

Índice

1. Teoría de números	6		
1.1. Funciones básicas	6	1.3.2. Potencia de un primo que divide a un factorial	11
1.1.1. Función piso y techo	6	1.3.3. Factorización de un factorial	11
1.1.2. Exponenciación y multiplicación binaria	6	1.3.4. Factorial módulo p	11
1.1.3. Mínimo común múltiplo y máximo común divisor	6	1.3.5. Factorización usando Pollard-Rho	11
1.1.4. Euclides extendido e inverso modular	6	1.4. Funciones aritméticas famosas	12
1.1.5. Todos los inversos módulo p	7	1.4.1. Función σ	12
1.1.6. Exponenciación binaria modular	7	1.4.2. Función Ω	12
1.1.7. Teorema chino del residuo	7	1.4.3. Función ω	12
1.1.8. Teorema chino del residuo generalizado	7	1.4.4. Función φ de Euler	12
1.1.9. Coeficiente binomial	7	1.4.5. Función μ	12
1.1.10. Fibonacci	8	1.5. Orden multiplicativo, raíces primitivas y raíces de la unidad	13
1.2. Cribas	8	1.5.1. Función λ de Carmichael	13
1.2.1. Criba de divisores	8	1.5.2. Orden multiplicativo módulo m	13
1.2.2. Criba de primos	8	1.5.3. Número de raíces primitivas (generadores) módulo m	13
1.2.3. Criba de factor primo más pequeño	8	1.5.4. Test individual de raíz primitiva módulo m	13
1.2.4. Criba de factor primo más grande	9	1.5.5. Test individual de raíz k -ésima de la unidad módulo m	13
1.2.5. Criba de factores primos	9	1.5.6. Encontrar la primera raíz primitiva módulo m	14
1.2.6. Criba de la función φ de Euler	9	1.5.7. Encontrar la primera raíz k -ésima de la unidad módulo m	14
1.2.7. Criba de la función μ	9	1.5.8. Logaritmo discreto	14
1.2.8. Triángulo de Pascal	9	1.5.9. Raíz k -ésima discreta	14
1.2.9. Segmented sieve	9	1.5.10. Algoritmo de Tonelli-Shanks para raíces cuadradas módulo p	15
1.2.10. Criba de primos lineal	10	1.6. Particiones	15
1.2.11. Criba lineal para funciones multiplicativas	10	1.6.1. Función P (particiones de un entero positivo)	15
1.3. Factorización	10	1.6.2. Función Q (particiones de un entero positivo en distintos sumandos)	15
1.3.1. Factorización de un número	10	1.6.3. Número de factorizaciones ordenadas	16

1.6.4. Número de factorizaciones no ordenadas	16	4. FFT	28
1.7. Otros	17	4.1. Declaraciones previas	28
1.7.1. Cambio de base	17	4.2. FFT con raíces de la unidad complejas	28
1.7.2. Fracciones continuas	17	4.3. FFT con raíces de la unidad en \mathbb{Z}_p (NTT)	28
1.7.3. Ecuación de Pell	18	4.3.1. Valores para escoger el generador y el módulo	29
1.7.4. Números de Bell	18	4.4. Multiplicación de polinomios (convolución lineal)	29
1.7.5. Números de Stirling	18	4.5. Aplicaciones	29
1.7.6. Números de Euler	18	4.5.1. Multiplicación de números enteros grandes	29
1.7.7. Prime counting function in sublinear time	19	4.5.2. Recíproco de un polinomio	30
1.7.8. Suma de la función piso	19	4.5.3. Raíz cuadrada de un polinomio	30
2. Números racionales	20	4.5.4. Logaritmo y exponencial de un polinomio	30
2.1. Estructura <code>fraccion</code>	20	4.5.5. Cociente y residuo de dos polinomios	31
3. Álgebra lineal	21	4.5.6. Multievaluación rápida	31
3.1. Estructura <code>matrix</code>	21	4.5.7. DFT con tamaño de vector arbitrario (algoritmo de Bluestein)	32
3.2. Transpuesta y traza	23	4.6. Convolución de dos vectores reales con solo dos FFT's	32
3.3. Gauss Jordan	23	4.7. Convolución con módulo arbitrario	32
3.4. Matriz escalonada por filas y reducida por filas	23	4.8. Transformada rápida de Walsh-Hadamard	33
3.5. Matriz inversa	23	5. Geometría	34
3.6. Determinante	24	5.1. Estructura <code>point</code>	34
3.7. Matriz de cofactores y adjunta	24	5.2. Líneas y segmentos	35
3.8. Factorización $PA = LU$	24	5.2.1. Verificar si un punto pertenece a una línea o segmento	35
3.9. Polinomio característico	25	5.2.2. Intersección de líneas	35
3.10. Gram-Schmidt	25	5.2.3. Intersección línea-segmento	35
3.11. Recurrencias lineales	25	5.2.4. Intersección de segmentos	35
3.12. Berlekamp-Massey	25	5.2.5. Distancia punto-recta	36
3.13. Simplex	26	5.3. Círculos	36

