que empieza en u
bool hamilton(int s, int u) {
    bool& ans = memo[s][u];
    if (seen[s][u]) return ans;
    seen[s][u] = true, s ^= (1<<u);
    if (s == 0) return ans = true;
    forn(v,n) if (adj[u][v] && (s&(1<<v)) && hamilton(s,v)) return
        ans = true;
    return ans = false;
}
// true sii existe camino hamiltoniano. complejidad O((1<<n)*n*n)
bool hamilton() {
    forn(s,1<<n) forn(u,n) seen[s][u] = false;
    forn(u,n) if (hamilton((1<<n)-1,u)) return true;
    return false;
}
```

6.8. Tarjan SCC

```cpp
vector<int> g[MAXN], ss;
int n, num, order[MAXN], lnk[MAXN], nsc, cmp[MAXN];
void scc(int u) {
    order[u] = lnk[u] = ++num;
    ss.pb(u); cmp[u] = -2;
    for (auto v : g[u]) {
        if (order[v] == 0) {
            scc(v);
            lnk[u] = min(lnk[u], lnk[v]);
        }
        else if (cmp[v] == -2) {
            lnk[u] = min(lnk[u], lnk[v]);
        }
    }
    if (lnk[u] == order[u]) {
        int v;
        do { v = ss.back(); cmp[v] = nsc; ss.pop_back(); }
        while (v != u);
```

```
      nsc++;
   }
}
void tarjan() {
   memset(order, 0, sizeof(order)); num = 0;
   memset(cmp, -1, sizeof(cmp)); nsc = 0;
   forn (i, n) if (order[i] == 0) scc(i);
}
```

6.9. Bellman-Ford

```
const int INF=2e9; int n;
vector<pair<int,int> > g[MAXN]; // u->[(v,cost)]
ll dist[MAXN];
void bford(int src){ // O(nm)
   fill(dist,dist+n,INF);dist[src]=0;
   forr(_,0,n)forr(x,0,n)if(dist[x]!=INF)for(auto t:g[x]){
      dist[t.fst]=min(dist[t.fst],dist[x]+t.snd);
   }
   forr(x,0,n)if(dist[x]!=INF)for(auto t:g[x]){
      if(dist[t.fst]>dist[x]+t.snd){
         // neg cycle: all nodes reachable from t.fst have
         // -INF distance
         // to reconstruct neg cycle: save "prev" of each
         // node, go up from t.fst until repeating a node.
         // this node and all nodes between the two
         // occurences form a neg cycle
      }
   }
}
```

6.10. Puentes y Articulacion

```
// solo para grafos no dirigidos
vector<int> g[MAXN];
int n, num, root, rootChildren;
int order[MAXN], lnk[MAXN], art[MAXN];
void bridge_art(int u, int p) {
   order[u] = lnk[u] = ++num;
   for (auto v : g[u]) if (v != p) {
      if (u == root) rootChildren++;
      if (order[v] == 0) {
         bridge_art(v, u);
         if (lnk[v] >= order[u])   // para puntos de
            art[u] = 1;            // articulacion.
         if (lnk[v] > order[u])    // para puentes.
            handle_bridge(u, v);
      }
      lnk[u] = min(lnk[u], lnk[v]);
   }
}
void run() {
   memset(order, 0, sizeof(order));
   memset(art, 0, sizeof(art)); num = 0;
   forn (i, n) {
      if (order[i] == 0) {
         root = i; bridge_art(i, -1);
         art[i] = (rootChildren > 1);
      }
   }
}
```

6.11. Kruskal

```
int uf[MAXN];
void uf_init(){memset(uf,-1,sizeof(uf));}
int uf_find(int x){return uf[x]<0?x:uf[x]=uf_find(uf[x]);}
bool uf_join(int x, int y){
   x=uf_find(x);y=uf_find(y);
   if(x==y)return false;
   if(uf[x]>uf[y])swap(x,y);
   uf[x]+=uf[y];uf[y]=x;
   return true;
}
vector<pair<ll,pair<int,int> > > es; // edges (cost,(u,v))
ll kruskal(){ // assumes graph is connected
   sort(es.begin(),es.end());uf_init();
   ll r=0;
   forr(i,0,es.size()){
      int x=es[i].snd.fst,y=es[i].snd.snd;
      if(uf_join(x,y))r+=es[i].fst; // (x,y,c) belongs to mst
```

