

Índice

1. Teoría de números	5		
1.1. Funciones básicas	5	1.3.2. Potencia de un primo que divide a un factorial	10
1.1.1. Función piso y techo	5	1.3.3. Factorización de un factorial	10
1.1.2. Exponenciación y multiplicación binaria	5	1.3.4. Factorización usando Pollard-Rho	10
1.1.3. Mínimo común múltiplo y máximo común divisor	5	1.4. Funciones multiplicativas famosas	11
1.1.4. Euclides extendido e inverso modular	5	1.4.1. Función σ	11
1.1.5. Todos los inversos módulo p	6	1.4.2. Función Ω	11
1.1.6. Exponenciación binaria modular	6	1.4.3. Función ω	11
1.1.7. Teorema chino del residuo	6	1.4.4. Función φ de Euler	11
1.1.8. Coeficiente binomial	6	1.4.5. Función μ	11
1.1.9. Fibonacci	6	1.5. Orden multiplicativo, raíces primitivas y raíces de la unidad	12
1.2. Cribas	7	1.5.1. Función λ de Carmichael	12
1.2.1. Criba de divisores	7	1.5.2. Orden multiplicativo módulo m	12
1.2.2. Criba de primos	7	1.5.3. Número de raíces primitivas (generadores) módulo m	12
1.2.3. Criba de factor primo más pequeño	7	1.5.4. Test individual de raíz primitiva módulo m	12
1.2.4. Criba de factores primos	7	1.5.5. Test individual de raíz k -ésima de la unidad módulo m	12
1.2.5. Criba de la función φ de Euler	8	1.5.6. Encontrar la primera raíz primitiva módulo m	13
1.2.6. Criba de la función μ	8	1.5.7. Encontrar la primera raíz k -ésima de la unidad módulo m	13
1.2.7. Triángulo de Pascal	8	1.5.8. Logaritmo discreto	13
1.2.8. Segmented sieve	8	1.5.9. Raíz k -ésima discreta	13
1.2.9. Criba de primos lineal	9	1.6. Particiones	14
1.2.10. Criba lineal para funciones multiplicativas	9	1.6.1. Función P (particiones de un entero positivo)	14
1.3. Factorización	9	1.6.2. Función Q (particiones de un entero positivo en distintos sumandos)	14
1.3.1. Factorización de un número	9	1.6.3. Número de factorizaciones ordenadas	15

1.6.4. Número de factorizaciones no ordenadas	15	4.3. FFT con raíces de la unidad discretas (NTT)	26
1.7. Otros	16	4.3.1. Otros valores para escoger la raíz y el módulo	26
1.7.1. Cambio de base	16	4.4. Aplicaciones	27
1.7.2. Fracciones continuas	16	4.4.1. Multiplicación de polinomios	27
1.7.3. Ecuación de Pell	17	4.4.2. Multiplicación de números enteros grandes	27
1.7.4. Números de Bell	17	4.4.3. Inverso de un polinomio	27
2. Números racionales	17	4.4.4. Raíz cuadrada de un polinomio	28
2.1. Estructura <code>fraccion</code>	17	5. Geometría	29
3. Álgebra lineal	19	5.1. Estructura <code>point</code>	29
3.1. Estructura <code>matrix</code>	19	5.2. Líneas y segmentos	30
3.2. Gauss Jordan	20	5.2.1. Verificar si un punto pertenece a una línea o segmento	30
3.3. Matriz inversa	21	5.2.2. Intersección de líneas	30
3.4. Transpuesta	21	5.2.3. Intersección línea-segmento	30
3.5. Traza	21	5.2.4. Intersección de segmentos	31
3.6. Determinante	21	5.2.5. Distancia punto-recta	31
3.7. Matriz de cofactores y adjunta	22	5.3. Círculos	31
3.8. Factorización $PA = LU$	22	5.3.1. Distancia punto-círculo	31
3.9. Polinomio característico	22	5.3.2. Proyección punto exterior a círculo	31
3.10. Gram-Schmidt	23	5.3.3. Puntos de tangencia de punto exterior	31
3.11. Recurrencias lineales	23	5.3.4. Intersección línea-círculo	31
3.12. Simplex	23	5.3.5. Centro y radio a través de tres puntos	32
4. FFT	25	5.3.6. Intersección de círculos	32
4.1. Funciones previas	25	5.3.7. Tangentes	32
4.2. FFT con raíces de la unidad complejas	25	5.4. Polígonos	33
		5.4.1. Perímetro y área de un polígono	33

5.4.2. Envolverte convexa (convex hull) de un polígono	33	6.13. Detectar ciclos	40
5.4.3. Verificar si un punto pertenece al perímetro de un polígono	34	6.14. Puentes y puntos de articulación	41
5.4.4. Verificar si un punto pertenece a un polígono	34	6.15. Componentes fuertemente conexas	41
5.4.5. Centroides de un polígono	34	6.16. Árbol mínimo de expansión (Kruskal)	41
5.4.6. Pares de puntos antipodales	34	6.17. Máximo emparejamiento bipartito	42
5.4.7. Diámetro y ancho	34	6.18. Circuito euleriano	42
5.4.8. Smallest enclosing rectangle	35	7. Árboles	43
5.5. Par de puntos más cercanos	35	7.1. Estructura <code>tree</code>	43
5.6. Vantage Point Tree (puntos más cercanos a cada punto)	35	7.2. k -ésimo ancestro	43
5.7. Suma Minkowski	36	7.3. LCA	43
6. Grafos	37	7.4. Distancia entre dos nodos	44
6.1. Estructura <code>disjointSet</code>	37	7.5. HLD	44
6.2. Estructura <code>edge</code>	37	7.6. Link Cut	44
6.3. Estructura <code>path</code>	37	8. Flujos	44
6.4. Estructura <code>graph</code>	37	8.1. Estructura <code>flowEdge</code>	44
6.5. DFS genérica	38	8.2. Estructura <code>flowGraph</code>	44
6.6. Dijkstra con reconstrucción del camino más corto con menos vértices	38	8.3. Algoritmo de Edmonds-Karp $O(VE^2)$	45
6.7. Bellman Ford con reconstrucción del camino más corto con menos vértices	39	8.4. Algoritmo de Dinic $O(V^2E)$	45
6.8. Floyd	39	8.5. Flujo máximo de costo mínimo	46
6.9. Cerradura transitiva $O(V^3)$	39	9. Estructuras de datos	47
6.10. Cerradura transitiva $O(V^2)$	39	9.1. Segment Tree	47
6.11. Verificar si el grafo es bipartito	40	9.1.1. Point updates, range queries	47
6.12. Orden topológico	40	9.1.2. Dinamic with lazy propagation	47
		9.2. Fenwick Tree	48

9.3. Sqrt Decomposition	48
9.4. AVL Tree	49
9.5. Treap	52
9.6. Ordered Set C++	54
9.7. Splay Tree	54
9.8. Sparse table	54
9.9. Wavelet Tree	55
9.10. Red Black Tree	55
10. Cadenas	56
10.1. Trie	56
10.2. KMP	56
10.3. Aho-Corasick	57
10.4. Rabin-Karp	58
10.5. Suffix Array	58
10.6. Función Z	58
11. Varios	59
11.1. Lectura y escritura de <code>__int128</code>	59
11.2. Longest Common Subsequence (LCS)	59
11.3. Longest Increasing Subsequence (LIS)	59
11.4. Levenshtein Distance	60
11.5. Día de la semana	60
11.6. 2SAT	60
11.7. Código Gray	61

1. Teoría de números

1.1. Funciones básicas

1.1.1. Función piso y techo

```
lli piso(lli a, lli b){
    if((a >= 0 && b > 0) || (a < 0 && b < 0)){
        return a / b;
    }else{
        if(a % b == 0) return a / b;
        else return a / b - 1;
    }
}

lli techo(lli a, lli b){
    if((a >= 0 && b > 0) || (a < 0 && b < 0)){
        if(a % b == 0) return a / b;
        else return a / b + 1;
    }else{
        return a / b;
    }
}
```

1.1.2. Exponenciación y multiplicación binaria

```
lli pow(lli b, lli e){
    lli ans = 1;
    while(e){
        if(e & 1) ans *= b;
        e >>= 1;
        b *= b;
    }
    return ans;
}

lli multMod(lli a, lli b, lli n){
    lli ans = 0;
    a %= n, b %= n;
    if(abs(b) > abs(a)) swap(a, b);
```

```
    if(b < 0){
        a *= -1, b *= -1;
    }
    while(b){
        if(b & 1) ans = (ans + a) % n;
        b >>= 1;
        a = (a + a) % n;
    }
    return ans;
}
```

1.1.3. Mínimo común múltiplo y máximo común divisor

```
lli gcd(lli a, lli b){
    lli r;
    while(b != 0) r = a % b, a = b, b = r;
    return a;
}

lli lcm(lli a, lli b){
    return b * (a / gcd(a, b));
}

lli gcd(vector<lli> & nums){
    lli ans = 0;
    for(lli & num : nums) ans = gcd(ans, num);
    return ans;
}

lli lcm(vector<lli> & nums){
    lli ans = 1;
    for(lli & num : nums) ans = lcm(ans, num);
    return ans;
}
```

1.1.4. Euclides extendido e inverso modular

```
lli extendedGcd(lli a, lli b, lli & s, lli & t){
    lli q, r0 = a, r1 = b, ri, s0 = 1, s1 = 0, si, t0 = 0, t1 =
    ↪ 1, ti;
```

```

while(r1){
    q = r0 / r1;
    ri = r0 % r1, r0 = r1, r1 = ri;
    si = s0 - s1 * q, s0 = s1, s1 = si;
    ti = t0 - t1 * q, t0 = t1, t1 = ti;
}
s = s0, t = t0;
return r0;
}

lli modularInverse(lli a, lli m){
    lli r0 = a, r1 = m, ri, s0 = 1, s1 = 0, si;
    while(r1){
        si = s0 - s1 * (r0 / r1), s0 = s1, s1 = si;
        ri = r0 % r1, r0 = r1, r1 = ri;
    }
    if(r0 < 0) s0 *= -1;
    if(s0 < 0) s0 += m;
    return s0;
}

```

1.1.5. Todos los inversos módulo p

```

//number of ordered factorizations of n
lli orderedFactorizations(lli n){
    //skip the factorization if you already know the powers
    auto fact = factorize(n);
    int k = 0, q = 0;
    vector<int> powers(fact.size() + 1);
    for(auto & f : fact){

```

1.1.6. Exponenciación binaria modular

```

lli powMod(lli b, lli e, lli m){
    lli ans = 1;
    b %= m;
    if(e < 0){
        b = modularInverse(b, m);

```

```

        e *= -1;
    }
    while(e){
        if(e & 1) ans = (ans * b) % m;
        e >>= 1;
        b = (b * b) % m;
    }
    return ans;
}

```

1.1.7. Teorema chino del residuo

```

pair<lli, lli> chinese(vector<lli> & a, vector<lli> & n){
    lli prod = 1, p, ans = 0;
    for(lli & ni : n) prod *= ni;
    for(int i = 0; i < a.size(); ++i){
        p = prod / n[i];
        ans = (ans + (a[i] % n[i]) * modularInverse(p, n[i]) % prod
            ↪ * p) % prod;
    }
    if(ans < 0) ans += prod;
    return make_pair(ans, prod);
}

```

1.1.8. Coeficiente binomial

```

lli ncr(lli n, lli r){
    if(r < 0 || r > n) return 0;
    r = min(r, n - r);
    lli ans = 1;
    for(lli den = 1, num = n; den <= r; den++, num--){
        ans = ans * num / den;
    }
    return ans;
}

```

1.1.9. Fibonacci

```

//very fast fibonacci
inline void modula(lli & n){

```

```

    while(n >= mod) n -= mod;
}

lli fibo(lli n){
    array<lli, 2> F = {1, 0};
    lli p = 1;
    for(lli v = n; v >= 1; p <= 1);
    array<lli, 4> C;
    do{
        int d = (n & p) != 0;
        C[0] = C[3] = 0;
        C[d] = F[0] * F[0] % mod;
        C[d+1] = (F[0] * F[1] << 1) % mod;
        C[d+2] = F[1] * F[1] % mod;
        F[0] = C[0] + C[2] + C[3];
        F[1] = C[1] + C[2] + (C[3] << 1);
        modula(F[0]), modula(F[1]);
    }while(p >= 1);
    return F[1];
}

```

1.2. Cribas

1.2.1. Criba de divisores

```

vector<lli> divisorsSum;
vector<vector<int>> divisors;
void divisorsSieve(int n){
    divisorsSum.resize(n + 1, 0);
    divisors.resize(n + 1);
    for(int i = 1; i <= n; ++i){
        for(int j = i; j <= n; j += i){
            divisorsSum[j] += i;
            divisors[j].push_back(i);
        }
    }
}

```

1.2.2. Criba de primos

```

vector<int> primes;
vector<bool> isPrime;
void primesSieve(int n){
    isPrime.resize(n + 1, true);
    isPrime[0] = isPrime[1] = false;
    primes.push_back(2);
    for(int i = 4; i <= n; i += 2) isPrime[i] = false;
    int limit = sqrt(n);
    for(int i = 3; i <= n; i += 2){
        if(isPrime[i]){
            primes.push_back(i);
            if(i <= limit)
                for(int j = i * i; j <= n; j += 2 * i)
                    isPrime[j] = false;
        }
    }
}

```

1.2.3. Criba de factor primo más pequeño

```

vector<int> lowestPrime;
void lowestPrimeSieve(int n){
    lowestPrime.resize(n + 1, 1);
    lowestPrime[0] = lowestPrime[1] = 0;
    for(int i = 2; i <= n; ++i) lowestPrime[i] = (i & 1 ? i : 2);
    int limit = sqrt(n);
    for(int i = 3; i <= limit; i += 2)
        if(lowestPrime[i] == i)
            for(int j = i * i; j <= n; j += 2 * i)
                if(lowestPrime[j] == j) lowestPrime[j] = i;
}

```

1.2.4. Criba de factores primos

```

vector<vector<int>> primeFactors;
void primeFactorsSieve(lli n){
    primeFactors.resize(n + 1);
}

```

```

for(int i = 0; i < primes.size(); ++i){
    int p = primes[i];
    for(int j = p; j <= n; j += p)
        primeFactors[j].push_back(p);
}
}

```

1.2.5. Criba de la función φ de Euler

```

vector<int> Phi;
void phiSieve(int n){
    Phi.resize(n + 1);
    for(int i = 1; i <= n; ++i) Phi[i] = i;
    for(int i = 2; i <= n; ++i)
        if(Phi[i] == i)
            for(int j = i; j <= n; j += i)
                Phi[j] -= Phi[j] / i;
}

```

1.2.6. Criba de la función μ

```

vector<int> Mu;
void muSieve(int n){
    Mu.resize(n + 1, -1);
    Mu[0] = 0, Mu[1] = 1;
    for(int i = 2; i <= n; ++i)
        if(Mu[i])
            for(int j = 2*i; j <= n; j += i)
                Mu[j] -= Mu[i];
}

```

1.2.7. Triángulo de Pascal

```

vector<vector<lli>> Ncr;
void ncrSieve(lli n){
    Ncr.resize(n + 1);
    Ncr[0] = {1};
    for(lli i = 1; i <= n; ++i){
        Ncr[i].resize(i + 1);

```

```

        Ncr[i][0] = Ncr[i][i] = 1;
        for(lli j = 1; j <= i / 2; j++){
            Ncr[i][i - j] = Ncr[i][j] = Ncr[i - 1][j - 1] + Ncr[i - 1][j];
        }
    }
}

```

1.2.8. Segmented sieve

```

vector<int> segmented_sieve(int limit){
    const int L1D_CACHE_SIZE = 32768;
    int raiz = sqrt(limit);
    int segment_size = max(raiz, L1D_CACHE_SIZE);
    int s = 3, n = 3;
    vector<int> primes(1, 2), tmp, next;
    vector<char> sieve(segment_size);
    vector<bool> is_prime(raiz + 1, 1);
    for(int i = 2; i * i <= raiz; i++){
        if(is_prime[i])
            for(int j = i * i; j <= raiz; j += i)
                is_prime[j] = 0;
    }
    for(int low = 0; low <= limit; low += segment_size){
        fill(sieve.begin(), sieve.end(), 1);
        int high = min(low + segment_size - 1, limit);
        for(; s * s <= high; s += 2){
            if(is_prime[s]){
                tmp.push_back(s);
                next.push_back(s * s - low);
            }
        }
        for(size_t i = 0; i < tmp.size(); i++){
            int j = next[i];
            for(int k = tmp[i] * 2; j < segment_size; j += k)
                sieve[j] = 0;
            next[i] = j - segment_size;
        }
        for(; n <= high; n += 2)
            if(sieve[n - low])
                primes.push_back(n);
    }
}