5.3.1. Distancia punto-círculo	36	6.2. Definiciones	45
5.3.2. Proyección punto exterior a círculo	36	6.3. DFS genérica	46
5.3.3. Puntos de tangencia de punto exterior	36	6.4. Dijkstra	46
5.3.4. Intersección línea-círculo	36	6.5. Bellman Ford	46
5.3.5. Centro y radio a través de tres puntos	36	6.6. Floyd	47
5.3.6. Intersección de círculos	37	6.7. Cerradura transitiva $O(V^3)$	47
5.3.7. Contención de círculos	37	6.8. Cerradura transitiva $O(V^2)$	47
5.3.8. Tangentes	37	6.9. Verificar si el grafo es bipartito	47
5.3.9. Smallest enclosing circle	38	6.10. Orden topológico	48
5.4. Polígonos	38	6.11. Detectar ciclos	48
5.4.1. Perímetro y área de un polígono	38	6.12. Puentes y puntos de articulación	48
5.4.2. Envolverte convexa (convex hull) de un polígono . .	38	6.13. Componentes fuertemente conexas	49
5.4.3. Verificar si un punto pertenece al perímetro de un polígono	39	6.14. Árbol mínimo de expansión (Kruskal)	49
5.4.4. Verificar si un punto pertenece a un polígono	39	6.15. Máximo emparejamiento bipartito	49
5.4.5. Verificar si un punto pertenece a un polígono con- vexo $O(\log n)$	39	6.16. Circuito euleriano	50
5.4.6. Cortar un polígono con una recta	39	7. Árboles	50
5.4.7. Centroide de un polígono	40	7.1. Estructura <code>tree</code>	50
5.4.8. Pares de puntos antipodales	40	7.2. k -ésimo ancestro	51
5.4.9. Diámetro y ancho	40	7.3. LCA	51
5.4.10. Smallest enclosing rectangle	40	7.4. Distancia entre dos nodos	51
5.5. Par de puntos más cercanos	41	7.5. HLD	51
5.6. Vantage Point Tree (puntos más cercanos a cada punto) . .	41	7.6. Link Cut	51
5.7. Suma Minkowski	42	8. Flujos	52
5.8. Triangulación de Delaunay	42	8.1. Estructura <code>flowEdge</code>	52
6. Grafos	45	8.2. Estructura <code>flowGraph</code>	52
6.1. Disjoint Set	45	8.3. Algoritmo de Edmonds-Karp $O(VE^2)$	52