```cpp
    }
    return r; // total cost
}
```

6.12. Chequeo Bipartito

```cpp
int n;
vector<int> g[MAXN];

bool color[MAXN];
bool bicolor() {
    vector<bool> seen(n);
    auto dfs = [&](auto&& me, int u, bool c) -> bool {
        color[u] = c, seen[u] = true;
        for (int v : g[u]) {
            if (seen[v] && color[v] == color[u]) return false;
            if (!seen[v] && !me(me,v,!c)) return false;
        }
        return true;
    };
    forn(u,n) if (!seen[u] && !dfs(dfs,u,0)) return false;
    return true;
}
```

6.13. Centroid Decomposition

```cpp
bool vis[MAXN]; //para centroides
vector<int> g[MAXN]; int size[MAXN];
vector<int> g1[MAXN]; //para centroides
void calcsz(int u, int p) {
    size[u] = 1;
    for (int v : g[u]) if (v != p && !vis[v]) {
        calcsz(v, u); size[u] += size[v]; }
}
int cendfs(int u, int p, int ts) {
    int maximo = 0, pesado, r;
    for (int v : g[u]) if (v != p && !vis[v]) {
        if (maximo < size[v]) {
            maximo = size[v]; pesado = v; }
    }
    if (maximo <= (ts/2)) {
        vis[u] = true;
        for (int v : g[u]) if (!vis[v]) {
            if (v == p) calcsz(v, u);
            r = cendfs(v, u, hijos[v]);
            add_edge(g1, u, r);
        }
        r = u;
    }
    else r = cendfs(pesado, u, ts);
    return r;
}
// euler para responder en el arbol de centroides
int te[MAXN], ts[MAXN]; vector<Partial> euler;
void do_euler(int u, int p, Partial &p) {
    te[u] = sz(euler); euler.pb(c);
    for (int v : g[u]) if (v != p && !vis[v]) {
        do_euler(v, u, p); } //cambiar p
    ts[u] = sz(euler);
}
Sol oncen(int u, int p) {
    do_euler(u, p, Partial{});
    vis[u] = true; //no tocar visitados
    Sol r{};
    for (int v : g1[u]) if (v != p) {
        r = max(r, oncen(v, u)); }
    return r;
}
```

6.14. HLD

```cpp
vector<int> g[MAXN];
int wg[MAXN],dad[MAXN],dep[MAXN]; // weight,father,depth
void dfs1(int x){
    wg[x]=1;
    for(int y:g[x])if(y!=dad[x]){
        dad[y]=x;dep[y]=dep[x]+1;dfs1(y);
        wg[x]+=wg[y];
    }
}
int curpos,pos[MAXN],head[MAXN];
```

```
11 void hld(int x, int c){
12     if(c<0)c=x;
13     pos[x]=curpos++;head[x]=c;
14     int mx=-1;
15     for(int y:g[x])if(y!=dad[x]&&(mx<0||wg[mx]<wg[y]))mx=y;
16     if(mx>=0)hld(mx,c);
17     for(int y:g[x])if(y!=mx&&y!=dad[x])hld(y,-1);
18 }
19 void hld_init(){dad[0]=-1;dep[0]=0;dfs1(0);curpos=0;hld(0,-1);}
20 int query(int x, int y, RMQ& rmq){
21     int r=neutro; //neutro del rmq
22     while(head[x]!=head[y]){
23         if(dep[head[x]]>dep[head[y]])swap(x,y);
24         r=oper(r,rmq.get(pos[head[y]],pos[y]+1));
25         y=dad[head[y]];
26     }
27     if(dep[x]>dep[y])swap(x,y); // now x is lca
28     r=oper(r,rmq.get(pos[x],pos[y]+1));
29     return r;
30 }
31 // hacer una vez al principio hld_init() después de armar el grafo en g
32 // para querys pasar los dos nodos del camino y un stree que tiene en pos[x] el valor del nodo x
33 // for updating: rmq.set(pos[x],v);
34 // queries on edges: - assign values of edges to "child" node () ***
35 //                   - change pos[x] to pos[x]+1 in query (line 28)
36 // *** if(dep[u] > dep[v]) rmq.upd(pos[u], w) para cada arista (u,v)
```

6.15. Max Tree Matching