```



```
    return primes;
}
```

1.2.9. Criba de primos lineal

```
vector<int> linearPrimeSieve(int n){
    vector<int> primes;
    vector<bool> isPrime(n+1, true);
    for(int i = 2; i <= n; ++i){
        if(isPrime[i])
            primes.push_back(i);
        for(int p : primes){
            int d = i * p;
            if(d > n) break;
            isPrime[d] = false;
            if(i % p == 0) break;
        }
    }
    return primes;
}
```

1.2.10. Criba lineal para funciones multiplicativas

```
//suppose f(n) is a multiplicative function and
//we want to find f(1), f(2), ..., f(n)
//we have f(pq) = f(p)f(q) if gcd(p, q) = 1
//and f(p^a) = g(p, a), where p is prime and a>0
vector<int> generalSieve(int n, function<int(int, int)> g){
    vector<int> f(n+1, 1), cnt(n+1), acum(n+1), primes;
    vector<bool> isPrime(n+1, true);
    for(int i = 2; i <= n; ++i){
        if(isPrime[i]){ //case base: f(p)
            primes.push_back(i);
            f[i] = g(i, 1);
            cnt[i] = 1;
            acum[i] = i;
        }
        for(int p : primes){
            int d = i * p;
```

```
            if(d > n) break;
            isPrime[d] = false;
            if(i % p == 0){ //gcd(i, p) != 1
                f[d] = f[i / acum[i]] * g(p, cnt[i] + 1);
                cnt[d] = cnt[i] + 1;
                acum[d] = acum[i] * p;
                break;
            }else{ //gcd(i, p) = 1
                f[d] = f[i] * g(p, 1);
                cnt[d] = 1;
                acum[d] = p;
            }
        }
    }
    return f;
}
```

1.3. Factorización

1.3.1. Factorización de un número

```
vector<pair<lli, int>> factorize(lli n){
    vector<pair<lli, int>> f;
    for(lli p : primes){
        if(p * p > n) break;
        int pot = 0;
        while(n % p == 0){
            pot++;
            n /= p;
        }
        if(pot) f.emplace_back(p, pot);
    }
    if(n > 1) f.emplace_back(n, 1);
    return f;
}
```

1.3.2. Potencia de un primo que divide a un factorial

```
lli potInFactorial(lli n, lli p){
    lli ans = 0, div = n;
    while(div /= p) ans += div;
    return ans;
}
```

1.3.3. Factorización de un factorial

```
vector<pair<lli, lli>> factorizeFactorial(lli n){
    vector<pair<lli, lli>> f;
    for(lli p : primes){
        if(p > n) break;
        f.emplace_back(p, potInFactorial(n, p));
    }
    return f;
}
```

1.3.4. Factorización usando Pollard-Rho

```
bool isPrimeMillerRabin(lli n){
    if(n < 2) return false;
    if(n == 2) return true;
    lli d = n - 1, s = 0;
    while(!(d & 1)){
        d >>= 1;
        ++s;
    }
    for(int i = 0; i < 16; ++i){
        lli a = 1 + rand() % (n - 1);
        lli m = powMod(a, d, n);
        if(m == 1 || m == n - 1) goto exit;
        for(int k = 0; k < s - 1; ++k){
            m = m * m % n;
            if(m == n - 1) goto exit;
        }
        return false;
    }
    exit:;
}
```

```
    }
    return true;
}

lli getFactor(lli n){
    lli a = 1 + rand() % (n - 1);
    lli b = 1 + rand() % (n - 1);
    lli x = 2, y = 2, d = 1;
    while(d == 1 || d == -1){
        x = x * (x + b) % n + a;
        y = y * (y + b) % n + a;
        y = y * (y + b) % n + a;
        d = gcd(x - y, n);
    }
    return abs(d);
}

map<lli, int> fact; //don't forget to clean
void factorizePollardRho(lli n){
    while(n > 1 && !isPrimeMillerRabin(n)){
        lli f;
        do{
            f = getFactor(n);
        }while(f == n);
        n /= f;
        factorizePollardRho(f);
        for(auto & it : fact){
            while(n % it.first == 0){
                n /= it.first;
                ++it.second;
            }
        }
    }
    if(n > 1) ++fact[n];
}
```

1.4. Funciones multiplicativas famosas

1.4.1. Función σ

```
//divisor power sum of n
//if pot=0 we get the number of divisors
//if pot=1 we get the sum of divisors
lli sigma(lli n, lli pot){
    lli ans = 1;
    auto f = factorize(n);
    for(auto & factor : f){
        lli p = factor.first;
        int a = factor.second;
        if(pot){
            lli p_pot = pow(p, pot);
            ans *= (pow(p_pot, a + 1) - 1) / (p_pot - 1);
        }else{
            ans *= a + 1;
        }
    }
    return ans;
}
```

1.4.2. Función Ω

```
//number of total primes with multiplicity dividing n
int Omega(lli n){
    int ans = 0;
    auto f = factorize(n);
    for(auto & factor : f)
        ans += factor.second;
    return ans;
}
```

1.4.3. Función ω

```
//number of distinct primes dividing n
int omega(lli n){
    int ans = 0;
```

```
    auto f = factorize(n);
    for(auto & factor : f)
        ++ans;
    return ans;
}
```

1.4.4. Función φ de Euler

```
//number of coprimes with n less than n
lli phi(lli n){
    lli ans = n;
    auto f = factorize(n);
    for(auto & factor : f)
        ans -= ans / factor.first;
    return ans;
}
```

1.4.5. Función μ

```
//1 if n is square-free with an even number of prime factors
//-1 if n is square-free with an odd number of prime factors
//0 is n has a square prime factor
int mu(lli n){
    int ans = 1;
    auto f = factorize(n);
    for(auto & factor : f){
        if(factor.second > 1) return 0;
        ans *= -1;
    }
    return ans;
}
```

1.5. Orden multiplicativo, raíces primitivas y raíces de la unidad

1.5.1. Función λ de Carmichael

```
//the smallest positive integer k such that for
//every coprime x with n, x^k=1 mod n
lli carmichaelLambda(lli n){
    lli ans = 1;
    auto f = factorize(n);
    for(auto & factor : f){
        lli p = factor.first;
        int a = factor.second;
        lli tmp = pow(p, a);
        tmp -= tmp / p;
        if(a <= 2 || p >= 3) ans = lcm(ans, tmp);
        else ans = lcm(ans, tmp >> 1);
    }
    return ans;
}
```

1.5.2. Orden multiplicativo módulo m

```
// the smallest positive integer k such that x^k = 1 mod m
lli multiplicativeOrder(lli x, lli m){
    if(gcd(x, m) != 1) return -1;
    lli order = phi(m);
    auto f = factorize(order);
    for(auto & factor : f){
        lli p = factor.first;
        int a = factor.second;
        order /= pow(p, a);
        lli tmp = powMod(x, order, m);
        while(tmp != 1){
            tmp = powMod(tmp, p, m);
            order *= p;
        }
    }
    return order;
}
```

1.5.3. Número de raíces primitivas (generadores) módulo m

```
//number of generators modulo m
lli numberOfGenerators(lli m){
    lli phi_m = phi(m);
    lli lambda_m = carmichaelLambda(m);
    if(phi_m == lambda_m) return phi(phi_m);
    else return 0;
}
```

1.5.4. Test individual de raíz primitiva módulo m

```
//test if order(x, m) = phi(m), i.e., x is a generator for Z/mZ
bool testPrimitiveRoot(lli x, lli m){
    if(gcd(x, m) != 1) return false;
    lli order = phi(m);
    auto f = factorize(order);
    for(auto & factor : f){
        lli p = factor.first;
        if(powMod(x, order / p, m) == 1) return false;
    }
    return true;
}
```

1.5.5. Test individual de raíz k -ésima de la unidad módulo m

```
//test if x^k = 1 mod m and k is the smallest for such x, i.e.,
↪ x^(k/p) != 1 for every prime divisor of k
bool testPrimitiveKthRootUnity(lli x, lli k, lli m){
    if(powMod(x, k, m) != 1) return false;
    auto f = factorize(k);
    for(auto & factor : f){
        lli p = factor.first;
        if(powMod(x, k / p, m) == 1) return false;
    }
    return true;
}
```

1.5.6. Encontrar la primera raíz primitiva módulo m

```
lli findFirstGenerator(lli m){
    lli order = phi(m);
    if(order != carmichaelLambda(m)) return -1; //just an
    → optimization, not required
    auto f = factorize(order);
    for(lli x = 1; x < m; x++){
        if(gcd(x, m) != 1) continue;
        bool test = true;
        for(auto & factor : f){
            lli p = factor.first;
            if(powMod(x, order / p, m) == 1){
                test = false;
                break;
            }
        }
        if(test) return x;
    }
    return -1;
}
```

1.5.7. Encontrar la primera raíz k -ésima de la unidad módulo m

```
lli findFirstPrimitiveKthRootUnity(lli k, lli m){
    if(carmichaelLambda(m) % k != 0) return -1; //just an
    → optimization, not required
    auto f = factorize(k);
    for(lli x = 1; x < m; x++){
        if(powMod(x, k, m) != 1) continue;
        bool test = true;
        for(auto & factor : f){
            lli p = factor.first;
            if(powMod(x, k / p, m) == 1){
                test = false;
                break;
            }
        }
        if(test) return x;
    }
}
```

```
}
    return -1;
}
```

1.5.8. Logaritmo discreto

```
// a^x = b mod m, a and m coprime
pair<lli, lli> discreteLogarithm(lli a, lli b, lli m){
    if(gcd(a, m) != 1) return make_pair(-1, 0);
    lli order = multiplicativeOrder(a, m);
    lli n = sqrt(order) + 1;
    lli a_n = powMod(a, n, m);
    lli ans = 0;
    unordered_map<lli, lli> firstHalf;
    lli current = a_n;
    for(lli p = 1; p <= n; p++){
        firstHalf[current] = p;
        current = (current * a_n) % m;
    }
    current = b % m;
    for(lli q = 0; q <= n; q++){
        if(firstHalf.count(current)){
            lli p = firstHalf[current];
            lli x = n * p - q;
            return make_pair(x % order, order);
        }
        current = (current * a) % m;
    }
    return make_pair(-1, 0);
}
```

1.5.9. Raíz k -ésima discreta

```
// x^k = b mod m, m has at least one generator
vector<lli> discreteRoot(lli k, lli b, lli m){
    if(b % m == 0) return {0};
    lli g = findFirstGenerator(m);
    lli power = powMod(g, k, m);
    auto y0 = discreteLogarithm(power, b, m);
```

```

    if(y0.first == -1) return {};
    lli phi_m = phi(m);
    lli d = gcd(k, phi_m);
    vector<lli> x(d);
    x[0] = powMod(g, y0.first, m);
    lli inc = powMod(g, phi_m / d, m);
    for(lli i = 1; i < d; i++)
        x[i] = x[i - 1] * inc % m;
    sort(x.begin(), x.end());
    return x;
}

```

1.6. Particiones

1.6.1. Función P (particiones de un entero positivo)

```

lli mod = 1e9 + 7;

vector<lli> P;

//number of ways to write n as a sum of positive integers
lli partitionsP(int n){
    if(n < 0) return 0;
    if(P[n]) return P[n];
    int pos1 = 1, pos2 = 2, inc1 = 4, inc2 = 5;
    lli ans = 0;
    for(int k = 1; k <= n; k++){
        lli tmp = (n >= pos1 ? P[n - pos1] : 0) + (n >= pos2 ? P[n
        ↪ - pos2] : 0);
        if(k & 1) ans += tmp;
        else ans -= tmp;
        if(n < pos2) break;
        pos1 += inc1, pos2 += inc2;
        inc1 += 3, inc2 += 3;
    }
    ans %= mod;
    if(ans < 0) ans += mod;
    return ans;
}

```

```

void calculateFunctionP(int n){
    P.resize(n + 1);
    P[0] = 1;
    for(int i = 1; i <= n; i++)
        P[i] = partitionsP(i);
}

```

1.6.2. Función Q (particiones de un entero positivo en distintos sumandos)

```

vector<lli> Q;

bool isPerfectSquare(int n){
    int r = sqrt(n);
    return r * r == n;
}

int s(int n){
    int r = 1 + 24 * n;
    if(isPerfectSquare(r)){
        int j;
        r = sqrt(r);
        if((r + 1) % 6 == 0) j = (r + 1) / 6;
        else j = (r - 1) / 6;
        if(j & 1) return -1;
        else return 1;
    }else{
        return 0;
    }
}

//number of ways to write n as a sum of distinct positive
↪ integers
//number of ways to write n as a sum of odd positive integers
lli partitionsQ(int n){
    if(n < 0) return 0;
    if(Q[n]) return Q[n];
    int pos = 1, inc = 3;
    lli ans = 0;
    int limit = sqrt(n);

```

```

for(int k = 1; k <= limit; k++){
    if(k & 1) ans += Q[n - pos];
    else ans -= Q[n - pos];
    pos += inc;
    inc += 2;
}
ans <= 1;
ans += s(n);
ans %= mod;
if(ans < 0) ans += mod;
return ans;
}

```

```

void calculateFunctionQ(int n){
    Q.resize(n + 1);
    Q[0] = 1;
    for(int i = 1; i <= n; i++)
        Q[i] = partitionsQ(i);
}

```

1.6.3. Número de factorizaciones ordenadas

```

//number of ordered factorizations of n
lli orderedFactorizations(lli n){
    //skip the factorization if you already know the powers
    auto fact = factorize(n);
    int k = 0, q = 0;
    vector<int> powers(fact.size() + 1);
    for(auto & f : fact){
        powers[k + 1] = f.second;
        q += f.second;
        ++k;
    }
    vector<lli> prod(q + 1, 1);
    //we need Ncr until the max_power+Omega(n) row
    //module if needed
    for(int i = 0; i <= q; i++){
        for(int j = 1; j <= k; j++){
            prod[i] = prod[i] * Ncr[powers[j] + i][powers[j]];
        }
    }
}

```

```

}
lli ans = 0;
for(int j = 1; j <= q; j++){
    int alt = 1;
    for(int i = 0; i < j; i++){
        ans = ans + alt * Ncr[j][i] * prod[j - i - 1];
        alt *= -1;
    }
}
return ans;
}

```

1.6.4. Número de factorizaciones no ordenadas

```

//Number of unordered factorizations of n with
//largest part at most m
//Call unorderedFactorizations(n, n) to get all of them
//Add this to the main to speed up the map:
//mem.reserve(1024); mem.max_load_factor(0.25);
struct HASH{
    size_t operator()(const pair<int,int>&x) const{
        return hash<long long>()(((long long)x.first)^(((long
        ↪ long)x.second)<<32));
    }
};
unordered_map<pair<int, int>, lli, HASH> mem;
lli unorderedFactorizations(int m, int n){
    if(m == 1 && n == 1) return 1;
    if(m == 1) return 0;
    if(n == 1) return 1;
    if(mem.count({m, n})) return mem[{m, n}];
    lli ans = 0;
    int l = sqrt(n);
    for(int i = 1; i <= l; ++i){
        if(n % i == 0){
            int a = i, b = n / i;
            if(a <= m) ans += unorderedFactorizations(a, b);
            if(a != b && b <= m) ans += unorderedFactorizations(b,
            ↪ a);
        }
    }
}