8.4. Algoritmo de Dinic $O(V^2E)$	52	10.6. Función Z	68
8.5. Flujo máximo de costo mínimo	53		
8.6. Hungariano	53		
9. Estructuras de datos	54	11. Varios	69
9.1. Segment Tree	54	11.1. Lectura y escritura de <code>__int128</code>	69
9.1.1. Minimalistic: Point updates, range queries	54	11.2. Longest Common Subsequence (LCS)	69
9.1.2. Dynamic: Range updates and range queries	55	11.3. Longest Increasing Subsequence (LIS)	69
9.1.3. Static: Range updates and range queries	55	11.4. Levenshtein Distance	69
9.1.4. Persistent: Point updates, range queries	56	11.5. Día de la semana	70
9.2. Fenwick Tree	57	11.6. 2SAT	70
9.3. SQRT Decomposition	57	11.7. Código Gray	70
9.4. AVL Tree	58	11.8. Contar número de unos en binario en un rango	71
9.5. Treap	61	11.9. Números aleatorios en C++11	71
9.6. Sparse table	64		
9.6.1. Normal	64	12. Fórmulas y notas	71
9.6.2. Disjoint	64	12.1. Números de Stirling del primer tipo	71
9.7. Wavelet Tree	65	12.2. Números de Stirling del segundo tipo	71
9.8. Ordered Set C++	65	12.3. Números de Euler	71
9.9. Splay Tree	66	12.4. Números de Catalan	72
9.10. Red Black Tree	66	12.5. Números de Bell	72
10. Cadenas	66	12.6. Números de Bernoulli	72
10.1. Trie	66	12.7. Fórmula de Faulhaber	72
10.2. KMP	67	12.8. Función Beta	72
10.3. Aho-Corasick	67	12.9. Función zeta de Riemann	72
10.4. Rabin-Karp	68	12.10. Funciones generadoras	73
10.5. Suffix Array	68	12.11. Números armónicos	73
		12.12. Aproximación de Stirling	73
		12.13. Ternas pitagóricas	73
		12.14. Árbol de Stern–Brocot	73

12.15Combinatoria 73

12.16Grafos 74

12.17Teoría de números 75

12.18Primos 76

12.19Números primos de Mersenne 76

12.20Números primos de Fermat 76

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 power(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

```

//find all inverses (from 1 to p-1) modulo p
vector<lli> allInverses(lli p){
    vector<lli> ans(p);
    ans[1] = 1;
    for(lli i = 2; i < p; ++i)
        ans[i] = p - (p / i) * ans[p % i] % p;
    return ans;
}

```

1.1.6. Exponenciación binaria modular

```

lli powerMod(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> & m){
    lli prod = 1, p, ans = 0;
    for(lli & ni : m) prod *= ni;
    for(int i = 0; i < a.size(); ++i){
        p = prod / m[i];
        ans += (a[i] % m[i]) * modularInverse(p, m[i]) % prod * p %
        ↪ prod;
        while(ans >= prod) ans -= prod; while(ans < 0) ans += prod;
    }
    return {ans, prod};
}

```

1.1.8. Teorema chino del residuo generalizado

```

//generalized chinese remainder theorem
//the modulus doesn't need to be pairwise coprime
pair<lli, lli> crt(const vector<lli> & a, const vector<lli> & m){
    lli a0 = a[0] % m[0], m0 = m[0], a1, m1, s, t, d, M;
    for(int i = 1; i < a.size(); ++i){
        a1 = a[i] % m[i], m1 = m[i];
        d = extendedGcd(m0, m1, s, t);
        if((a0 - a1) % d != 0) return {0, 0}; //error, no solution
        M = m0 * (m1 / d);
        a0 = a0 * t % M * (m1 / d) % M + a1 * s % M * (m0 / d) % M;
        while(a0 >= M) a0 -= M; while(a0 < 0) a0 += M;
        m0 = M;
    }
    while(a0 >= m0) a0 -= m0; while(a0 < 0) a0 += m0;
    return {a0, m0};
}

```

1.1.9. 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;
}

```

1.1.10. 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 factor primo más grande

```
vector<int> greatestPrime;
void greatestPrimeSieve(int n){
    greatestPrime.resize(n + 1, 1);
    greatestPrime[0] = greatestPrime[1] = 0;
    for(int i = 2; i <= n; ++i) greatestPrime[i] = i;
    for(int i = 2; i <= n; i++){
        if(greatestPrime[i] == i)
            for(int j = i; j <= n; j += i)
                greatestPrime[j] = i;
    }
}
```

1.2.5. 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);
    }
}
```