```
int n, r, p[MAXN]; // número de nodos, raíz, y lista de padres
vector<int> g[MAXN]; // lista de adyancencia

int match[MAXN];
// encuentra el max matching del árbol. complejidad O(n)
int maxmatch() {
    fill(match,match+n,-1);
    int size = 0;
    auto dfs = [&](auto&& me, int u) -> int {
        for (auto v : g[u]) if (v != p[u])
            if (match[u] == me(me,v)) match[u] = v, match[v] = u;
        size += match[u] >= 0;
        return match[u];
    };
    dfs(dfs,r);
    return size;
}
```

6.16. Min Tree Vertex Cover

```
int n, r, p[MAXN]; // número de nodos, raíz, y lista de padres
vector<int> g[MAXN]; // lista de adyancencia

bool cover[MAXN];
// encuentra el min vertex cover del árbol. complejidad O(n)
int mincover() {
    fill(cover,cover+n,false);
    int size = 0;
    auto dfs = [&](auto&& me, int u) -> bool {
        for (auto v : g[u]) if (v != p[u] && !me(me,v)) cover[u] = true;
        size += cover[u];
        return cover[u];
    };
    dfs(dfs,r);
    return size;
}
```

6.17. 2-SAT

```
struct TwoSatSolver{
    int n_vars;
    int n_vertices;
    vector<vector<int>> adj, adj_t;
    vector<bool> used;
    vector<int> order,comp;
    vector<bool> assignment;
    TwoSatSolver(int _n_vars) : n_vars(_n_vars),
        n_vertices(2*n_vars), adj(n_vertices),
        adj_t(n_vertices), used(n_vertices),
        order(), comp(n_vertices, -1), assignment(n_vars){
        order.reserve(n_vertices);
    }
    void dfs1(int v){
        used[v] = true;
        for(int u : adj[v]){
            if(!used[u]) dfs1(u);
        }
        order.pb(v);
    }
    void dfs2(int v, int c1){
        comp[v] = c1;
        for(int u : adj_t[v]){
            if(comp[u] == -1) dfs2(u, c1);
        }
    }
    bool solve_2SAT(){
        order.clear();
        used.assign(n_vertices, false);
        forn(i, n_vertices){
            if(!used[i]) dfs1(i);
        }
        comp.assign(n_vertices, -1);
        for(int i = 0, j = 0; i < n_vertices; ++i){
            int v = order[n_vertices - i - 1];
            if(comp[v] == -1) dfs2(v, j++);
        }
        assignment.assign(n_vars, false);
        for(int i = 0; i < n_vertices; i+=2){
            if(comp[i] == comp[i+1]) return false;
            assignment[i/2] = comp[i] > comp[i+1];
        }
        return true;
    }
    void add_disjunction(int a, bool na, int b, bool nb){
        a = 2 * a ^ na;
        b = 2 * b ^ nb;
        int neg_a = a ^ 1;
        int neg_b = b ^ 1;
        adj[neg_a].pb(b);
        adj[neg_b].pb(a);
        adj_t[b].pb(neg_a);
        adj_t[a].pb(neg_b);
    }
};
```

6.18. K Colas

```
const int K=9999; // en general, K = MAX_DIST+1
vector<Datos> colas[K];
int cola_actual = 0, ult_cola = -1;
// push toma la dist actual y la siguiente
#define push(d,nd,args...)
    colas[(cola_actual+nd-d)%K].emplace_back(nd, args)
#define pop colas[cola_actual].pop_back
#define top colas[cola_actual].back
// PUSHEAR POSICION INICIAL
for (; ; cola_actual = (cola_actual+1)%K) {
    if (ult_cola == cola) break; // dimos la vuelta
    if (colas[cola_actual].size()) ult_cola = cola;
    while (colas[cola_actual].size()) {