```

```

    }
    return mem[{m, n}] = ans;
}

```

1.7. Otros

1.7.1. Cambio de base

```

string decimalToBaseB(lli n, lli b){
    string ans = "";
    lli d;
    do{
        d = n % b;
        if(0 <= d && d <= 9) ans = (char)(48 + d) + ans;
        else if(10 <= d && d <= 35) ans = (char)(55 + d) + ans;
        n /= b;
    }while(n != 0);
    return ans;
}

lli baseBtoDecimal(const string & n, lli b){
    lli ans = 0;
    for(const char & d : n){
        if(48 <= d && d <= 57) ans = ans * b + (d - 48);
        else if(65 <= d && d <= 90) ans = ans * b + (d - 55);
        else if(97 <= d && d <= 122) ans = ans * b + (d - 87);
    }
    return ans;
}

```

1.7.2. Fracciones continuas

```

//continued fraction of (p+sqrt(n))/q, where p,n,q are positive
//  integers
//returns a vector of terms and the length of the period,
//the periodic part is taken from the right of the array
pair<vector<lli>, int> ContinuedFraction(lli p, lli n, lli q){
    vector<lli> coef;
    lli r = sqrt(n);

```

```

    //Skip this if you know that n is not a perfect square
    if(r * r == n){
        lli num = p + r;
        lli den = q;
        lli residue;
        while(den){
            residue = num % den;
            coef.push_back(num / den);
            num = den;
            den = residue;
        }
        return make_pair(coef, 0);
    }
    if((n - p * p) % q != 0){
        n *= q * q;
        p *= q;
        q *= q;
        r = sqrt(n);
    }
    lli a = (r + p) / q;
    coef.push_back(a);
    int period = 0;
    map<pair<lli, lli>, int> pairs;
    while(true){
        p = a * q - p;
        q = (n - p * p) / q;
        a = (r + p) / q;
        //if p=0 and q=1, we can just ask if q==1 after inserting a
        if(pairs.count(make_pair(p, q))){
            period -= pairs[make_pair(p, q)];
            break;
        }
        coef.push_back(a);
        pairs[make_pair(p, q)] = period++;
    }
    return make_pair(coef, period);
}

```


1.7.3. Ecuación de Pell

```
//first solution (x, y) to the equation  $x^2 - ny^2 = 1$ ,  $n$  IS NOT a
↪ perfect square
pair<lli, lli> PellEquation(lli n){
    vector<lli> cf = ContinuedFraction(0, n, 1).first;
    lli num = 0, den = 1;
    int k = cf.size() - 1;
    for(int i = ((k & 1) ? (2 * k - 1) : (k - 1)); i >= 0; i--){
        lli tmp = den;
        int pos = i % k;
        if(pos == 0 && i != 0) pos = k;
        den = num + cf[pos] * den;
        num = tmp;
    }
    return make_pair(den, num);
}
```

1.7.4. Números de Bell

```
Bell.resize(n + 1);
Bell[0] = {1};
for(int i = 1; i <= n; ++i){
    Bell[i].resize(i + 1);
    Bell[i][0] = Bell[i - 1][i - 1];
    for(int j = 1; j <= i; ++j)
        Bell[i][j] = Bell[i][j - 1] + Bell[i - 1][j - 1];
}

ostream &operator<<(ostream &os, const __int128 & value){
    char buffer[64];
    char *pos = end(buffer) - 1;
```

2. Números racionales

2.1. Estructura fraccion

```
struct fraccion{
    ll num, den;
    fraccion(){
        num = 0, den = 1;
    }
    fraccion(ll x, ll y){
        if(y < 0)
            x *= -1, y *= -1;
        ll d = __gcd(abs(x), abs(y));
        num = x/d, den = y/d;
    }
    fraccion(ll v){
        num = v;
        den = 1;
    }
    fraccion operator+(const fraccion& f) const{
        ll d = __gcd(den, f.den);
        return fraccion(num*(f.den/d) + f.num*(den/d),
            ↪ den*(f.den/d));
    }
    fraccion operator-() const{
        return fraccion(-num, den);
    }
    fraccion operator-(const fraccion& f) const{
        return *this + (-f);
    }
    fraccion operator*(const fraccion& f) const{
        return fraccion(num*f.num, den*f.den);
    }
    fraccion operator/(const fraccion& f) const{
        return fraccion(num*f.den, den*f.num);
    }
    fraccion operator+=(const fraccion& f){
        *this = *this + f;
        return *this;
    }
    fraccion operator-=(const fraccion& f){
```

```

    *this = *this - f;
    return *this;
}
fraccion operator++(int xd){
    *this = *this + 1;
    return *this;
}
fraccion operator--(int xd){
    *this = *this - 1;
    return *this;
}
fraccion operator*=(const fraccion& f){
    *this = *this * f;
    return *this;
}
fraccion operator/=(const fraccion& f){
    *this = *this / f;
    return *this;
}
bool operator==(const fraccion& f) const{
    ll d = __gcd(den, f.den);
    return (num*(f.den/d) == (den/d)*f.num);
}
bool operator!=(const fraccion& f) const{
    ll d = __gcd(den, f.den);
    return (num*(f.den/d) != (den/d)*f.num);
}
bool operator >(const fraccion& f) const{
    ll d = __gcd(den, f.den);
    return (num*(f.den/d) > (den/d)*f.num);
}
bool operator <(const fraccion& f) const{
    ll d = __gcd(den, f.den);
    return (num*(f.den/d) < (den/d)*f.num);
}
bool operator >=(const fraccion& f) const{
    ll d = __gcd(den, f.den);
    return (num*(f.den/d) >= (den/d)*f.num);
}
bool operator <=(const fraccion& f) const{
    ll d = __gcd(den, f.den);

```

```

        return (num*(f.den/d) <= (den/d)*f.num);
    }
    fraccion inverso() const{
        return fraccion(den, num);
    }
    fraccion fabs() const{
        fraccion nueva;
        nueva.num = abs(num);
        nueva.den = den;
        return nueva;
    }
    double value() const{
        return (double)num / (double)den;
    }
    string str() const{
        stringstream ss;
        ss << num;
        if(den != 1) ss << "/" << den;
        return ss.str();
    }
};

ostream &operator<<(ostream &os, const fraccion &f) {
    return os << f.str();
}

istream &operator>>(istream &is, fraccion &f){
    ll num = 0, den = 1;
    string str;
    is >> str;
    size_t pos = str.find("/");
    if(pos == string::npos){
        istringstream(str) >> num;
    }else{
        istringstream(str.substr(0, pos)) >> num;
        istringstream(str.substr(pos + 1)) >> den;
    }
    f = fraccion(num, den);
    return is;
}

```

3. Álgebra lineal

3.1. Estructura matrix

```
template <typename entrada>
struct matrix{
    vector< vector<entrada> > A;
    int m, n;

    matrix(int _m, int _n){
        m = _m, n = _n;
        A.resize(m, vector<entrada>(n, 0));
    }

    vector<entrada> & operator[] (int i){
        return A[i];
    }

    void multiplicarFilaPorEscalar(int k, entrada c){
        for(int j = 0; j < n; j++) A[k][j] *= c;
    }

    void intercambiarFilas(int k, int l){
        swap(A[k], A[l]);
    }

    void sumaMultiploFilaAOtra(int k, int l, entrada c){
        for(int j = 0; j < n; j++) A[k][j] += c * A[l][j];
    }

    matrix operator+(const matrix & B) const{
        if(m == B.m && n == B.n){
            matrix<entrada> C(m, n);
            for(int i = 0; i < m; i++){
                for(int j = 0; j < n; j++){
                    C[i][j] = A[i][j] + B.A[i][j];
                }
            }
            return C;
        }else{
```

```
            return *this;
        }
    }

    matrix operator+=(const matrix & M){
        *this = *this + M;
        return *this;
    }

    matrix operator-() const{
        matrix<entrada> C(m, n);
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                C[i][j] = -A[i][j];
            }
        }
        return C;
    }

    matrix operator-(const matrix & B) const{
        return *this + (-B);
    }

    matrix operator-=(const matrix & M){
        *this = *this + (-M);
        return *this;
    }

    matrix operator*(const matrix & B) const{
        if(n == B.m){
            matrix<entrada> C(m, B.n);
            for(int i = 0; i < m; i++){
                for(int j = 0; j < B.n; j++){
                    for(int k = 0; k < n; k++){
                        C[i][j] += A[i][k] * B.A[k][j];
                    }
                }
            }
            return C;
        }else{
            return *this;
        }
    }
}
```

```

    }
}

matrix operator*(const entrada & c) const{
    matrix<entrada> C(m, n);
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            C[i][j] = A[i][j] * c;
        }
    }
    return C;
}

matrix operator*=(const matrix & M){
    *this = *this * M;
    return *this;
}

matrix operator*=(const entrada & c){
    *this = *this * c;
    return *this;
}

matrix operator^(lli b) const{
    matrix<entrada> ans = matrix<entrada>::identidad(n);
    matrix<entrada> A = *this;
    while(b){
        if(b & 1) ans *= A;
        b >>= 1;
        if(b) A *= A;
    }
    return ans;
}

matrix operator^(lli n){
    *this = *this ^ n;
    return *this;
}

bool operator==(const matrix & B) const{
    if(m == B.m && n == B.n){

```

```

        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if(A[i][j] != B.A[i][j]) return false;
            }
        }
        return true;
    }else{
        return false;
    }
}

bool operator!=(const matrix & B) const{
    return !(*this == B);
}

```

3.2. Gauss Jordan

```

//For every elemental operation that we apply to the matrix,
//we will call to callback(operation, source row, dest row,
↪ value).
//It returns the rank of the matrix, and modifies it
int gauss_jordan(bool full = true, bool makeOnes = true,
↪ function<void(int, int, int, entrada)>callback = NULL){
    int i = 0, j = 0;
    while(i < m && j < n){
        if(A[i][j] == 0){
            for(int f = i + 1; f < m; f++){
                if(A[f][j] != 0){
                    intercambiarFilas(i, f);
                    if(callback) callback(2, i, f, 0);
                    break;
                }
            }
        }
        if(A[i][j] != 0){
            entrada inv_mult = A[i][j].inverso();
            if(makeOnes && A[i][j] != 1){
                multiplicarFilaPorEscalar(i, inv_mult);
                if(callback) callback(1, i, 0, inv_mult);
            }

```

```

    for(int f = (full ? 0 : (i + 1)); f < m; f++){
        if(f != i && A[f][j] != 0){
            entrada inv_adit = -A[f][j];
            if(!makeOnes) inv_adit *= inv_mult;
            sumaMultiploFilaA0tra(f, i, inv_adit);
            if(callback) callback(3, f, i, inv_adit);
        }
    }
    i++;
}
j++;
}
return i;
}

void eliminacion_gaussiana(){
    gauss_jordan(false);
}

```

3.3. Matriz inversa

```

static matrix identidad(int n){
    matrix<entrada> id(n, n);
    for(int i = 0; i < n; i++){
        id[i][i] = 1;
    }
    return id;
}

matrix<entrada> inversa(){
    if(m == n){
        matrix<entrada> tmp = *this;
        matrix<entrada> inv = matrix<entrada>::identidad(n);
        auto callback = [&](int op, int a, int b, entrada e){
            if(op == 1){
                inv.multiplicarFilaPorEscalar(a, e);
            }else if(op == 2){
                inv.intercambiarFilas(a, b);
            }else if(op == 3){
                inv.sumaMultiploFilaA0tra(a, b, e);
            }
        };
    }
}

```

```

    }
};
if(tmp.gauss_jordan(true, true, callback) == n){
    return inv;
}else{
    return *this;
}
}else{
    return *this;
}
}
}

```

3.4. Transpuesta

```

matrix<entrada> transpuesta(){
    matrix<entrada> T(n, m);
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            T[j][i] = A[i][j];
        }
    }
    return T;
}

```

3.5. Traza

```

entrada traza(){
    entrada sum = 0;
    for(int i = 0; i < min(m, n); i++){
        sum += A[i][i];
    }
    return sum;
}

```

3.6. Determinante

```

entrada determinante(){
    if(m == n){

```

```

matrix<entrada> tmp = *this;
entrada det = 1;
auto callback = [&](int op, int a, int b, entrada e){
    if(op == 1){
        det /= e;
    }else if(op == 2){
        det *= -1;
    }
};
if(tmp.gauss_jordan(false, true, callback) != n) det = 0;
return det;
}else{
    return 0;
}
}

```

3.7. Matriz de cofactores y adjunta

```

matrix<entrada> menor(int x, int y){
    matrix<entrada> M(0, 0);
    for(int i = 0; i < m; i++){
        if(i != x){
            M.A.push_back(vector<entrada>());
            for(int j = 0; j < n; j++){
                if(j != y){
                    M.A.back().push_back(A[i][j]);
                }
            }
        }
    }
    M.m = m - 1;
    M.n = n - 1;
    return M;
}

entrada cofactor(int x, int y){
    entrada ans = menor(x, y).determinante();
    if((x + y) % 2 == 1) ans *= -1;
    return ans;
}

```

```

matrix<entrada> cofactores(){
    matrix<entrada> C(m, n);
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            C[i][j] = cofactor(i, j);
        }
    }
    return C;
}

matrix<entrada> adjunta(){
    return cofactores().transpuesta();
}

```

3.8. Factorización $PA = LU$

```

vector< matrix<entrada> > PA_LU(){
    matrix<entrada> U = *this;
    matrix<entrada> L = matrix<entrada>::identidad(n);
    matrix<entrada> P = matrix<entrada>::identidad(n);
    auto callback = [&](int op, int a, int b, entrada e){
        if(op == 2){
            L.intercambiarFilas(a, b);
            P.intercambiarFilas(a, b);
            L.A[a][a] = L.A[b][b] = 1;
            L.A[a][a + 1] = L.A[b][b - 1] = 0;
        }else if(op == 3){
            L.A[a][b] = -e;
        }
    };
    U.gauss_jordan(false, false, callback);
    return {P, L, U};
}

```

3.9. Polinomio característico

```

vector<entrada> polinomio(){
    matrix<entrada> M(n, n);
}

```

```

vector<entrada> coef(n + 1);
matrix<entrada> I = matrix<entrada>::identidad(n);
coef[n] = 1;
for(int i = 1; i <= n; i++){
    M = (*this) * M + I * coef[n - i + 1];
    coef[n - i] = -((*this) * M).traza() / i;
}
return coef;
}

```

3.10. Gram-Schmidt

```

matrix<entrada> gram_schmidt(){ //los vectores son las filas
↳ de la matriz
    matrix<entrada> B = (*this) * (*this).transpuesta();
    matrix<entrada> ans = *this;
    auto callback = [&](int op, int a, int b, entrada e){
        if(op == 1){
            ans.multiplicarFilaPorEscalar(a, e);
        }else if(op == 2){
            ans.intercambiarFilas(a, b);
        }else if(op == 3){
            ans.sumaMultiploFilaAotra(a, b, e);
        }
    };
    B.gauss_jordan(false, false, callback);
    return ans;
}

```

3.11. Recurrencias lineales

```

//Solves a linear recurrence relation of degree d of the form
//F(n) = a(d-1)*F(n-1) + a(d-2)*F(n-2) + ... + a(1)*F(n-(d-1))
↳ + a(0)*F(n-d)
//with initial values F(0), F(1), ..., F(d-1)
//It finds the nth term of the recurrence, F(n)
//The values of a[0,...,d) are in the array P[]
lli solveRecurrence(lli *P, lli *init, int deg, lli n){
    lli *ans = new lli[deg]();
}

```

```

lli *R = new lli[2*deg]();
ans[0] = 1;
lli p = 1;
for(lli v = n; v >= 1; p <= 1);
do{
    int d = (n & p) != 0;
    fill(R, R + 2*deg, 0);
    //if deg(mod-1)^2 overflows, just do mod in the
    ↳ multiplications
    for(int i = 0; i < deg; i++){
        for(int j = 0; j < deg; j++){
            R[i + j + d] += ans[i] * ans[j];
        }
    }
    for(int i = 0; i < 2*deg; ++i) R[i] %= mod;
    for(int i = deg-1; i >= 0; i--){
        R[i + deg] %= mod;
        for(int j = 0; j < deg; j++){
            R[i + j] += R[i + deg] * P[j];
        }
    }
    for(int i = 0; i < deg; i++) R[i] %= mod;
    copy(R, R + deg, ans);
}while(p >= 1);
lli nValue = 0;
for(int i = 0; i < deg; i++){
    nValue += ans[i] * init[i];
}
return nValue % mod;
}