1.2.6. 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;
}
```

1.2.7. 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];
}
```

1.2.8. 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];
        }
    }
}
```

1.2.9. 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;
}

```

1.2.10. 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;
}

```

1.2.11. Criba lineal para funciones multiplicativas

*//suppose $f(n)$ is a multiplicative function and
 //we want to find $f(1), f(2), \dots, f(n)$
 //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)
            f[i] = g(i, 1);
            primes.push_back(i);
            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. Factorial módulo p

```

//Finds (n!/p^m) mod p, where m is the largest power of p
//that divides n!, p must be prime
lli factmod(lli n, lli p){
    lli ans = 1;
    while(n > 1){
        ans = ans * ((n/p) % 2 ? p-1 : 1) % p;
        for(lli i = 2; i <= n%p; ++i)
            ans = ans * i % p;
        n /= p;
    }
    return ans;
}

```

1.3.5. Factorización usando Pollard-Rho

```

bool isPrimeMillerRabin(lli n){
    if(n < 2) return false;
    if(n == 2) return true;
    lli d = n - 1, s = 0;
    for(; !(d & 1); d >>= 1, ++s);
    for(int i = 0; i < 16; ++i){
        lli a = 1 + rand() % (n - 1);
        lli m = powerMod(a, d, n);
        if(m == 1 || m == n - 1) goto exit;
        for(int k = 0; k < s; ++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){
        x = x * (x + b) % n + a;
        y = y * (y + b) % n + a;
        y = y * (y + b) % n + a;
        d = gcd(abs(x - y), n);
    }
    return d;
}

map<lli, int> fact;
void factorizePollardRho(lli n, bool clean = true){
    if(clean) fact.clear();
    while(n > 1 && !isPrimeMillerRabin(n)){
        lli f = n;
        for(; f == n; f = getFactor(n));
        n /= f;
        factorizePollardRho(f, false);
        for(auto & it : fact){
            while(n % it.first == 0){

```

```

        n /= it.first;
        ++it.second;
    }
}
}
if(n > 1) ++fact[n];
}

```

1.4. Funciones aritméticas 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 = power(p, pot);
            ans *= (power(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 = power(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 0;
    lli order = phi(m);
    auto f = factorize(order);
    for(auto & factor : f){
        lli p = factor.first;
        int a = factor.second;
        order /= power(p, a);
        lli tmp = powerMod(x, order, m);
        while(tmp != 1){
            tmp = powerMod(tmp, p, m);
            order *= p;
        }
    }
    return order;
}
```

```
//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(powerMod(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(powerMod(x, k, m) != 1) return false;
    auto f = factorize(k);
    for(auto & factor : f){
        lli p = factor.first;
        if(powerMod(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(powerMod(x, order / p, m) == 1){
                test = false;
                break;
            }
        }
        if(test) return x;
    }
    return -1; //not found
}
```

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(powerMod(x, k, m) != 1) continue;
        bool test = true;
        for(auto & factor : f){
            lli p = factor.first;
            if(powerMod(x, k / p, m) == 1){
                test = false;
                break;
            }
        }
        if(test) return x;
    }
    return -1; //not found
}
```

1.5.8. Logaritmo discreto

```
// Solves for x in the equation a^x = b mod m
pair<lli, lli> discreteLogarithm(lli a, lli b, lli m){
    lli m1 = m, pw = 1, d, x, y, nonRep = 0;
    for(;; (d = gcd(a, m1)) > 1; ++nonRep, m1 /= d, pw = pw * a % m){
        if(pw == b) return {nonRep, 0}; //aperiodic solution found
    }
    d = extendedGcd(pw, m, x, y);
    if(b % d > 0) return {-1, 0}; //solution not found
    b = x * (b / d) % m;
    if(b < 0) b += m;
    lli order = multiplicativeOrder(a, m1);
    lli n = sqrt(order) + 1;
    lli a_n = powerMod(a, n, m1);
    unordered_map<lli, lli> firstHalf;
    pw = a_n;
    for(lli p = 1; p <= n; ++p, pw = pw * a_n % m1){
        firstHalf[pw] = p;
    }
    pw = b % m1;
    for(lli q = 0; q <= n; ++q, pw = pw * a % m1){
        if(firstHalf.count(pw)) return {nonRep + (n * firstHalf[pw] -
            ↪ q) % order, order}; //periodic solution found
        }
    }
    return {-1, 0}; //solution not found
}
```

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 = powerMod(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] = powerMod(g, y0.first, m);
    lli inc = powerMod(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;
}