    }
}
```

7. Flujo

7.1. Dinic

```
// complejidad O(V^2*E)
struct Dinic{
    int nodes,src,dst;
    vector<int> dist,q,work;
    struct edge {int to,rev;ll f,cap;};
    vector<vector<edge>> g;
    Dinic(int x):nodes(x),g(x),dist(x),q(x),work(x){}
    void add_edge(int s, int t, ll cap){
        g[s].pb((edge){t,sz(g[t]),0,cap});
        g[t].pb((edge){s,sz(g[s])-1,0,0});
    }
    bool dinic_bfs(){
        fill(all(dist),-1);dist[src]=0;
        int qt=0;q[qt++]=src;
        for(int qh=0;qh<qt;qh++){
            int u=q[qh];
            forn(i,sz(g[u])){
                edge &e=g[u][i];int v=g[u][i].to;
                if(dist[v]<0&&e.f<e.cap)dist[v]=dist[u]+1,q[qt++]=v;
            }
        }
        return dist[dst]>=0;
    }
    ll dinic_dfs(int u, ll f){
        if(u==dst)return f;
        for(int &i=work[u];i<sz(g[u]);i++){
            edge &e=g[u][i];
            if(e.cap<=e.f)continue;
            int v=e.to;
            if(dist[v]==dist[u]+1){
                ll df=dinic_dfs(v,min(f,e.cap-e.f));
                if(df>0){e.f+=df;g[v][e.rev].f-=df;return df;}
            }
        }
        return 0;
    }
    ll max_flow(int _src, int _dst){
        src=_src;dst=_dst;
        ll result=0;
        while(dinic_bfs()){
            fill(all(work),0);
            while(ll delta=dinic_dfs(src,INF))result+=delta;
        }
        return result;
    }
};
```

7.2. Min Cost Max Flow

```
typedef ll tf;
typedef ll tc;
const tf INFFLOW=1e9;
const tc INFCOST=1e9;
// complejidad O(V^2*E*log(V))
struct MCF{
    int n;
    vector<tc> prio, pot; vector<tf> curflow; vector<int>
        prevedge,prevnode;
    priority_queue<pair<tc, int>, vector<pair<tc, int>>,
        greater<pair<tc, int>>> q;
    struct edge{int to, rev; tf f, cap; tc cost;};
    vector<vector<edge>> g;
    MCF(int
        n):n(n),prio(n),curflow(n),prevedge(n),prevnode(n),pot(n),g(n){}
    void add_edge(int s, int t, tf cap, tc cost) {
        g[s].pb((edge){t,sz(g[t]),0,cap,cost});
        g[t].pb((edge){s,sz(g[s])-1,0,0,-cost});
    }
    pair<tf,tc> get_flow(int s, int t) {
        tf flow=0; tc flowcost=0;
        while(1){
            q.push({0, s});
            fill(all(prio),INFCOST);
            prio[s]=0; curflow[s]=INFFLOW;
            while(!q.empty()) {
                auto cur=q.top();
                tc d=cur.fst;
                int u=cur.snd;
                q.pop();
                if(d!=prio[u]) continue;
                for(int i=0; i<sz(g[u]); ++i) {
                    edge &e=g[u][i];
                    int v=e.to;
                    if(e.cap<=e.f) continue;
                    tc nprio=prio[u]+e.cost+pot[u]-pot[v];
                    if(prio[v]>nprio) {
                        prio[v]=nprio;
                        q.push({nprio, v});
                        prevnode[v]=u; prevedge[v]=i;
```

```cpp
                  curflow[v]=min(curflow[u], e.cap-e.f);
               }
            }
         }
         if(prio[t]==INFCOST) break;
         forr(i,0,n) pot[i]+=prio[i];
         tf df=min(curflow[t], INFFLOW-flow);
         flow+=df;
         for(int v=t; v!=s; v=prevnode[v]) {
            edge &e=g[prevnode[v]][prevedge[v]];
            e.f+=df; g[v][e.rev].f-=df;
            flowcost+=df*e.cost;
         }
      }
      return {flow,flowcost};
   }
};
```

7.3. Hopcroft Karp