```

3.12. Simplex

```

/*
Parametric Self-Dual Simplex method
Solve a canonical LP:
    min or max. c x
    s.t. A x <= b
        x >= 0
*/
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-9, oo =
↳ numeric_limits<double>::infinity();

```

```

typedef vector<double> vec;
typedef vector<vec> mat;

pair<vec, double> simplexMethodPD(mat &A, vec &b, vec &c, bool
↪ mini = true){
    int n = c.size(), m = b.size();
    mat T(m + 1, vec(n + m + 1));
    vector<int> base(n + m), row(m);

    for(int j = 0; j < m; ++j){
        for(int i = 0; i < n; ++i)
            T[j][i] = A[j][i];
        row[j] = n + j;
        T[j][n + j] = 1;
        base[n + j] = 1;
        T[j][n + m] = b[j];
    }

    for(int i = 0; i < n; ++i)
        T[m][i] = c[i] * (mini ? 1 : -1);

    while(true){
        int p = 0, q = 0;
        for(int i = 0; i < n + m; ++i)
            if(T[m][i] <= T[m][p])
                p = i;

        for(int j = 0; j < m; ++j)
            if(T[j][n + m] <= T[q][n + m])
                q = j;

        double t = min(T[m][p], T[q][n + m]);

        if(t >= -eps){
            vec x(n);
            for(int i = 0; i < m; ++i)
                if(row[i] < n) x[row[i]] = T[i][n + m];
            return {x, T[m][n + m] * (mini ? -1 : 1)}; // optimal
        }

        if(t < T[q][n + m]){
            // tight on c -> primal update
            for(int j = 0; j < m; ++j)
                if(T[j][p] >= eps)
                    if(T[j][p] * (T[q][n + m] - t) >= T[q][p] * (T[j][n +
↪ m] - t))
                        q = j;

            if(T[q][p] <= eps)
                return {vec(n), oo * (mini ? 1 : -1)}; // primal
↪ infeasible
        }else{
            // tight on b -> dual update
            for(int i = 0; i < n + m + 1; ++i)
                T[q][i] = -T[q][i];

            for(int i = 0; i < n + m; ++i)
                if(T[q][i] >= eps)
                    if(T[q][i] * (T[m][p] - t) >= T[q][p] * (T[m][i] -
↪ t))
                        p = i;

            if(T[q][p] <= eps)
                return {vec(n), oo * (mini ? -1 : 1)}; // dual
↪ infeasible
        }

        for(int i = 0; i < m + n + 1; ++i)
            if(i != p) T[q][i] /= T[q][p];

        T[q][p] = 1; // pivot(q, p)
        base[p] = 1;
        base[row[q]] = 0;
        row[q] = p;

        for(int j = 0; j < m + 1; ++j){
            if(j != q){
                double alpha = T[j][p];
                for(int i = 0; i < n + m + 1; ++i)
                    T[j][i] -= T[q][i] * alpha;
            }
        }
    }

```



```

    }
}

return {vec(n), oo};
}

int main(){
    int m, n;
    bool mini = true;
    cout << "Numero de restricciones: ";
    cin >> m;
    cout << "Numero de incognitas: ";
    cin >> n;
    mat A(m, vec(n));
    vec b(m), c(n);
    for(int i = 0; i < m; ++i){
        cout << "Restriccion #" << (i + 1) << ": ";
        for(int j = 0; j < n; ++j){
            cin >> A[i][j];
        }
        cin >> b[i];
    }
    cout << "[0]Max o [1]Min?: ";
    cin >> mini;
    cout << "Coeficientes de " << (mini ? "min" : "max") << " z: ";
    cin >> c;
    for(int i = 0; i < n; ++i){
        cin >> c[i];
    }
    cout.precision(6);
    auto ans = simplexMethodPD(A, b, c, mini);
    cout << (mini ? "Min" : "Max") << " z = " << ans.second << ", ";
    cout << "cuando: \n";
    for(int i = 0; i < ans.first.size(); ++i){
        cout << "x_" << (i + 1) << " = " << ans.first[i] << "\n";
    }
    return 0;
}

```

4. FFT

4.1. Funciones previas

```

typedef complex<double> comp;
typedef long long int lli;
double PI = acos(-1.0);

```

```

int nearestPowerOfTwo(int n){
    int ans = 1;
    while(ans < n) ans <= 1;
    return ans;
}

```

4.2. FFT con raíces de la unidad complejas

```

void fft(vector<comp> & X, int inv){
    int n = X.size();
    int len, len2, i, j, k;
    for(i = 1; j = 0; i < n - 1; ++i){
        for (k = n >> 1; (j ^= k) < k; k >>= 1);
        if (i < j) swap(X[i], X[j]);
    }
    double ang;
    comp t, u, v;
    vector<comp> wlen_pw(n >> 1);
    wlen_pw[0] = 1;
    for(len = 2; len <= n; len <= 1){
        ang = inv == -1 ? -2 * PI / len : 2 * PI / len;
        len2 = len >> 1;
        comp wlen(cos(ang), sin(ang));
        for(i = 1; i < len2; ++i){
            wlen_pw[i] = wlen_pw[i - 1] * wlen;
        }
        for(i = 0; i < n; i += len){
            for(j = 0; j < len2; ++j){
                t = X[i + j + len2] * wlen_pw[j];
                X[i + j + len2] = X[i + j] - t;
                X[i + j] += t;
            }
        }
    }
}

```

```

    }
}
if(inv == -1){
    for(i = 0; i < n; ++i){
        X[i] /= n;
    }
}
}
}

```

4.3. FFT con raíces de la unidad discretas (NTT)

```

int inverse(int a, int n){
    int r0 = a, r1 = n, ri, s0 = 1, s1 = 0, si;
    while(r1){
        si = s0 - s1 * (r0 / r1), s0 = s1, s1 = si;
        ri = r0 % r1, r0 = r1, r1 = ri;
    }
    if(s0 < 0) s0 += n;
    return s0;
}

```

```

const int p = 7340033;
const int root = 5;
const int root_1 = inverse(root, p);
const int root_pw = 1 << 20;

void ntt(vector<int> & X, int inv){
    int n = X.size();
    int len, len2, wlen, i, j, k, u, v, w;
    for(i = 1, j = 0; i < n - 1; ++i){
        for(k = n >> 1; (j ^= k) < k; k >>= 1);
        if(i < j) swap(X[i], X[j]);
    }
    for(len = 2; len <= n; len <= 1){
        len2 = len >> 1;
        wlen = (inv == -1) ? root_1 : root;
        for(i = len; i < root_pw; i <= 1){
            wlen = (lli)wlen * wlen % p;
        }
        for(i = 0; i < n; i += len){

```

```

            w = 1;
            for(j = 0; j < len2; ++j){
                u = X[i + j], v = (lli)X[i + j + len2] * w % p;
                X[i + j] = u + v < p ? u + v : u + v - p;
                X[i + j + len2] = u - v < 0 ? u - v + p : u - v;
                w = (lli)w * wlen % p;
            }
        }
    }
    if(inv == -1){
        int nrev = inverse(n, p);
        for(i = 0; i < n; ++i){
            X[i] = (lli)X[i] * nrev % p;
        }
    }
}

```

4.3.1. Otros valores para escoger la raíz y el módulo

Raíz n -ésima de la unidad (ω)	ω^{-1}	Tamaño máximo del arreglo (n)	Módulo p
15	30584	2^{14}	$4 \times 2^{14} + 1 = 65537$
9	7282	2^{15}	$2 \times 2^{15} + 1 = 65537$
3	21846	2^{16}	$1 \times 2^{16} + 1 = 65537$
8	688129	2^{17}	$6 \times 2^{17} + 1 = 786433$
5	471860	2^{18}	$3 \times 2^{18} + 1 = 786433$
12	3364182	2^{19}	$11 \times 2^{19} + 1 = 5767169$
5	4404020	2^{20}	$7 \times 2^{20} + 1 = 7340033$
38	21247462	2^{21}	$11 \times 2^{21} + 1 = 23068673$
21	49932191	2^{22}	$25 \times 2^{22} + 1 = 104857601$
4	125829121	2^{23}	$20 \times 2^{23} + 1 = 167772161$
31	128805723	2^{23}	$119 \times 2^{23} + 1 = 998244353$
2	83886081	2^{24}	$10 \times 2^{24} + 1 = 167772161$
17	29606852	2^{25}	$5 \times 2^{25} + 1 = 167772161$
30	15658735	2^{26}	$7 \times 2^{26} + 1 = 469762049$
137	749463956	2^{27}	$15 \times 2^{27} + 1 = 2013265921$

4.4. Aplicaciones

4.4.1. Multiplicación de polinomios

```
void multiplyPolynomials(vector<comp> & A, vector<comp> & B){
    int degree = A.size() + B.size() - 2;
    int size = nearestPowerOfTwo(degree + 1);
    A.resize(size);
    B.resize(size);
    fft(A, 1);
    fft(B, 1);
    for(int i = 0; i < size; i++){
        A[i] *= B[i];
    }
    fft(A, -1);
    A.resize(degree + 1);
}

void multiplyPolynomials(vector<int> & A, vector<int> & B){
    int degree = A.size() + B.size() - 2;
    int size = nearestPowerOfTwo(degree + 1);
    A.resize(size);
    B.resize(size);
    ntt(A, 1);
    ntt(B, 1);
    for(int i = 0; i < size; i++){
        A[i] = (1ll)iA[i] * B[i] % p;
    }
    ntt(A, -1);
    A.resize(degree + 1);
}
```

4.4.2. Multiplicación de números enteros grandes

```
string multiplyNumbers(const string & a, const string & b){
    int sgn = 1;
    int pos1 = 0, pos2 = 0;
    while(pos1 < a.size() && (a[pos1] < '1' || a[pos1] > '9')){
        if(a[pos1] == '-') sgn *= -1;
        ++pos1;
    }
```

```
    }
    while(pos2 < b.size() && (b[pos2] < '1' || b[pos2] > '9')){
        if(b[pos2] == '-') sgn *= -1;
        ++pos2;
    }
    vector<int> X(a.size() - pos1, Y(b.size() - pos2);
    if(X.empty() || Y.empty()) return "0";
    for(int i = pos1, j = X.size() - 1; i < a.size(); ++i){
        X[j--] = a[i] - '0';
    }
    for(int i = pos2, j = Y.size() - 1; i < b.size(); ++i){
        Y[j--] = b[i] - '0';
    }
    multiplyPolynomials(X, Y);
    stringstream ss;
    if(sgn == -1) ss << "-";
    int carry = 0;
    for(int i = 0; i < X.size(); ++i){
        X[i] += carry;
        carry = X[i] / 10;
        X[i] %= 10;
    }
    while(carry){
        X.push_back(carry % 10);
        carry /= 10;
    }
    for(int i = X.size() - 1; i >= 0; --i){
        ss << X[i];
    }
    return ss.str();
}
```

4.4.3. Inverso de un polinomio

```
vector<int> inversePolynomial(vector<int> & A){
    vector<int> R(1, inverse(A[0], p));
    while(R.size() < A.size()){
        int c = 2 * R.size();
        R.resize(c);
        vector<int> TR = R;
```

```

    TR.resize(nearestPowerOfTwo(2 * c));
    vector<int> TF(TR.size());
    for(int i = 0; i < c; ++i){
        TF[i] = A[i];
    }
    ntt(TR, 1);
    ntt(TF, 1);
    for(int i = 0; i < TR.size(); ++i){
        TR[i] = (lli)TR[i] * TR[i] % p * TF[i] % p;
    }
    ntt(TR, -1);
    TR.resize(2 * c);
    for(int i = 0; i < c; ++i){
        R[i] = R[i] + R[i] - TR[i];
        while(R[i] < 0) R[i] += p;
        while(R[i] >= p) R[i] -= p;
    }
}
R.resize(A.size());
return R;
}

        if(R[i] >= p) R[i] -= p;
        R[i] = (lli)R[i] * inv2 % p;
    }
}
R.resize(A.size());
return R;
}

```

4.4.4. Raíz cuadrada de un polinomio

```

const int inv2 = inverse(2, p);

vector<int> sqrtPolynomial(vector<int> & A){
    int r0 = 1; //r0^2 = A[0] mod p
    vector<int> R(1, r0);
    while(R.size() < A.size()){
        int c = 2 * R.size();
        R.resize(c);
        vector<int> TF(c);
        for(int i = 0; i < c; ++i){
            TF[i] = A[i];
        }
        vector<int> IR = inversePolynomial(R);
        multiplyPolynomials(TF, IR);
        for(int i = 0; i < c; ++i){
            R[i] = R[i] + TF[i];
        }
    }
}

```

5. Geometría

5.1. Estructura point

```
double eps = 1e-8;
# define M_PI 3.14159265358979323846
# define M_E 2.71828182845904523536

struct point{
    double x, y;

    point(): x(0), y(0){}

    point(double x, double y): x(x), y(y){}

    void read(){cin >> x >> y;}

    point operator+(const point & p) const{return point(x + p.x,
        ↪ y + p.y);}

    point operator-(const point & p) const{return point(x - p.x,
        ↪ y - p.y);}

    point operator*(const double & k) const{return point(x * k, y
        ↪ * k);}

    point operator/(const double & k) const{return point(x / k, y
        ↪ / k);}

    point operator+=(const point & p){*this = *this + p; return
        ↪ *this;}

    point operator-=(const point & p){*this = *this - p; return
        ↪ *this;}

    point operator*=(const double & p){*this = *this * p; return
        ↪ *this;}

    point operator/=(const double & p){*this = *this / p; return
        ↪ *this;}
```

```
point rotate(const double angle) const{
    return point(x * cos(angle) - y * sin(angle), x *
        ↪ sin(angle) + y * cos(angle));
}

point rotate(const double angle, const point & p){
    return p + ((*this) - p).rotate(angle);
}

point perpendicular() const{
    return point(-y, x);
}

double dot(const point & p) const{
    return x * p.x + y * p.y;
}

double length() const{
    return hypot(x, y);
}

double cross(const point & p) const{
    return x * p.y - y * p.x;
}

point normalize() const{
    return (*this) / length();
}

point projection(const point & p) const{
    return (*this) * p.dot(*this) / dot(*this);
}

point normal(const point & p) const{
    return p - projection(p);
}

bool operator==(const point & p) const{
    return abs(x - p.x) < eps && abs(y - p.y) < eps;
}

bool operator!=(const point & p) const{
    return !(*this == p);
}

bool operator<(const point & p) const{
    if(abs(x - p.x) < eps) return y < p.y;
    return x < p.x;
```

```

    }
    bool operator>(const point & p) const{
        if(abs(x - p.x) < eps) return y > p.y;
        return x > p.x;
    }
};

istream &operator>>(istream &is, point & P){
    point p;
    is >> p.x >> p.y;
    P = p;
    return is;
}

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

int sgn(double x){
    if(abs(x) < eps){
        return 0;
    }else if(x > 0){
        return 1;
    }else{
        return -1;
    }
}

```

5.2. Líneas y segmentos

5.2.1. Verificar si un punto pertenece a una línea o segmento

```

bool pointInLine(const point & a, const point & v, const point
↪ & p){
    //line a+tv, point p
    return abs((p - a).cross(v)) < eps;
}

bool pointInSegment(point a, point b, const point & p){
    //segment ab, point p

```

```

    if(a > b) swap(a, b);
    return pointInLine(a, b - a, p) && !(p < a || p > b);
}

```

5.2.2. Intersección de líneas

```

int intersectLinesInfo(const point & a1, const point & v1,
↪ const point & a2, const point & v2){
    //line a1+tv1
    //line a2+tv2
    double det = v1.cross(v2);
    if(abs(det) < eps){
        if(abs((a2 - a1).cross(v1)) < eps){
            return -1; //infinity points
        }else{
            return 0; //no points
        }
    }else{
        return 1; //single point
    }
}

point intersectLines(const point & a1, const point & v1, const
↪ point & a2, const point & v2){
    //lines a1+tv1, a2+tv2
    //assuming that they intersect
    double det = v1.cross(v2);
    return a1 + v1 * ((a2 - a1).cross(v2) / det);
}

```

5.2.3. Intersección línea-segmento

```

int intersectLineSegmentInfo(const point & a, const point & v,
↪ const point & c, const point & d){
    //line a+tv, segment cd
    point v2 = d - c;
    double det = v.cross(v2);
    if(abs(det) < eps){
        if(abs((c - a).cross(v)) < eps){