```

1.5.10. Algoritmo de Tonelli-Shanks para raíces cuadradas módulo p

```

//finds x such that x^2 = a mod p
lli sqrtMod(lli a, lli p){
    a %= p;
    if(a < 0) a += p;
    if(a == 0) return 0;
    assert(powerMod(a, (p - 1) / 2, p) == 1);
    if(p % 4 == 3) return powerMod(a, (p + 1) / 4, p);
    lli s = p - 1;
    int r = 0;
    while((s & 1) == 0) ++r, s >>= 1;
    lli n = 2;
    while(powerMod(n, (p - 1) / 2, p) != p - 1) ++n;
    lli x = powerMod(a, (s + 1) / 2, p);
    lli b = powerMod(a, s, p);
    lli g = powerMod(n, s, p);
    while(true){
        lli t = b;
        int m = 0;
        for(; m < r; ++m){
            if(t == 1) break;
            t = t * t % p;
        }
        if(m == 0) return x;
        lli gs = powerMod(g, 1 << (r - m - 1), p);
        g = gs * gs % p;
        x = x * gs % p;
        b = b * g % p;
        r = m;
    }
}

```

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 <= q; 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 {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({p, q})){
            period -= pairs[{p, q}];
            break;
        }
    }
}

```

```

    }
    coef.push_back(a);
    pairs[{p, q}] = period++;
}
return {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 {den, num};
}

```

1.7.4. Números de Bell

```

//number of ways to partition a set of  $n$  elements
//the  $n$ th bell number is at Bell[n][0]
vector<vector<int>> Bell;
void bellNumbers(int n){
    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];
    }
}

```

1.7.5. Números de Stirling

```

//s(n, k) represents the number of permutations
//of  $n$  elements with  $k$  disjoint cycles
vector<vector<lli>> stirling1;
void stirlingNumber1stKind(lli n){
    stirling1.resize(n+1, vector<lli>(n+1));
    stirling1[0][0] = 1;
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= i; ++j)
            stirling1[i][j] = (i-1) * stirling1[i-1][j] +
            ↪ stirling1[i-1][j-1];
}

//S(n, k) represents the number of ways to
//partition a set of  $n$  object into  $k$  non-empty
//distinct subsets
vector<vector<lli>> stirling2;
void stirlingNumber2ndKind(lli n){
    stirling2.resize(n+1, vector<lli>(n+1));
    stirling2[0][0] = 1;
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= i; ++j)
            stirling2[i][j] = j * stirling2[i-1][j] +
            ↪ stirling2[i-1][j-1];
}

```

1.7.6. Números de Euler

```

//euler(n, k) represents the number of permutations
//of  $1, \dots, n$  with exactly  $k$  numbers greater than
//the previous number
vector<vector<lli>> euler;
void eulerianNumbers(lli n){
    euler.resize(n+1, vector<lli>(n+1));
    for(int i = 1; i <= n; ++i){
        euler[i][0] = 1;
        for(int j = 1; j < i; ++j)
            euler[i][j] = (i-j) * euler[i-1][j-1] + (j+1) *
            ↪ euler[i-1][j];
    }
}

```

1.7.7. Prime counting function in sublinear time