```cpp
int n, m;          // número de nodos en ambas partes
vector<int> g[MAXN]; // lista de adyacencia [0,n) -> [0,m)

int mat[MAXN]; // matching [0,n) -> [0,m)
int inv[MAXM]; // matching [0,m) -> [0,n)
// encuentra el max matching del grafo bipartito
// complejidad O(sqrt(n+m)*e), donde e es el número de aristas
int hopkarp() {
   fill(mat,mat+n,-1);
   fill(inv,inv+m,-1);
   int size = 0;
   vector<int> d(n);
   auto bfs = [&] {
      bool aug = false;
      queue<int> q;
      forn(u,n) if (mat[u] < 0) q.push(u); else d[u] = -1;
      while (!q.empty()) {
         int u = q.front();
         q.pop();
         for (auto v : g[u]) {
            if (inv[v] < 0) aug = true;
            else if (d[inv[v]] < 0) d[inv[v]] = d[u] + 1,
               q.push(inv[v]);
         }
      }
      return aug;
   };
   auto dfs = [&](auto&& me, int u) -> bool {
      for (auto v : g[u]) if (inv[v] < 0) {
         mat[u] = v, inv[v] = u;
         return true;
      }
      for (auto v : g[u]) if (d[inv[v]] > d[u] && me(me,inv[v])) {
         mat[u] = v, inv[v] = u;
         return true;
      }
      d[u] = 0;
      return false;
   };
   while (bfs()) forn(u,n) if (mat[u] < 0) size += dfs(dfs,u);
   return size;
}
```

7.4. Kuhn

```cpp
int n, m;          // número de nodos en ambas partes
vector<int> g[MAXN]; // lista de adyacencia [0,n) -> [0,m)

int mat[MAXN]; // matching [0,n) -> [0,m)
int inv[MAXM]; // matching [0,m) -> [0,n)
// encuentra el max matching del grafo bipartito
// complejidad O(n*e), donde e es el número de aristas
int kuhn() {
   fill(mat,mat+n,-1);
   fill(inv,inv+m,-1);
   int root, size = 0;
   vector<int> seen(n,-1);
   auto dfs = [&](auto&& me, int u) -> bool {
      seen[u] = root;
      for (auto v : g[u]) if (inv[v] < 0) {
            mat[u] = v, inv[v] = u;
            return true;
        }
        for (auto v : g[u]) if (seen[inv[v]] < root &&
             me(me,inv[v])) {
            mat[u] = v, inv[v] = u;
            return true;
        }
        return false;
    };
    forn(u,n) size += dfs(dfs,root=u);
    return size;
}
```

7.5. Min Vertex Cover Bipartito

```cpp
// requisito: max matching bipartito, por defecto Hopcroft-Karp

vector<bool> cover[2]; // nodos cubiertos en ambas partes
// encuentra el min vertex cover del grafo bipartito
// misma complejidad que el algoritmo de max matching bipartito
     elegido
int konig() {
    cover[0].assign(n,true);
    cover[1].assign(m,false);
    int size = hopkarp(); // alternativamente, también funciona
         con Kuhn
    auto dfs = [&](auto&& me, int u) -> void {
        cover[0][u] = false;
        for (auto v : g[u]) if (!cover[1][v]) {
            cover[1][v] = true;
            me(me,inv[v]);
        }
    };
    forn(u,n) if (mat[u] < 0) dfs(dfs,u);
    return size;
}
```

7.6. Hungarian