```

```

    return -1; //infinity points
}else{
    return 0; //no point
}
}else{
    return sgn(v.cross(c - a)) != sgn(v.cross(d - a)); //1:
    ↪ single point, 0: no point
}
}
}

```

5.2.4. Intersección de segmentos

```

int intersectSegmentsInfo(const point & a, const point & b,
    ↪ const point & c, const point & d){
    //segment ab, segment cd
    point v1 = b - a, v2 = d - c;
    int t = sgn(v1.cross(c - a)), u = sgn(v1.cross(d - a));
    if(t == u){
        if(t == 0){
            if(pointInSegment(a, b, c) || pointInSegment(a, b, d) ||
                ↪ pointInSegment(c, d, a) || pointInSegment(c, d, b)){
                return -1; //infinity points
            }else{
                return 0; //no point
            }
        }else{
            return 0; //no point
        }
    }else{
        return sgn(v2.cross(a - c)) != sgn(v2.cross(b - c)); //1:
        ↪ single point, 0: no point
    }
}
}

```

5.2.5. Distancia punto-recta

```

double distancePointLine(const point & a, const point & v,
    ↪ const point & p){
    //line: a + tv, point p

```

```

    return abs(v.cross(p - a)) / v.length();
}

```

5.3. Círculos

5.3.1. Distancia punto-círculo

```

double distancePointCircle(const point & p, const point & c,
    ↪ double r){
    //point p, center c, radius r
    return max(0.0, (p - c).length() - r);
}

```

5.3.2. Proyección punto exterior a círculo

```

point projectionPointCircle(const point & p, const point & c,
    ↪ double r){
    //point p (outside the circle), center c, radius r
    return c + (p - c) / (p - c).length() * r;
}

```

5.3.3. Puntos de tangencia de punto exterior

```

pair<point, point> pointsOfTangency(const point & p, const
    ↪ point & c, double r){
    //point p (outside the circle), center c, radius r
    point v = (p - c).normalize() * r;
    double theta = acos(r / (p - c).length());
    return {c + v.rotate(-theta), c + v.rotate(theta)};
}

```

5.3.4. Intersección línea-círculo

```

vector<point> intersectLineCircle(const point & a, const point
    ↪ & v, const point & c, double r){
    //line a+tv, center c, radius r
    double A = v.dot(v);

```

```

double B = (a - c).dot(v);
double C = (a - c).dot(a - c) - r * r;
double D = B*B - A*C;
if(abs(D) < eps) return {a + v * (-B/A)}; //line tangent to
↪ circle
else if(D < 0) return {}; //no intersection
else{ //two points of intersection (chord)
    D = sqrt(D);
    double t1 = (-B + D) / A;
    double t2 = (-B - D) / A;
    return {a + v * t1, a + v * t2};
}
}

```

5.3.5. Centro y radio a través de tres puntos

```

pair<point, double> getCircle(const point & m, const point & n,
↪ const point & p){
    //find circle that passes through points p, q, r
    point c = intersectLines((n + m) / 2, (n -
↪ m).perpendicular(), (p + n) / 2, (p -
↪ n).perpendicular());
    double r = (c - m).length();
    return {c, r};
}

```

5.3.6. Intersección de círculos

```

vector<point> intersectionCircles(const point & c1, double r1,
↪ const point & c2, double r2){
    //circle 1 with center c1 and radius r1
    //circle 2 with center c2 and radius r2
    double A = 2*r1*(c2.y - c1.y);
    double B = 2*r1*(c2.x - c1.x);
    double C = (c1 - c2).dot(c1 - c2) + r1*r1 - r2*r2;
    double D = A*A + B*B - C*C;
    if(abs(D) < eps) return {c1 + point(B, A) * r1 / C};
    else if(D < 0) return {};
    else{

```

```

        D = sqrt(D);
        double cos1 = (B*C + A*D) / (A*A + B*B);
        double sin1 = (A*C - B*D) / (A*A + B*B);
        double cos2 = (B*C - A*D) / (A*A + B*B);
        double sin2 = (A*C + B*D) / (A*A + B*B);
        return {c1 + point(cos1, sin1) * r1, c1 + point(cos2, sin2)
↪ * r1};
    }
}

```

5.3.7. Tangentes

```

int circleInsideCircle(const point & c1, double r1, const point
↪ & c2, double r2){
    //test if circle 2 is inside circle 1
    //returns "-1" if 2 touches internally 1, "1" if 2 is inside
↪ 1, "0" if they overlap
    double l = r1 - r2 - (c1 - c2).length();
    return (l > eps ? 1 : (abs(l) < eps ? -1 : 0));
}

```

```

int circleOutsideCircle(const point & c1, double r1, const
↪ point & c2, double r2){
    //test if circle 2 is outside circle 1
    //returns "-1" if they touch externally, "1" if 2 is outside
↪ 1, "0" if they overlap
    double l = (c1 - c2).length() - (r1 + r2);
    return (l > eps ? 1 : (abs(l) < eps ? -1 : 0));
}

```

```

vector<vector<point>> commonExteriorTangents(const point & c1,
↪ double r1, const point & c2, double r2){
    //returns a vector of segments or a single point
    if(r1 < r2) return commonExteriorTangents(c2, r2, c1, r1);
    if(c1 == c2 && abs(r1-r2) < 0) return {};
    int in = circleInsideCircle(c1, r1, c2, r2);
    if(in == 1) return {};
    else if(in == -1) return {{c1 + (c2 - c1).normalize() * r1}};
    else{
        pair<point, point> t;

```



```

    if(abs(r1-r2) < eps)
        t = {c1 - (c2 - c1).perpendicular(), c1 + (c2 -
            ↪ c1).perpendicular()};
    else
        t = pointsOfTangency(c2, c1, r1 - r2);
    t.first = (t.first - c1).normalize();
    t.second = (t.second - c1).normalize();
    return {{c1 + t.first * r1, c2 + t.first * r2}, {c1 +
        ↪ t.second * r1, c2 + t.second * r2}};
}
}

vector<vector<point>> commonInteriorTangents(const point & c1,
    ↪ double r1, const point & c2, double r2){
    if(c1 == c2 && abs(r1-r2) < 0) return {};
    int out = circleOutsideCircle(c1, r1, c2, r2);
    if(out == 0) return {};
    else if(out == -1) return {{c1 + (c2 - c1).normalize() *
        ↪ r1}};
    else{
        auto t = pointsOfTangency(c2, c1, r1 + r2);
        t.first = (t.first - c1).normalize();
        t.second = (t.second - c1).normalize();
        return {{c1 + t.first * r1, c2 - t.first * r2}, {c1 +
            ↪ t.second * r1, c2 - t.second * r2}};
    }
}

```

5.4. Polígonos

5.4.1. Perímetro y área de un polígono

```

double perimeter(vector<point> & P){
    int n = P.size();
    double ans = 0;
    for(int i = 0; i < n; i++){
        ans += (P[i] - P[(i + 1) % n]).length();
    }
    return ans;
}

```

```

double area(vector<point> & P){
    int n = P.size();
    double ans = 0;
    for(int i = 0; i < n; i++){
        ans += P[i].cross(P[(i + 1) % n]);
    }
    return abs(ans / 2);
}

```

5.4.2. Envolverte convexa (convex hull) de un polígono

```

vector<point> convexHull(vector<point> P){
    sort(P.begin(), P.end());
    vector<point> L, U;
    for(int i = 0; i < P.size(); i++){
        while(L.size() >= 2 && (L[L.size() - 2] -
            ↪ P[i]).cross(L[L.size() - 1] - P[i]) <= 0){
            L.pop_back();
        }
        L.push_back(P[i]);
    }
    for(int i = P.size() - 1; i >= 0; i--){
        while(U.size() >= 2 && (U[U.size() - 2] -
            ↪ P[i]).cross(U[U.size() - 1] - P[i]) <= 0){
            U.pop_back();
        }
        U.push_back(P[i]);
    }
    L.pop_back();
    U.pop_back();
    L.insert(L.end(), U.begin(), U.end());
    return L;
}

```

5.4.3. Verificar si un punto pertenece al perímetro de un polígono

```
bool pointInPerimeter(vector<point> & P, const point & p){
    int n = P.size();
    for(int i = 0; i < n; i++){
        if(pointInSegment(P[i], P[(i + 1) % n], p)){
            return true;
        }
    }
    return false;
}
```

5.4.4. Verificar si un punto pertenece a un polígono

```
int pointInPolygon(vector<point> & P, const point & p){
    if(pointInPerimeter(P, p)){
        return -1; //point in the perimeter
    }
    point bottomLeft = (*min_element(P.begin(), P.end())) -
        ↪ point(M_E, M_PI);
    int n = P.size();
    int rays = 0;
    for(int i = 0; i < n; i++){
        rays += (intersectSegmentsInfo(p, bottomLeft, P[i], P[(i +
        ↪ 1) % n]) == 1 ? 1 : 0);
    }
    return rays & 1; //0: point outside, 1: point inside
}
```

5.4.5. Centroide de un polígono

```
point centroid(vector<point> & P){
    point num;
    double den = 0;
    int n = P.size();
    for(int i = 0; i < n; ++i){
        double cross = P[i].cross(P[(i + 1) % n]);
        num += (P[i] + P[(i + 1) % n]) * cross;
    }
}
```

```
den += cross;
}
return num / (3.0 * den);
}
```

5.4.6. Pares de puntos antipodales

```
vector<pair<int, int>> antipodalPairs(vector<point> & P){
    vector<pair<int, int>> ans;
    int n = P.size(), k = 1;
    auto f = [&](int u, int v, int w){return
        ↪ abs((P[v%n]-P[u%n]).cross(P[w%n]-P[u%n]));};
    while(f(n-1, 0, k+1) > f(n-1, 0, k)) ++k;
    for(int i = 0, j = k; i <= k && j < n; ++i){
        ans.emplace_back(i, j);
        while(j < n-1 && f(i, i+1, j+1) > f(i, i+1, j))
            ans.emplace_back(i, ++j);
    }
    return ans;
}
```

5.4.7. Diámetro y ancho

```
pair<double, double> diameterAndWidth(vector<point> & P){
    int n = P.size(), k = 0;
    auto dot = [&](int a, int b){return
        ↪ (P[(a+1)%n]-P[a]).dot(P[(b+1)%n]-P[b]);};
    auto cross = [&](int a, int b){return
        ↪ (P[(a+1)%n]-P[a]).cross(P[(b+1)%n]-P[b]);};
    double diameter = numeric_limits<double>::min();
    double width = numeric_limits<double>::max();
    while(dot(0, k) > eps) k = (k+1) % n;
    for(int i = 0; i < n; ++i){
        while(cross(i, k) > eps) k = (k+1) % n;
        //pair: (i, k)
        diameter = max(diameter, (P[k] - P[i]).length());
        width = min(width, distancePointLine(P[i], P[(i+1)%n] -
        ↪ P[i], P[k]));
    }
}
```

```

    return make_pair(diameter, width);
}

```

5.4.8. Smallest enclosing rectangle

```

pair<double, double> smallestEnclosingRectangle(vector<point> &
↪ P){
    int n = P.size();
    auto dot = [&](int a, int b){return
↪ (P[(a+1)%n]-P[a]).dot(P[(b+1)%n]-P[b]);};
    auto cross = [&](int a, int b){return
↪ (P[(a+1)%n]-P[a]).cross(P[(b+1)%n]-P[b]);};
    double perimeter = numeric_limits<double>::max(), area =
↪ perimeter;
    for(int i = 0, j = 0, k = 0, m = 0; i < n; ++i){
        while(dot(i, j) > eps) j = (j+1) % n;
        if(!i) k = j;
        while(cross(i, k) > eps) k = (k+1) % n;
        if(!i) m = k;
        while(dot(i, m) < -eps) m = (m+1) % n;
        //pairs: (i, k) , (j, m)
        point v = P[(i+1)%n] - P[i];
        double h = distancePointLine(P[i], v, P[k]);
        double w = distancePointLine(P[j], v.perpendicular(),
↪ P[m]);
        perimeter = min(perimeter, 2 * (h + w));
        area = min(area, h * w);
    }
    return make_pair(area, perimeter);
}

```

5.5. Par de puntos más cercanos

```

bool comp1(const point & a, const point & b){
    return a.y < b.y;
}
pair<point, point> closestPairOfPoints(vector<point> P){
    sort(P.begin(), P.end(), comp1);
    set<point> S;

```

```

    double ans = 1e9;
    point p, q;
    int pos = 0;
    for(int i = 0; i < P.size(); ++i){
        while(pos < i && abs(P[i].y - P[pos].y) >= ans){
            S.erase(P[pos++]);
        }
        auto lower = S.lower_bound({P[i].x - ans - eps, -1e9});
        auto upper = S.upper_bound({P[i].x + ans + eps, -1e9});
        for(auto it = lower; it != upper; ++it){
            double d = (P[i] - *it).length();
            if(d < ans){
                ans = d;
                p = P[i];
                q = *it;
            }
        }
        S.insert(P[i]);
    }
    return {p, q};
}

```

5.6. Vantage Point Tree (puntos más cercanos a cada punto)

```

struct vantage_point_tree{
    struct node
    {
        point p;
        double th;
        node *l, *r;
    }*root;

    vector<pair<double, point>> aux;

    vantage_point_tree(vector<point> &ps){
        for(int i = 0; i < ps.size(); ++i)
            aux.push_back({ 0, ps[i] });
        root = build(0, ps.size());
    }

```

```

node *build(int l, int r){
    if(l == r)
        return 0;
    swap(aux[l], aux[l + rand() % (r - l)]);
    point p = aux[l++].second;
    if(l == r)
        return new node({ p });
    for(int i = l; i < r; ++i)
        aux[i].first = (p - aux[i].second).length() * (p -
            ↪ aux[i].second).length();
    int m = (l + r) / 2;
    nth_element(aux.begin() + l, aux.begin() + m, aux.begin() +
        ↪ r);
    return new node({ p, sqrt(aux[m].first), build(l, m),
        ↪ build(m, r) });
}

priority_queue<pair<double, node*>> que;

void k_nn(node *t, point p, int k){
    if(!t)
        return;
    double d = (p - t->p).length();
    if(que.size() < k)
        que.push({ d, t });
    else if(que.top().first > d){
        que.pop();
        que.push({ d, t });
    }
    if(!t->l && !t->r)
        return;
    if(d < t->th){
        k_nn(t->l, p, k);
        if(t->th - d <= que.top().first)
            k_nn(t->r, p, k);
    }else{
        k_nn(t->r, p, k);
        if(d - t->th <= que.top().first)
            k_nn(t->l, p, k);
    }
}

```

```

}

vector<point> k_nn(point p, int k){
    k_nn(root, p, k);
    vector<point> ans;
    for(; !que.empty(); que.pop())
        ans.push_back(que.top().second->p);
    reverse(ans.begin(), ans.end());
    return ans;
}
};

```

5.7. Suma Minkowski

```

vector<point> minkowskiSum(vector<point> A, vector<point> B){
    int na = (int)A.size(), nb = (int)B.size();
    if(A.empty() || B.empty()) return {};

    rotate(A.begin(), min_element(A.begin(), A.end()), A.end());
    rotate(B.begin(), min_element(B.begin(), B.end()), B.end());

    int pa = 0, pb = 0;
    vector<point> M;

    while(pa < na && pb < nb){
        M.push_back(A[pa] + B[pb]);
        double x = (A[(pa + 1) % na] - A[pa]).cross(B[(pb + 1) %
            ↪ nb] - B[pb]);
        if(x <= eps) pb++;
        if(-eps <= x) pa++;
    }

    while(pa < na) M.push_back(A[pa++] + B[0]);
    while(pb < nb) M.push_back(B[pb++] + A[0]);

    return M;
}