```
const lli inv_2 = modularInverse(2, Mod);
const lli inv_6 = modularInverse(6, Mod);
const lli inv_30 = modularInverse(30, Mod);

lli sum(lli n, int k){
    n %= Mod;
    if(k == 0) return n;
    if(k == 1) return n * (n + 1) % Mod * inv_2 % Mod;
    if(k == 2) return n * (n + 1) % Mod * (2*n + 1) % Mod * inv_6 %
        ↪ Mod;
    if(k == 3) return powMod(n * (n + 1) % Mod * inv_2 % Mod, 2,
        ↪ Mod);
    if(k == 4) return n * (n + 1) % Mod * (2*n + 1) % Mod *
        ↪ (3*n*(n+1)%Mod - 1) % Mod * inv_30 % Mod;
    return 1;
}
```

*//finds the sum of the kth powers of the primes
 //less than or equal to n (0<=k<=4, add more if you need)*

```
lli SumPrimePi(lli n, int k){
    lli v = sqrt(n), p, temp, q, j, end, i, d;
    vector<lli> lo(v+2), hi(v+2);
    vector<bool> used(v+2);
    for(p = 1; p <= v; p++){
        lo[p] = sum(p, k) - 1;
        hi[p] = sum(n/p, k) - 1;
    }
    for(p = 2; p <= v; p++){
        if(lo[p] == lo[p-1]) continue;
        temp = lo[p-1];
        q = p * p;
        hi[1] -= (hi[p] - temp) * powMod(p, k, Mod) % Mod;
        if(hi[1] < 0) hi[1] += Mod;
        j = 1 + (p & 1);
        end = (v <= n/q) ? v : n/q;
        for(i = p + j; i <= 1 + end; i += j){
            if(used[i]) continue;
            d = i * p;
            if(d <= v)
                hi[i] -= (hi[d] - temp) * powMod(p, k, Mod) % Mod;
            else
                hi[i] -= (lo[n/d] - temp) * powMod(p, k, Mod) % Mod;
        }
    }
}
```

```
        if(hi[i] < 0) hi[i] += Mod;
    }
    if(q <= v)
        for(i = q; i <= end; i += p*j)
            used[i] = true;
    for(i = v; i >= q; i--){
        lo[i] -= (lo[i/p] - temp) * powMod(p, k, Mod) % Mod;
        if(lo[i] < 0) lo[i] += Mod;
    }
}
return hi[1] % Mod;
}
```

1.7.8. Suma de la función piso

*//finds sum(floor(p*i/q), 1<=i<=n)*

```
lli floorsSum(lli p, lli q, lli n){
    lli t = gcd(p, q);
    p /= t, q /= t;
    lli s = 0, z = 1;
    while(q && n){
        t = p/q;
        s += z*t*n*(n+1)/2;
        p -= q*t;
        t = n/q;
        s += z*p*t*(n+1) - z*t*(p*q*t + p + q - 1)/2;
        n -= q*t;
        t = n*p/q;
        s += z*t*n;
        n = t;
        swap(p, q);
        z = -z;
    }
    return s;
}
```

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 T>
struct matrix{
    vector<vector<T>> A;
    int m, n;

    matrix(int m, int n): m(m), n(n){
        A.resize(m, vector<T>(n, 0));
    }

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

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

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

    matrix operator+(const matrix & B) const{
        assert(m == B.m && n == B.n); //same dimensions
        matrix<T> C(m, n);
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                C[i][j] = A[i][j] + B[i][j];
            }
        }
        return C;
    }

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

    matrix operator-() const{

```

```

    matrix<T> 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{
    assert(n == B.m); // #columns of 1st matrix = #rows of 2nd
    ↪ matrix
    matrix<T> 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[k][j];
    return C;
}

matrix operator*(const T & c) const{
    matrix<T> 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 T & c){
    *this = *this * c;
    return *this;
}

```

```

matrix operator^(lli b) const{
    matrix<T> ans = matrix<T>::identity(n);
    matrix<T> 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) return false;
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++)
            if(A[i][j] != B[i][j]) return false;
    return true;
}

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

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

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

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

```

3.2. Transpuesta y traza

```

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

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