```cpp
typedef long double td; typedef vector<int> vi; typedef vector<td>
     vd;
const td INF=1e100;//for maximum set INF to 0, and negate costs
bool zero(td x){return fabs(x)<1e-9;}//change to x==0, for ints/ll
struct Hungarian{
    int n; vector<vd> cs; vi L, R;
    Hungarian(int N, int M):n(max(N,M)),cs(n,vd(n)),L(n),R(n){
        forr(x,0,N)forr(y,0,M)cs[x][y]=INF;
    }
    void set(int x,int y,td c){cs[x][y]=c;}
    td assign() {
        int mat = 0; vd ds(n), u(n), v(n); vi dad(n), sn(n);
        forr(i,0,n)u[i]=*min_element(all(cs[i]));
        forr(j,0,n){
            v[j]=cs[0][j]-u[0];
            forr(i,1,n)v[j]=min(v[j],cs[i][j]-u[i]);
        }
        L=R=vi(n, -1);
        forr(i,0,n)forr(j,0,n) {
            if(R[j]==-1&&zero(cs[i][j]-u[i]-v[j])){
                L[i]=j;R[j]=i;mat++;break;
        } }
        for(;mat<n;mat++){
            int s=0, j=0, i;
            while(L[s] != -1)s++;
            fill(all(dad),-1);fill(all(sn),0);
            forr(k,0,n)ds[k]=cs[s][k]-u[s]-v[k];
            for(;;){
                j = -1;
                forr(k,0,n)if(!sn[k]&&(j==-1||ds[k]<ds[j]))j=k;
                sn[j] = 1; i = R[j];
                if(i == -1) break;
                forr(k,0,n)if(!sn[k]){
                    auto new_ds=ds[j]+cs[i][k]-u[i]-v[k];
                    if(ds[k] > new_ds){ds[k]=new_ds;dad[k]=j;}
                }
            }
            forr(k,0,n)if(k!=j&&sn[k]){auto
                 w=ds[k]-ds[j];v[k]+=w,u[R[k]]-=w;}
            u[s] += ds[j];
            while(dad[j]>=0){int d = dad[j];R[j]=R[d];L[R[j]]=j;j=d;}
            R[j]=s;L[s]=j;
        }
        td value=0;forr(i,0,n)value+=cs[i][L[i]];
        return value;
    }
};
```

8. Optimización

8.1. Ternary Search

```
// mínimo entero de f en (l,r)
ll ternary(auto f, ll l, ll r) {
    for (ll d = r-l; d > 2; d = r-l) {
        ll a = l+d/3, b = r-d/3;
        if (f(a) > f(b)) l = a; else r = b;
    }
    return l+1; // retorna un punto, no un resultado de evaluar f
}

// mínimo real de f en (l,r)
// para error < EPS, usar iters = log((r-l)/EPS)/log(1.618)
double golden(auto f, double l, double r, int iters) {
    constexpr double ratio = (3-sqrt(5))/2;
    double x1 = l+(r-l)*ratio, f1 = f(x1);
    double x2 = r-(r-l)*ratio, f2 = f(x2);
    while (iters--) {
        if (f1 > f2) l=x1, x1=x2, f1=f2, x2=r-(r-l)*ratio, f2=f(x2);
        else         r=x2, x2=x1, f2=f1, x1=l+(r-l)*ratio, f1=f(x1);
    }
    return (l+r)/2; // retorna un punto, no un resultado de evaluar f
}
```

8.2. Longest Increasing Subsequence

```
1  // subsecuencia creciente más larga
2  // para no decreciente, borrar la línea 9 con el continue
3  template<class Type> vector<int> lis(vector<Type>& a) {
4      int n = sz(a);
5      vector<int> seq, prev(n,-1), idx(n+1,-1);
6      vector<Type> dp(n+1,INF); dp[0] = -INF;
7      forn(i,n) {
8          int l = int(upper_bound(all(dp),a[i])-begin(dp));
9          if (dp[l-1] == a[i]) continue;
10         prev[i] = idx[l-1], idx[l] = i, dp[l] = a[i];
11     }
12     dforn(i,n+1) {
13         if (dp[i] < INF) {
14             for (int k = idx[i]; k >= 0; k = prev[k]) seq.pb(k);
15             reverse(all(seq));
16             break;
17         }
18     }
19     return seq;
20 }
```

9. Otros

9.1. Mo

```
int n,sq,nq; // array size, sqrt(array size), #queries
struct qu{int l,r,id;};
qu qs[MAXN];
ll ans[MAXN]; // ans[i] = answer to ith query
bool qcomp(const qu &a, const qu &b){
    if(a.l/sq!=b.l/sq) return a.l<b.l;
    return (a.l/sq)&1?a.r<b.r:a.r>b.r;
}
void mos(){
    forn(i,nq)qs[i].id=i;
    sq=sqrt(n)+.5;
    sort(qs,qs+nq,qcomp);
    int l=0,r=0;
    init();
    forn(i,nq){
        qu q=qs[i];
        while(l>q.l)add(--l);
        while(r<q.r)add(r++);
        while(l<q.l)remove(l++);
        while(r>q.r)remove(--r);
        ans[q.id]=get_ans();
    }
}
```

9.2. Divide and Conquer Optimization