```

6. Grafos

6.1. Estructura disjointSet

```
struct disjointSet{
    int N;
    vector<short int> rank;
    vi parent, count;

    disjointSet(int N): N(N), parent(N), count(N), rank(N){}

    void makeSet(int v){
        count[v] = 1;
        parent[v] = v;
    }

    int findSet(int v){
        if(v == parent[v]) return v;
        return parent[v] = findSet(parent[v]);
    }

    void unionSet(int a, int b){
        a = findSet(a), b = findSet(b);
        if(a == b) return;
        if(rank[a] < rank[b]){
            parent[a] = b;
            count[b] += count[a];
        }else{
            parent[b] = a;
            count[a] += count[b];
            if(rank[a] == rank[b]) ++rank[a];
        }
    }
};
```

6.2. Estructura edge

```
struct edge{
    int source, dest, cost;
```

```
edge(): source(0), dest(0), cost(0){}
```

```
edge(int dest, int cost): dest(dest), cost(cost){}
```

```
edge(int source, int dest, int cost): source(source),
    ↪ dest(dest), cost(cost){}
```

```
bool operator==(const edge & b) const{
    return source == b.source && dest == b.dest && cost ==
        ↪ b.cost;
}
bool operator<(const edge & b) const{
    return cost < b.cost;
}
bool operator>(const edge & b) const{
    return cost > b.cost;
}
};
```

6.3. Estructura path

```
struct path{
    int cost = inf;
    vi vertices;
    int size = 1;
    int previous = -1;
};
```

6.4. Estructura graph

```
struct graph{
    vector<vector<edge>> adjList;
    vector<vb> adjMatrix;
    vector<vi> costMatrix;
    vector<edge> edges;
    int V = 0;
    bool dir = false;

    graph(int n, bool dir): V(n), dir(dir), adjList(n), edges(n),
        ↪ adjMatrix(n, vb(n)), costMatrix(n, vi(n)){
```

```

for(int i = 0; i < n; ++i)
    for(int j = 0; j < n; ++j)
        costMatrix[i][j] = (i == j ? 0 : inf);
}

void add(int source, int dest, int cost){
    adjList[source].emplace_back(source, dest, cost);
    edges.emplace_back(source, dest, cost);
    adjMatrix[source][dest] = true;
    costMatrix[source][dest] = cost;
    if(!dir){
        adjList[dest].emplace_back(dest, source, cost);
        adjMatrix[dest][source] = true;
        costMatrix[dest][source] = cost;
    }
}

void buildPaths(vector<path> & paths){
    for(int i = 0; i < V; i++){
        int actual = i;
        for(int j = 0; j < paths[i].size; j++){
            paths[i].vertices.push_back(actual);
            actual = paths[actual].previous;
        }
        reverse(paths[i].vertices.begin(),
            ↪ paths[i].vertices.end());
    }
}

```

6.5. DFS genérica

```

void dfs(int u, vi & status, vi & parent){
    status[u] = 1;
    for(edge & current : adjList[u]){
        int v = current.dest;
        if(status[v] == 0){ //not visited
            parent[v] = u;
            dfs(v, status, parent);
        }else if(status[v] == 1){ //explored
            if(v == parent[u]){

```

```

        //bidirectional node u<-->v
    }else{
        //back edge u-v
    }
    }else if(status[v] == 2){ //visited
        //forward edge u-v
    }
    }
    status[u] = 2;
}

```

6.6. Dijkstra con reconstrucción del camino más corto con menos vértices

```

vector<path> dijkstra(int start){
    priority_queue<edge, vector<edge>, greater<edge>> cola;
    vector<path> paths(V, path());
    vb relaxed(V);
    cola.push(edge(start, 0));
    paths[start].cost = 0;
    while(!cola.empty()){
        int u = cola.top().dest; cola.pop();
        relaxed[u] = true;
        for(edge & current : adjList[u]){
            int v = current.dest;
            if(relaxed[v]) continue;
            int nuevo = paths[u].cost + current.cost;
            if(nuevo == paths[v].cost && paths[u].size + 1 <
                ↪ paths[v].size){
                paths[v].previous = u;
                paths[v].size = paths[u].size + 1;
            }else if(nuevo < paths[v].cost){
                paths[v].previous = u;
                paths[v].size = paths[u].size + 1;
                cola.push(edge(v, nuevo));
                paths[v].cost = nuevo;
            }
        }
    }
    buildPaths(paths);
}

```

```

    return paths;
}

```

6.7. Bellman Ford con reconstrucción del camino más corto con menos vértices

```

vector<path> bellmanFord(int start){
    vector<path> paths(V, path());
    vi processed(V);
    vb inQueue(V);
    queue<int> Q;
    paths[start].cost = 0;
    Q.push(start);
    while(!Q.empty()){
        int u = Q.front(); Q.pop(); inQueue[u] = false;
        if(paths[u].cost == inf) continue;
        ++processed[u];
        if(processed[u] == V){
            cout << "Negative cycle\n";
            return {};
        }
        for(edge & current : adjList[u]){
            int v = current.dest;
            int nuevo = paths[u].cost + current.cost;
            if(nuevo == paths[v].cost && paths[u].size + 1 <
               ↪ paths[v].size){
                paths[v].previous = u;
                paths[v].size = paths[u].size + 1;
            }else if(nuevo < paths[v].cost){
                if(!inQueue[v]){
                    Q.push(v);
                    inQueue[v] = true;
                }
                paths[v].previous = u;
                paths[v].size = paths[u].size + 1;
                paths[v].cost = nuevo;
            }
        }
    }
    buildPaths(paths);
}

```

```

    return paths;
}

```

6.8. Floyd

```

vector<vi> floyd(){
    vector<vi> tmp = costMatrix;
    for(int k = 0; k < V; ++k)
        for(int i = 0; i < V; ++i)
            for(int j = 0; j < V; ++j)
                if(tmp[i][k] != inf && tmp[k][j] != inf)
                    tmp[i][j] = min(tmp[i][j], tmp[i][k] + tmp[k][j]);
    return tmp;
}

```

6.9. Cerradura transitiva $O(V^3)$

```

vector<vb> transitiveClosure(){
    vector<vb> tmp = adjMatrix;
    for(int k = 0; k < V; ++k)
        for(int i = 0; i < V; ++i)
            for(int j = 0; j < V; ++j)
                tmp[i][j] = tmp[i][j] || (tmp[i][k] && tmp[k][j]);
    return tmp;
}

```

6.10. Cerradura transitiva $O(V^2)$

```

vector<vb> transitiveClosureDFS(){
    vector<vb> tmp(V, vb(V));
    function<void(int, int)> dfs = [&](int start, int u){
        for(edge & current : adjList[u]){
            int v = current.dest;
            if(!tmp[start][v]){
                tmp[start][v] = true;
                dfs(start, v);
            }
        }
    }
}

```

```
};
for(int u = 0; u < V; u++)
    dfs(u, u);
return tmp;
}
```

6.11. Verificar si el grafo es bipartito

```
bool isBipartite(){
    vi side(V, -1);
    queue<int> q;
    for (int st = 0; st < V; ++st){
        if(side[st] != -1) continue;
        q.push(st);
        side[st] = 0;
        while(!q.empty()){
            int u = q.front();
            q.pop();
            for (edge & current : adjList[u]){
                int v = current.dest;
                if(side[v] == -1) {
                    side[v] = side[u] ^ 1;
                    q.push(v);
                }else{
                    if(side[v] == side[u]) return false;
                }
            }
        }
    }
    return true;
}
```

6.12. Orden topológico

```
vi topologicalSort(){
    int visited = 0;
    vi order, indegree(V);
    for(auto & node : adjList){
        for(edge & current : node){
```

```
            int v = current.dest;
            ++indegree[v];
        }
    }
    queue<int> Q;
    for(int i = 0; i < V; ++i){
        if(indegree[i] == 0) Q.push(i);
    }
    while(!Q.empty()){
        int source = Q.front();
        Q.pop();
        order.push_back(source);
        ++visited;
        for(edge & current : adjList[source]){
            int v = current.dest;
            --indegree[v];
            if(indegree[v] == 0) Q.push(v);
        }
    }
    if(visited == V) return order;
    else return {};
}
```

6.13. Detectar ciclos

```
bool hasCycle(){
    vi color(V);
    function<bool(int, int)> dfs = [&](int u, int parent){
        color[u] = 1;
        bool ans = false;
        int ret = 0;
        for(edge & current : adjList[u]){
            int v = current.dest;
            if(color[v] == 0)
                ans |= dfs(v, u);
            else if(color[v] == 1 && (dir || v != parent || ret++))
                ans = true;
        }
        color[u] = 2;
        return ans;
    }
```



```

};
for(int u = 0; u < V; ++u)
    if(color[u] == 0 && dfs(u, -1))
        return true;
return false;
}

```

6.14. Puentes y puntos de articulación

```

pair<vb, vector<edge>> articulationBridges(){
    vi low(V), label(V);
    vb points(V);
    vector<edge> bridges;
    int time = 0;
    function<int(int, int)> dfs = [&](int u, int p){
        label[u] = low[u] = ++time;
        int hijos = 0, ret = 0;
        for(edge & current : adjList[u]){
            int v = current.dest;
            if(v == p && !ret++) continue;
            if(!label[v]){
                ++hijos;
                dfs(v, u);
                if(label[u] <= low[v])
                    points[u] = true;
                if(label[u] < low[v])
                    bridges.push_back(current);
                low[u] = min(low[u], low[v]);
            }
            low[u] = min(low[u], label[v]);
        }
        return hijos;
    };
    for(int u = 0; u < V; ++u)
        if(!label[u])
            points[u] = dfs(u, -1) > 1;
    return make_pair(points, bridges);
}

```

6.15. Componentes fuertemente conexas

```

vector<vi> scc(){
    vi low(V), label(V);
    int time = 0;
    vector<vi> ans;
    stack<int> S;
    function<void(int)> dfs = [&](int u){
        label[u] = low[u] = ++time;
        S.push(u);
        for(edge & current : adjList[u]){
            int v = current.dest;
            if(!label[v]) dfs(v);
            low[u] = min(low[u], low[v]);
        }
        if(label[u] == low[u]){
            vi comp;
            while(S.top() != u){
                comp.push_back(S.top());
                low[S.top()] = V + 1;
                S.pop();
            }
            comp.push_back(S.top());
            S.pop();
            ans.push_back(comp);
            low[u] = V + 1;
        }
    };
    for(int u = 0; u < V; ++u)
        if(!label[u]) dfs(u);
    return ans;
}

```

6.16. Árbol mínimo de expansión (Kruskal)

```

vector<edge> kruskal(){
    sort(edges.begin(), edges.end());
    vector<edge> MST;
    disjointSet DS(V);
    for(int u = 0; u < V; ++u)

```

```

    DS.makeSet(u);
    int i = 0;
    while(i < edges.size() && MST.size() < V - 1){
        edge current = edges[i++];
        int u = current.source, v = current.dest;
        if(DS.findSet(u) != DS.findSet(v)){
            MST.push_back(current);
            DS.unionSet(u, v);
        }
    }
    return MST;
}

```

6.17. Máximo emparejamiento bipartito

```

bool tryKuhn(int u, vb & used, vi & left, vi & right){
    if(used[u]) return false;
    used[u] = true;
    for(edge & current : adjList[u]){
        int v = current.dest;
        if(right[v] == -1 || tryKuhn(right[v], used, left,
            ↪ right)){
            right[v] = u;
            left[u] = v;
            return true;
        }
    }
    return false;
}

bool augmentingPath(int u, vb & used, vi & left, vi & right){
    used[u] = true;
    for(edge & current : adjList[u]){
        int v = current.dest;
        if(right[v] == -1){
            right[v] = u;
            left[u] = v;
            return true;
        }
    }
}

```

```

for(edge & current : adjList[u]){
    int v = current.dest;
    if(!used[right[v]] && augmentingPath(right[v], used,
        ↪ left, right)){
        right[v] = u;
        left[u] = v;
        return true;
    }
}
return false;
}

```

*//vertices from the left side numbered from 0 to l-1
 //vertices from the right side numbered from 0 to r-1
 //graph[u] represents the left side
 //graph[u][v] represents the right side
 //we can use tryKuhn() or augmentingPath()*

```

vector<pair<int, int>> maxMatching(int l, int r){
    vi left(l, -1), right(r, -1);
    vb used(l);
    for(int u = 0; u < l; ++u){
        tryKuhn(u, used, left, right);
        fill(used.begin(), used.end(), false);
    }
    vector<pair<int, int>> ans;
    for(int u = 0; u < r; ++u){
        if(right[u] != -1){
            ans.emplace_back(right[u], u);
        }
    }
    return ans;
}

```

6.18. Circuito euleriano

7. Árboles

7.1. Estructura tree

```

struct tree{
    vi parent, level, weight;
    vector<vi> dists, DP;
    int n, root;

    void dfs(int u, graph & G){
        for(edge & curr : G.adjList[u]){
            int v = curr.dest;
            int w = curr.cost;
            if(v != parent[u]){
                parent[v] = u;
                weight[v] = w;
                level[v] = level[u] + 1;
                dfs(v, G);
            }
        }
    }

    tree(int n, int root): n(n), root(root), parent(n), level(n),
        ↪ weight(n), dists(n, vi(20)), DP(n, vi(20)){
        parent[root] = root;
    }

    tree(graph & G, int root): n(G.V), root(root), parent(G.V),
        ↪ level(G.V), weight(G.V), dists(G.V, vi(20)), DP(G.V,
        ↪ vi(20)){
        parent[root] = root;
        dfs(root, G);
    }

    void pre(){
        for(int u = 0; u < n; u++){
            DP[u][0] = parent[u];
            dists[u][0] = weight[u];
        }
        for(int i = 1; (1 << i) <= n; ++i){

```

```

            for(int u = 0; u < n; ++u){
                DP[u][i] = DP[DP[u][i - 1]][i - 1];
                dists[u][i] = dists[u][i - 1] + dists[DP[u][i - 1]][i - 1];
            }
        }
    }
}

```

7.2. k -ésimo ancestro

```

int ancestor(int p, int k){
    int h = level[p] - k;
    if(h < 0) return -1;
    int lg;
    for(lg = 1; (1 << lg) <= level[p]; ++lg);
    lg--;
    for(int i = lg; i >= 0; --i){
        if(level[p] - (1 << i) >= h){
            p = DP[p][i];
        }
    }
    return p;
}

```

7.3. LCA

```

int lca(int p, int q){
    if(level[p] < level[q]) swap(p, q);
    int lg;
    for(lg = 1; (1 << lg) <= level[p]; ++lg);
    lg--;
    for(int i = lg; i >= 0; --i){
        if(level[p] - (1 << i) >= level[q]){
            p = DP[p][i];
        }
    }
    if(p == q) return p;

    for(int i = lg; i >= 0; --i){

```

```

    if(DP[p][i] != -1 && DP[p][i] != DP[q][i]){
        p = DP[p][i];
        q = DP[q][i];
    }
}
return parent[p];
}

```

7.4. Distancia entre dos nodos

```

int dist(int p, int q){
    if(level[p] < level[q]) swap(p, q);
    int lg;
    for(lg = 1; (1 << lg) <= level[p]; ++lg);
    lg--;
    int sum = 0;
    for(int i = lg; i >= 0; --i){
        if(level[p] - (1 << i) >= level[q]){
            sum += dists[p][i];
            p = DP[p][i];
        }
    }
    if(p == q) return sum;

    for(int i = lg; i >= 0; --i){
        if(DP[p][i] != -1 && DP[p][i] != DP[q][i]){
            sum += dists[p][i] + dists[q][i];
            p = DP[p][i];
            q = DP[q][i];
        }
    }
    sum += dists[p][0] + dists[q][0];
    return sum;
}

```

7.5. HLD

7.6. Link Cut

8. Flujos

8.1. Estructura flowEdge

```

template<typename T>
struct flowEdge{
    int dest;
    T flow, capacity, cost;
    flowEdge *res;

    flowEdge(): dest(0), flow(0), capacity(0), cost(0),
        ↪ res(NULL){}
    flowEdge(int dest, T flow, T capacity, T cost = 0):
        ↪ dest(dest), flow(flow), capacity(capacity), cost(cost),
        ↪ res(NULL){}

    void addFlow(T flow){
        this->flow += flow;
        this->res->flow -= flow;
    }
};

```

8.2. Estructura flowGraph

```

template<typename T>
struct flowGraph{
    T inf = numeric_limits<T>::max();
    vector<vector<flowEdge<T>*>> adjList;
    vector<int> dist, pos;
    int V;
    flowGraph(int V): V(V), adjList(V), dist(V), pos(V){}
    ~flowGraph(){
        for(int i = 0; i < V; ++i)
            for(int j = 0; j < adjList[i].size(); ++j)
                delete adjList[i][j];
    }
    void addEdge(int u, int v, T capacity, T cost = 0){
        flowEdge<T> *uv = new flowEdge<T>(v, 0, capacity, cost);
        flowEdge<T> *vu = new flowEdge<T>(u, capacity, capacity,
            ↪ -cost);
    }
};

```

```

uv->res = vu;
vu->res = uv;
adjList[u].push_back(uv);
adjList[v].push_back(vu);
}

```

8.3. Algoritmo de Edmonds-Karp $O(VE^2)$

```

//Maximun Flow using Edmonds-Karp Algorithm  $O(VE^2)$ 
T edmondsKarp(int s, int t){
    T maxFlow = 0;
    vector<flowEdge<T>*> parent(V);
    while(true){
        fill(parent.begin(), parent.end(), nullptr);
        queue<int> Q;
        Q.push(s);
        while(!Q.empty() && !parent[t]){
            int u = Q.front(); Q.pop();
            for(flowEdge<T> *v : adjList[u]){
                if(!parent[v->dest] && v->capacity > v->flow){
                    parent[v->dest] = v;
                    Q.push(v->dest);
                }
            }
        }
        if(!parent[t]) break;
        T f = inf;
        for(int u = t; u != s; u = parent[u]->res->dest)
            f = min(f, parent[u]->capacity - parent[u]->flow);
        for(int u = t; u != s; u = parent[u]->res->dest)
            parent[u]->addFlow(f);
        maxFlow += f;
    }
    return maxFlow;
}

```

8.4. Algoritmo de Dinic $O(V^2E)$

```

//Maximun Flow using Dinic Algorithm  $O(EV^2)$ 
T blockingFlow(int u, int t, T flow){
    if(u == t) return flow;
    for(int &i = pos[u]; i < adjList[u].size(); ++i){
        flowEdge<T> *v = adjList[u][i];
        if(v->capacity > v->flow && dist[u] + 1 ==
            ⇨ dist[v->dest]){
            T fv = blockingFlow(v->dest, t, min(flow, v->capacity -
                ⇨ v->flow));
            if(fv > 0){
                v->addFlow(fv);
                return fv;
            }
        }
    }
    return 0;
}

T dinic(int s, int t){
    T maxFlow = 0;
    dist[t] = 0;
    while(dist[t] != -1){
        fill(dist.begin(), dist.end(), -1);
        queue<int> Q;
        Q.push(s);
        dist[s] = 0;
        while(!Q.empty()){
            int u = Q.front(); Q.pop();
            for(flowEdge<T> *v : adjList[u]){
                if(dist[v->dest] == -1 && v->flow != v->capacity){
                    dist[v->dest] = dist[u] + 1;
                    Q.push(v->dest);
                }
            }
        }
    }
    if(dist[t] != -1){
        T f;
        fill(pos.begin(), pos.end(), 0);
        while(f = blockingFlow(s, t, inf))
            maxFlow += f;
    }
}

```

```

    }
}
return maxFlow;
}

```

```

    parent[u]->addFlow(cap[t]);
}
return {maxFlow, minCost};
}

```

8.5. Flujo máximo de costo mínimo

```

//Max Flow Min Cost
pair<T, T> maxFlowMinCost(int s, int t){
    vector<bool> inQueue(V);
    vector<T> distance(V), cap(V);
    vector<flowEdge<T>*> parent(V);
    T maxFlow = 0, minCost = 0;
    while(true){
        fill(distance.begin(), distance.end(), inf);
        fill(parent.begin(), parent.end(), nullptr);
        fill(cap.begin(), cap.end(), 0);
        distance[s] = 0;
        cap[s] = inf;
        queue<int> Q;
        Q.push(s);
        while(!Q.empty()){
            int u = Q.front(); Q.pop(); inQueue[u] = 0;
            for(flowEdge<T> *v : adjList[u]){
                if(v->capacity > v->flow && distance[v->dest] >
                    distance[u] + v->cost){
                    distance[v->dest] = distance[u] + v->cost;
                    parent[v->dest] = v;
                    cap[v->dest] = min(cap[u], v->capacity - v->flow);
                    if(!inQueue[v->dest]){
                        Q.push(v->dest);
                        inQueue[v->dest] = true;
                    }
                }
            }
        }
        if(!parent[t]) break;
        maxFlow += cap[t];
        minCost += cap[t] * distance[t];
        for(int u = t; u != s; u = parent[u]->res->dest)

```

9. Estructuras de datos

9.1. Segment Tree

9.1.1. Point updates, range queries

```
template<typename T>
struct SegmentTree{
    int N;
    vector<T> ST;

    SegmentTree(int N): N(N){
        ST.assign(N << 1, 0);
    }

    //build from an array in O(n)
    void build(vector<T> & arr){
        for(int i = 0; i < N; ++i)
            ST[N + i] = arr[i];
        for(int i = N - 1; i > 0; --i)
            ST[i] = ST[i << 1] + ST[i << 1 | 1];
    }

    //single element update in i
    void update(int i, T value){
        ST[i += N] = value; //update the element accordingly
        while(i >>= 1)
            ST[i] = ST[i << 1] + ST[i << 1 | 1];
    }

    //range query, [l, r]
    T query(int l, int r){
        T res = 0;
        for(l += N, r += N; l <= r; l >>= 1, r >>= 1){
            if(l & 1) res += ST[l++];
            if(!(r & 1)) res += ST[r--];
        }
        return res;
    }
};
```

9.1.2. Dinamic with lazy propagation

```
template<typename T>
struct SegmentTreeDin{
    SegmentTreeDin *left, *right;
    int l, r;
    T value, lazy;

    SegmentTreeDin(int start, int end, vector<T> & arr):
        ↪ left(NULL), right(NULL), l(start), r(end), value(0),
        ↪ lazy(0){
        if(l == r) value = arr[l];
        else{
            int half = l + ((r - l) >> 1);
            left = new SegmentTreeDin(l, half, arr);
            right = new SegmentTreeDin(half+1, r, arr);
            value = left->value + right->value;
        }
    }

    void propagate(T dif){
        value += (r - l + 1) * dif;
        if(l != r){
            left->lazy += dif;
            right->lazy += dif;
        }
    }

    T query(int start, int end){
        if(lazy != 0){
            propagate(lazy);
            lazy = 0;
        }
        if(end < l || r < start) return 0;
        if(start <= l && r <= end) return value;
        else return left->query(start, end) + right->query(start,
        ↪ end);
    }

    void update(int start, int end, T dif){
        if(lazy != 0){
```

```

    propagate(lazy);
    lazy = 0;
}
if(end < l || r < start) return;
if(start <= l && r <= end) propagate(dif);
else{
    left->update(start, end, dif);
    right->update(start, end, dif);
    value = left->value + right->value;
}
}
}

void update(int i, T value){
    update(i, i, value);
}
};

```

9.2. Fenwick Tree

```

template<typename T>
struct FenwickTree{
    int N;
    vector<T> bit;

    FenwickTree(int N): N(N){
        bit.assign(N, 0);
    }

    void build(vector<T> & arr){
        for(int i = 0; i < arr.size(); ++i){
            update(i, arr[i]);
        }
    }

    //single element increment
    void update(int pos, T value){
        while(pos < N){
            bit[pos] += value;
            pos |= pos + 1;
        }
    }
}

```

```

}

//range query, [0, r]
T query(int r){
    T res = 0;
    while(r >= 0){
        res += bit[r];
        r = (r & (r + 1)) - 1;
    }
    return res;
}

//range query, [l, r]
T query(int l, int r){
    return query(r) - query(l - 1);
}
};

```

9.3. SQRT Decomposition

```

struct MQuery{
    int l, r, index, S;
    bool operator<(const MQuery & q) const{
        int c_o = l / S, c_q = q.l / S;
        if(c_o == c_q)
            return r < q.r;
        return c_o < c_q;
    }
};

template<typename T>
struct SQRT{
    int N, S;
    vector<T> A, B;

    SQRT(int N): N(N){
        this->S = sqrt(N + .0) + 1;
        A.assign(N, 0);
        B.assign(S, 0);
    }
}

```



```

void build(vector<T> & arr){
    A = vector<int>(arr.begin(), arr.end());
    for(int i = 0; i < N; ++i) B[i / S] += A[i];
}

//single element update
void update(int pos, T value){
    int k = pos / S;
    A[pos] = value;
    T res = 0;
    for(int i = k * S, end = min(N, (k + 1) * S) - 1; i <= end;
        ↪ ++i) res += A[i];
    B[k] = res;
}

//range query, [l, r]
T query(int l, int r){
    T res = 0;
    int c_l = l / S, c_r = r / S;
    if(c_l == c_r){
        for(int i = l; i <= r; ++i) res += A[i];
    }else{
        for(int i = l, end = (c_l + 1) * S - 1; i <= end; ++i)
            ↪ res += A[i];
        for(int i = c_l + 1; i <= c_r - 1; ++i) res += B[i];
        for(int i = c_r * S; i <= r; ++i) res += A[i];
    }
    return res;
}

//range queries offline using MO's algorithm
vector<T> MO(vector<MOQuery> & queries){
    vector<T> ans(queries.size());
    sort(queries.begin(), queries.end());
    T current = 0;
    int prevL = 0, prevR = -1;
    int i, j;
    for(const MOQuery & q : queries){
        for(i = prevL, j = min(prevR, q.l - 1); i <= j; ++i){
            //remove from the left

```

```

            current -= A[i];
        }
        for(i = prevL - 1; i >= q.l; --i){
            //add to the left
            current += A[i];
        }
        for(i = max(prevR + 1, q.l); i <= q.r; ++i){
            //add to the right
            current += A[i];
        }
        for(i = prevR; i >= q.r + 1; --i){
            //remove from the right
            current -= A[i];
        }
        prevL = q.l, prevR = q.r;
        ans[q.index] = current;
    }
    return ans;
}
};

```

9.4. AVL Tree

```

template<typename T>
struct AVLNode
{
    AVLNode<T> *left, *right;
    short int height;
    int size;
    T value;

    AVLNode(T value = 0): left(NULL), right(NULL), value(value),
        ↪ height(1), size(1){}

    inline short int balance(){
        return (right ? right->height : 0) - (left ? left->height :
            ↪ 0);
    }

    inline void update(){

```

```

    height = 1 + max(left ? left->height : 0, right ?
    ↪ right->height : 0);
    size = 1 + (left ? left->size : 0) + (right ? right->size :
    ↪ 0);
}

AVLNode *maxLeftChild(){
    AVLNode *ret = this;
    while(ret->left) ret = ret->left;
    return ret;
}
};

template<typename T>
struct AVLTree
{
    AVLNode<T> *root;

    AVLTree(): root(NULL){}

    inline int nodeSize(AVLNode<T> *& pos){return pos ?
    ↪ pos->size: 0;}

    int size(){return nodeSize(root);}

    void leftRotate(AVLNode<T> *& x){
        AVLNode<T> *y = x->right, *t = y->left;
        y->left = x, x->right = t;
        x->update(), y->update();
        x = y;
    }

    void rightRotate(AVLNode<T> *& y){
        AVLNode<T> *x = y->left, *t = x->right;
        x->right = y, y->left = t;
        y->update(), x->update();
        y = x;
    }

    void updateBalance(AVLNode<T> *& pos){
        short int bal = pos->balance();

        if(bal > 1){
            if(pos->right->balance() < 0) rightRotate(pos->right);
            leftRotate(pos);
        }else if(bal < -1){
            if(pos->left->balance() > 0) leftRotate(pos->left);
            rightRotate(pos);
        }
    }

    void insert(AVLNode<T> *&pos, T & value){
        if(pos){
            value < pos->value ? insert(pos->left, value) :
            ↪ insert(pos->right, value);
            pos->update(), updateBalance(pos);
        }else{
            pos = new AVLNode<T>(value);
        }
    }

    AVLNode<T> *search(T & value){
        AVLNode<T> *pos = root;
        while(pos){
            if(value == pos->value) break;
            pos = (value < pos->value ? pos->left : pos->right);
        }
        return pos;
    }

    void erase(AVLNode<T> *&pos, T & value){
        if(!pos) return;
        if(value < pos->value) erase(pos->left, value);
        else if(value > pos->value) erase(pos->right, value);
        else{
            if(!pos->left) pos = pos->right;
            else if(!pos->right) pos = pos->left;
            else{
                pos->value = pos->right->maxLeftChild()->value;
                erase(pos->right, pos->value);
            }
        }
        if(pos) pos->update(), updateBalance(pos);
    }
};

```

```

}

void insert(T value){insert(root, value);}

void erase(T value){erase(root, value);}

void updateVal(T old, T New){
    if(search(old))
        erase(old), insert(New);
}

T kth(int i){
    if(i < 0 || i >= nodeSize(root)) return -1;
    AVLNode<T> *pos = root;
    while(i != nodeSize(pos->left)){
        if(i < nodeSize(pos->left)){
            pos = pos->left;
        }else{
            i -= nodeSize(pos->left) + 1;
            pos = pos->right;
        }
    }
    return pos->value;
}

int lessThan(T & x){
    int ans = 0;
    AVLNode<T> *pos = root;
    while(pos){
        if(x > pos->value){
            ans += nodeSize(pos->left) + 1;
            pos = pos->right;
        }else{
            pos = pos->left;
        }
    }
    return ans;
}

int lessThanOrEqual(T & x){
    int ans = 0;

```

```

    AVLNode<T> *pos = root;
    while(pos){
        if(x < pos->value){
            pos = pos->left;
        }else{
            ans += nodeSize(pos->left) + 1;
            pos = pos->right;
        }
    }
    return ans;
}

int greaterThan(T & x){
    int ans = 0;
    AVLNode<T> *pos = root;
    while(pos){
        if(x < pos->value){
            ans += nodeSize(pos->right) + 1;
            pos = pos->left;
        }else{
            pos = pos->right;
        }
    }
    return ans;
}

int greaterThanOrEqual(T & x){
    int ans = 0;
    AVLNode<T> *pos = root;
    while(pos){
        if(x > pos->value){
            pos = pos->right;
        }else{
            ans += nodeSize(pos->right) + 1;
            pos = pos->left;
        }
    }
    return ans;
}

int equalTo(T & x){

```

```

    return lessThanOrEqual(x) - lessThan(x);
}

void build(AVLNode<T> *& pos, vector<T> & arr, int i, int j){
    if(i > j) return;
    int m = i + ((j - i) >> 1);
    pos = new AVLNode<T>(arr[m]);
    build(pos->left, arr, i, m - 1);
    build(pos->right, arr, m + 1, j);
    pos->update();
}

void build(vector<T> & arr){
    build(root, arr, 0, (int)arr.size() - 1);
}

void output(AVLNode<T> *pos, vector<T> & arr, int & i){
    if(pos){
        output(pos->left, arr, i);
        arr[++i] = pos->value;
        output(pos->right, arr, i);
    }
}

void output(vector<T> & arr){
    int i = -1;
    output(root, arr, i);
}
};

```

9.5. Treap

```

struct Treap{
    Treap *left, *right;
    int value;
    int key, size;

    Treap(int value = 0): value(value), key(rand()), size(1),
        ↪ left(NULL), right(NULL){}
}

```

```

inline void update(){
    size = 1 + (left ? left->size : 0) + (right ? right->size :
        ↪ 0);
}

inline int nodeSize(Treap* &pos){
    return pos ? pos->size : 0;
}

void merge(Treap* &T, Treap* T1, Treap* T2){
    if(!T1) T = T2;
    else if(!T2) T = T1;
    else if(T1->key > T2->key)
        merge(T1->right, T1->right, T2), T = T1;
    else
        merge(T2->left, T1, T2->left), T = T2;
    if(T) T->update();
}

void split(Treap* T, int x, Treap* &T1, Treap* &T2){
    if(!T)
        return void(T1 = T2 = NULL);
    if(x < T->value)
        split(T->left, x, T1, T->left), T2 = T;
    else
        split(T->right, x, T->right, T2), T1 = T;
    if(T) T->update();
}

Treap* search(Treap* T, int x){
    while(T){
        if(x == T->value) break;
        T = (x < T->value ? T->left : T->right);
    }
    return T;
}

void insert(Treap* &T, Treap* x){
    if(!T) T = x;
    else if(x->key > T->key)