```
vector<ll> dp_ant, dp_curr;

void compute(int l, int r, int optl, int optr){
    if(l == r) return;
    int m = (l+r)/2;
    ll dpm = 1e17;
    int optm = -1;
    forr(i, max(m+1, optl), optr+1){
        ll cost = C(m, i) + (i == n ? 0 : dp_ant[i]);
        if(cost < dpm) dpm = cost, optm = i;
    }
    dp_curr[m] = dpm;
    compute(l, m, optl, optm);
    compute(m+1, r, optm, optr);
}


forn(i, k){
    compute(0, n, 0, n);
    dp_ant = dp_curr;
}
cout << dp_curr[0] << endl;
```

9.3. Fijar el numero de decimales

```
// antes de imprimir decimales, con una sola vez basta
cout << fixed << setprecision(DECIMAL_DIG);
```

9.4. Hash Table (Unordered Map/ Unordered Set)

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
template<class Key,class Val=null_type>using
    htable=gp_hash_table<Key,Val>;
// como unordered_map (o unordered_set si Val es vacio), pero sin
    metodo count
```

9.5. Indexed Set

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
template<class Key, class Val=null_type>
using indexed_set = tree<Key, Val, less<Key>, rb_tree_tag,
                        tree_order_statistics_node_update>;
// indexed_set<char> s;
// char val = *s.find_by_order(0); // acceso por indice
// int idx = s.order_of_key('a'); // busca indice del valor
```

9.6. Subconjuntos

```
// iterar por mascaras O(2^n)
for(int bm=0; bm<(1<<n); bm++)
// subconjuntos de una mascara O(2^n)
for(int sbm=bm; sbm; sbm=(sbm-1)&bm)
// iterar por submascaras O(3^n)
for(int bm=0; bm<(1<<n); bm++)
    for(int sbm=bm; sbm; sbm=(sbm-1)&(bm))
// para superconjuntos (que contienen a bm),
// negar la mascara: bm=~bm
```

9.7. Simpson

```
// integra f en [a,b] llamándola 2*n veces
double simpson(auto f, double a, double b, int n=1e4) {
    double h = (b-a)/2/n, s = f(a);
    forr(i,1,2*n) s += f(a+i*h) * ((i%2)?4:2);
    return (s+f(b))*h/3;
}
```

9.8. Pragmas

```
#pragma GCC target("avx2")
#pragma GCC optimize("O3")
#pragma GCC optimize("unroll-loops")
```

9.9. Random

```cpp
unsigned seed =
    std::chrono::steady_clock::now().time_since_epoch().count();
mt19937 generator(seed);
generator(); // generar un número aleatorio entre 0 y 4294967295
// existe mt19937_64 para la versión de 64 bits, que probablemente
    sea más rápido

/*
// también se puede hacer lo siguiente para una versión hasta 3x
    más rápida:
#include <ext/random>
using namespace __gnu_cxx;
unsigned seed =
    std::chrono::steady_clock::now().time_since_epoch().count();
sfmt19937 generator(seed); // existe también sfmt19937_64
*/

uniform_int_distribution<ll> dist_int(L, R);
dist_int(generator); // generar un entero en [L, R]
    (cerrado-cerrado) con prob uniforme

uniform_real_distribution<double> dist_real(0.0, 1.0);
dist_real(generator); // generar un real en [0, 1)
    (cerrado-abierto) con prob uniforme
```

9.10. Utilidades de strings

```cpp
getline(cin, linea); // tomar toda la linea
stringstream ss(linea); // tratar una linea como stream
ss >> s; ss << s; // leer solo hasta un espacio, escribir a ss
tipo n; ss >> n; // leer de un stringstream (float, int, etc.)
int pos = s.find_first_of("aeoiu"); // devuelve -1 si no encuentra
int next = s.find_first_of("aeoiu", pos);
// s.find_first_not_of("aeoiu"); s.find_last_of();
s.substr(pos, next-pos); // substr(pos, len)
s.c_str(); // devuelve un puntero de C
ss.str(); // devuelve el string en ss
// isspace(); islower(); isupper(); isdigit(); isalpha();
// tolower(); toupper();
```