```

```

    split(T, x->value, x->left, x->right), T = x;
else
    insert(x->value < T->value ? T->left : T->right, x);
if(T) T->update();
}

void insert(Treap* &T, int x){insert(T, new Treap(x));}

void erase(Treap* &T, int x){
    if(!T) return;
    if(T->value == x)
        merge(T, T->left, T->right);
    else
        erase(x < T->value ? T->left : T->right, x);
    if(T) T->update();
}

Treap* updateVal(Treap* &T, int old, int New){
    if(search(T, old))
        erase(T, old), insert(T, New);
}

int lessThan(Treap* T, int x){
    int ans = 0;
    while(T){
        if(x > T->value){
            ans += nodeSize(T->left) + 1;
            T = T->right;
        }else{
            T = T->left;
        }
    }
    return ans;
}

int kth(Treap* T, int i){
    if(i < 0 || i >= nodeSize(T)) return -1;
    while(i != nodeSize(T->left)){
        if(i < nodeSize(T->left)){
            T = T->left;
        }else{

```

```

            i -= nodeSize(T->left) + 1;
            T = T->right;
        }
    }
    return T->value;
}

//implicit treap
void split(Treap* T, int i, Treap* &T1, Treap* &T2, int inc){
    if(!T)
        return void(T1 = T2 = NULL);
    int curr = inc + nodeSize(T->left);
    if(i <= curr)
        split(T->left, i, T1, T->left, inc), T2 = T;
    else
        split(T->right, i, T->right, T2, curr + 1), T1 = T;
    if(T) T->update();
}

//insert the element "x" at position "i"
void insert_at(Treap* &T, int x, int i){
    Treap *T1 = NULL, *T2 = NULL;
    split(T, i, T1, T2, 0);
    merge(T1, T1, new Treap(x));
    merge(T, T1, T2);
}

void erase_at(Treap* &T, int i, int inc){
    if(!T) return;
    int curr = inc + nodeSize(T->left);
    if(i == curr)
        merge(T, T->left, T->right);
    else
        erase_at(i < curr ? T->left : T->right, i, i < curr ? inc :
            ↪ curr + 1);
    if(T) T->update();
}

//delete element at position "i"
void erase_at(Treap* &T, int i){erase_at(T, i, 0);}

```

```
//update value of element at position "i" with "x"
```

```
void update_at(Treap* T, int x, int i){
    if(i < 0 || i >= nodeSize(T)) return;
    while(i != nodeSize(T->left)){
        if(i < nodeSize(T->left)){
            T = T->left;
        }else{
            i -= nodeSize(T->left) + 1;
            T = T->right;
        }
    }
    T->value = x;
}
```

```
void inorder(Treap* T){
    if(!T) return;
    inorder(T->left);
    cout << T->value << " ";
    inorder(T->right);
}
```

9.6. Ordered Set C++

```
#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;

int main(){
    int t, n, m;
    ordered_set conj;
    while(cin >> t && t != -1){
        cin >> n;
        if(t == 0){ //insert
            conj.insert(n);
        }else if(t == 1){ //search
            if(conj.find(n) != conj.end()) cout << "Found\n";
            else cout << "Not found\n";
        }
    }
}
```

```
}else if(t == 2){ //delete
    conj.erase(n);
}else if(t == 3){ //update
    cin >> m;
    if(conj.find(n) != conj.end()){
        conj.erase(n);
        conj.insert(n);
    }
}else if(t == 4){ //lower bound
    cout << conj.order_of_key(n) << "\n";
}else if(t == 5){ //get nth element
    auto pos = conj.find_by_order(n);
    if(pos != conj.end()) cout << *pos << "\n";
    else cout << "-1\n";
}
}
return 0;
}
```

9.7. Splay Tree

9.8. Sparse table

```
template<typename T>
struct SparseTable{
    vector<vector<T>> ST;
    vector<int> logs;
    int K, N;

    SparseTable(vector<T> & arr){
        N = arr.size();
        K = log2(N) + 2;
        ST.assign(K + 1, vector<T>(N));
        logs.assign(N + 1, 0);
        for(int i = 2; i <= N; ++i)
            logs[i] = logs[i >> 1] + 1;
        for(int i = 0; i < N; ++i)
            ST[0][i] = arr[i];
        for(int j = 1; j <= K; ++j)
            for(int i = 0; i + (1 << j) <= N; ++i)
```

```

    ST[j][i] = min(ST[j - 1][i], ST[j - 1][i + (1 << (j -
        ↪ 1))]);
}

T sum(int l, int r){
    T ans = 0;
    for(int j = K; j >= 0; --j){
        if((1 << j) <= r - l + 1){
            ans += ST[j][l];
            l += 1 << j;
        }
    }
    return ans;
}

T minimal(int l, int r){
    int j = logs[r - l + 1];
    return min(ST[j][l], ST[j][r - (1 << j) + 1]);
}
};

```

9.9. Wavelet Tree

```

struct WaveletTree{
    int lo, hi;
    WaveletTree *left, *right;
    vector<int> freq;

    //queries indexed in base 1, complexity O(log(max_element))
    //build from [from, to) with non-negative values in range [x,
    ↪ y]
    WaveletTree(vector<int>::iterator from, vector<int>::iterator
    ↪ to, int x, int y): lo(x), hi(y){
        if(lo == hi || from >= to) return;
        int m = (lo + hi) / 2;
        auto f = [m](int x){
            return x <= m;
        };
        freq.reserve(to - from + 1);
        freq.push_back(0);

```

```

        for(auto it = from; it != to; ++it)
            freq.push_back(freq.back() + f(*it));
        auto pivot = stable_partition(from, to, f);
        left = new WaveletTree(from, pivot, lo, m);
        right = new WaveletTree(pivot, to, m + 1, hi);
    }

    //kth element in [l, r]
    int kth(int l, int r, int k){
        if(l > r) return 0;
        if(lo == hi) return lo;
        int lb = freq[l - 1], rb = freq[r];
        int inLeft = rb - lb;
        if(k <= inLeft) return left->kth(lb + 1, rb, k);
        else return right->kth(l - lb, r - rb, k - inLeft);
    }

    //number of elements less than or equal to k in [l, r]
    int lessThanOrEqual(int l, int r, int k){
        if(l > r || k < lo) return 0;
        if(hi <= k) return r - l + 1;
        int lb = freq[l - 1], rb = freq[r];
        return left->lessThanOrEqual(lb + 1, rb, k) +
            ↪ right->lessThanOrEqual(l - lb, r - rb, k);
    }

    //number of elements equal to k in [l, r]
    int equalTo(int l, int r, int k){
        if(l > r || k < lo || k > hi) return 0;
        if(lo == hi) return r - l + 1;
        int lb = freq[l - 1], rb = freq[r];
        int m = (lo + hi) / 2;
        if(k <= m) return left->equalTo(lb + 1, rb, k);
        else return right->equalTo(l - lb, r - rb, k);
    }
};

```

9.10. Red Black Tree

10. Cadenas

10.1. Trie

```

struct Node{
    bool isWord = false;
    map<char, Node*> letters;
};

struct Trie{
    Node* root;

    Trie(){
        root = new Node();
    }

    inline bool exists(Node * actual, const char & c){
        return actual->letters.find(c) != actual->letters.end();
    }

    void InsertWord(const string& word){
        Node* current = root;
        for(auto & c : word){
            if(!exists(current, c))
                current->letters[c] = new Node();
            current = current->letters[c];
        }
        current->isWord = true;
    }

    bool FindWord(const string& word){
        Node* current = root;
        for(auto & c : word){
            if(!exists(current, c))
                return false;
            current = current->letters[c];
        }
        return current->isWord;
    }

    void printRec(Node * actual, string acum){

```

```

        if(actual->isWord){
            cout << acum << "\n";
        }
        for(auto & next : actual->letters)
            printRec(next.second, acum + next.first);
    }

    void printWords(const string & prefix){
        Node * actual = root;
        for(auto & c : prefix){
            if(!exists(actual, c)) return;
            actual = actual->letters[c];
        }
        printRec(actual, prefix);
    }
};

```

10.2. KMP

```

struct kmp{
    vector<int> aux;
    string pattern;

    kmp(string pattern){
        this->pattern = pattern;
        aux.resize(pattern.size());
        int i = 1, j = 0;
        while(i < pattern.size()){
            if(pattern[i] == pattern[j])
                aux[i++] = ++j;
            else{
                if(j == 0) aux[i++] = 0;
                else j = aux[j - 1];
            }
        }
    }

    vector<int> search(string & text){
        vector<int> ans;
        int i = 0, j = 0;

```



```

while(i < text.size() && j < pattern.size()){
    if(text[i] == pattern[j]){
        ++i, ++j;
        if(j == pattern.size()){
            ans.push_back(i - j);
            j = aux[j - 1];
        }
    }else{
        if(j == 0) ++i;
        else j = aux[j - 1];
    }
}
return ans;
}
};

```

10.3. Aho-Corasick

```

const int M = 26;
struct node{
    vector<int> child;
    int p = -1;
    char c = 0;
    int suffixLink = -1, endLink = -1;
    int id = -1;

    node(int p = -1, char c = 0) : p(p), c(c){
        child.resize(M, -1);
    }
};

struct AhoCorasick{
    vector<node> t;
    vector<int> lenghts;
    int wordCount = 0;

    AhoCorasick(){
        t.emplace_back();
    }
}

```

```

void add(const string & s){
    int u = 0;
    for(char c : s){
        if(t[u].child[c-'a'] == -1){
            t[u].child[c-'a'] = t.size();
            t.emplace_back(u, c);
        }
        u = t[u].child[c-'a'];
    }
    t[u].id = wordCount++;
    lenghts.push_back(s.size());
}

void link(int u){
    if(u == 0){
        t[u].suffixLink = 0;
        t[u].endLink = 0;
        return;
    }
    if(t[u].p == 0){
        t[u].suffixLink = 0;
        if(t[u].id != -1) t[u].endLink = u;
        else t[u].endLink = t[t[u].suffixLink].endLink;
        return;
    }
    int v = t[t[u].p].suffixLink;
    char c = t[u].c;
    while(true){
        if(t[v].child[c-'a'] != -1){
            t[u].suffixLink = t[v].child[c-'a'];
            break;
        }
        if(v == 0){
            t[u].suffixLink = 0;
            break;
        }
        v = t[v].suffixLink;
    }
    if(t[u].id != -1) t[u].endLink = u;
    else t[u].endLink = t[t[u].suffixLink].endLink;
}

```

```

void build(){
    queue<int> Q;
    Q.push(0);
    while(!Q.empty()){
        int u = Q.front(); Q.pop();
        link(u);
        for(int v = 0; v < M; ++v)
            if(t[u].child[v] != -1)
                Q.push(t[u].child[v]);
    }
}

int match(const string & text){
    int u = 0;
    int ans = 0;
    for(int j = 0; j < text.size(); ++j){
        int i = text[j] - 'a';
        while(true){
            if(t[u].child[i] != -1){
                u = t[u].child[i];
                break;
            }
            if(u == 0) break;
            u = t[u].suffixLink;
        }
        int v = u;
        while(true){
            v = t[v].endLink;
            if(v == 0) break;
            ++ans;
            int idx = j + 1 - lengths[t[v].id];
            cout << "Found word #" << t[v].id << " at position " <<
                idx << "\n";
            v = t[v].suffixLink;
        }
    }
    return ans;
}
};

```

10.4. Rabin-Karp

10.5. Suffix Array

10.6. Función Z

11. Varios

11.1. Lectura y escritura de `__int128`

```
//cout for __int128
ostream &operator<<(ostream &os, const __int128 & value){
    char buffer[64];
    char *pos = end(buffer) - 1;
    *pos = '\0';
    __int128 tmp = value < 0 ? -value : value;
    do{
        --pos;
        *pos = tmp % 10 + '0';
        tmp /= 10;
    }while(tmp != 0);
    if(value < 0){
        --pos;
        *pos = '-';
    }
    return os << pos;
}
```

```
//cin for __int128
istream &operator>>(istream &is, __int128 & value){
    char buffer[64];
    is >> buffer;
    char *pos = begin(buffer);
    int sgn = 1;
    value = 0;
    if(*pos == '-'){
        sgn = -1;
        ++pos;
    }else if(*pos == '+'){
        ++pos;
    }
    while(*pos != '\0'){
        value = (value << 3) + (value << 1) + (*pos - '0');
        ++pos;
    }
    value *= sgn;
    return is;
}
```

```
}
```

11.2. Longest Common Subsequence (LCS)

```
int lcs(string & a, string & b){
    int m = a.size(), n = b.size();
    vector<vector<int>> aux(m + 1, vector<int>(n + 1));
    for(int i = 1; i <= m; ++i){
        for(int j = 1; j <= n; ++j){
            if(a[i - 1] == b[j - 1])
                aux[i][j] = 1 + aux[i - 1][j - 1];
            else
                aux[i][j] = max(aux[i - 1][j], aux[i][j - 1]);
        }
    }
    return aux[m][n];
}
```

11.3. Longest Increasing Subsequence (LIS)

```
int lis(vector<int> & arr){
    if(arr.size() == 0) return 0;
    vector<int> aux(arr.size());
    int ans = 1;
    aux[0] = arr[0];
    for(int i = 1; i < arr.size(); ++i){
        if(arr[i] < aux[0])
            aux[0] = arr[i];
        else if(arr[i] > aux[ans - 1])
            aux[ans++] = arr[i];
        else
            aux[lower_bound(aux.begin(), aux.begin() + ans, arr[i]) -
                aux.begin()] = arr[i];
    }
    return ans;
}
```

11.4. Levenshtein Distance

```
int LevenshteinDistance(string & a, string & b){
    int m = a.size(), n = b.size();
    vector<vector<int>> aux(m + 1, vector<int>(n + 1));
    for(int i = 1; i <= m; ++i)
        aux[i][0] = i;
    for(int j = 1; j <= n; ++j)
        aux[0][j] = j;
    for(int j = 1; j <= n; ++j)
        for(int i = 1; i <= m; ++i)
            aux[i][j] = min({aux[i-1][j] + 1, aux[i][j-1] + 1,
                ↪ aux[i-1][j-1] + (a[i-1] != b[j-1])});
    return aux[m][n];
}
```

11.5. Día de la semana

```
//0:saturday, 1:sunday, ..., 6:friday
int dayOfWeek(int d, int m, lli y){
    if(m == 1 || m == 2){
        m += 12;
        --y;
    }
    int k = y % 100;
    lli j = y / 100;
    return (d + 13*(m+1)/5 + k + k/4 + j/4 + 5*j) % 7;
}
```

11.6. 2SAT

```
struct satisfiability_twosat{
    int n;
    vector<vector<int>> imp;

    satisfiability_twosat(int n) : n(n), imp(2 * n) {}

    void add_edge(int u, int v){imp[u].push_back(v);}
}
```

```
int neg(int u){return (n << 1) - u - 1;}

void implication(int u, int v){
    add_edge(u, v);
    add_edge(neg(v), neg(u));
}

vector<bool> solve(){
    int size = 2 * n;
    vector<int> S, B, I(size);

    function<void(int)> dfs = [&](int u){
        B.push_back(I[u] = S.size());
        S.push_back(u);

        for(int v : imp[u])
            if(!I[v]) dfs(v);
            else while (I[v] < B.back()) B.pop_back();

        if(I[u] == B.back())
            for(B.pop_back(), ++size; I[u] < S.size();
                ↪ S.pop_back())
                I[S.back()] = size;
    };

    for(int u = 0; u < 2 * n; ++u)
        if(!I[u]) dfs(u);

    vector<bool> values(n);

    for(int u = 0; u < n; ++u)
        if(I[u] == I[neg(u)]) return {};
        else values[u] = I[u] < I[neg(u)];

    return values;
}
```

11.7. Código Gray

```
//gray code  
int gray(int n){  
    return n ^ (n >> 1);  
}
```

```
//inverse gray code  
int inv_gray(int g){  
    int n = 0;  
    while(g){  
        n ^= g;  
        g >>= 1;  
    }  
    return n;  
}
```