```

3.3. Gauss Jordan

```

//full: true: reduce above and below the diagonal, false: reduce
↪ only below
//makeOnes: true: make the elements in the diagonal ones, false:
↪ leave the diagonal unchanged
//For every elemental operation that we apply to the matrix,
//we will call to callback(operation, k, l, value).
//operation 1: multiply row "k" by "value"
//operation 2: swap rows "k" and "l"
//operation 3: add "value" times the row "l" to the row "k"
//It returns the rank of the matrix, and modifies it
int gauss_jordan(bool full = true, bool makeOnes = true,
    ↪ function<void(int, int, int, T)>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){
                    swapRows(i, f);
                    if(callback) callback(2, i, f, 0);
                    break;
                }
            }
        }
        if(A[i][j] != 0){

```

```

            T inv_mult = A[i][j].inverso();
            if(makeOnes && A[i][j] != 1){
                scaleRow(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){
                    T inv_adit = -A[f][j];
                    if(!makeOnes) inv_adit *= inv_mult;
                    addRow(f, i, inv_adit);
                    if(callback) callback(3, f, i, inv_adit);
                }
            }
            i++;
        }
        j++;
    }
    return i;
}

```

```

void gaussian_elimination(){
    gauss_jordan(false);
}

```

3.4. Matriz escalonada por filas y reducida por filas

```

matrix<T> reducedRowEchelonForm(){
    matrix<T> asoc = *this;
    asoc.gauss_jordan();
    return asoc;
}

matrix<T> rowEchelonForm(){
    matrix<T> asoc = *this;
    asoc.gaussian_elimination();
    return asoc;
}

```

3.5. Matriz inversa

```

bool invertible(){
    assert(m == n); //this is defined only for square matrices

```

```

    matrix<T> tmp = *this;
    return tmp.gauss_jordan(false) == n;
}

matrix<T> inverse(){
    assert(m == n); //this is defined only for square matrices
    matrix<T> tmp = *this;
    matrix<T> inv = matrix<T>::identity(n);
    auto callback = [&](int op, int a, int b, T e){
        if(op == 1){
            inv.scaleRow(a, e);
        }else if(op == 2){
            inv.swapRows(a, b);
        }else if(op == 3){
            inv.addRow(a, b, e);
        }
    };
    assert(tmp.gauss_jordan(true, true, callback) == n); //check
    ↪ non-invertible
    return inv;
}

```

3.6. Determinante

```

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

```

3.7. Matriz de cofactores y adjunta

```

matrix<T> minor(int x, int y){
    matrix<T> M(m-1, n-1);
    for(int i = 0; i < m-1; ++i)
        for(int j = 0; j < n-1; ++j)
            M[i][j] = A[i < x ? i : i+1][j < y ? j : j+1];
    return M;
}

T cofactor(int x, int y){
    T ans = minor(x, y).determinant();
    if((x + y) % 2 == 1) ans *= -1;
    return ans;
}

matrix<T> cofactorMatrix(){
    matrix<T> 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<T> adjugate(){
    if(invertible()) return inverse() * determinant();
    return cofactorMatrix().transpose();
}

```

3.8. Factorización $PA = LU$

```

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

```



```

    }
};
U.gauss_jordan(false, false, callback);
return {P, L, U};
}

```

3.9. Polinomio característico

```

vector<T> characteristicPolynomial(){
    matrix<T> M(n, n);
    vector<T> coef(n + 1);
    matrix<T> I = matrix<T>::identity(n);
    coef[n] = 1;
    for(int i = 1; i <= n; i++){
        M = (*this) * M + I * coef[n - i + 1];
        coef[n - i] = -((*this) * M).trace() / i;
    }
    return coef;
}

```

3.10. Gram-Schmidt

```

matrix<T> gram_schmidt(){
    //vectors are rows of the matrix (also in the answer)
    //the answer doesn't have the vectors normalized
    matrix<T> B = (*this) * (*this).transpose();
    matrix<T> ans = *this;
    auto callback = [&](int op, int a, int b, T e){
        if(op == 1){
            ans.scaleRow(a, e);
        }else if(op == 2){
            ans.swapRows(a, b);
        }else if(op == 3){
            ans.addRow(a, b, e);
        }
    };
    B.gauss_jordan(false, false, callback);
    return ans;
}

```

3.11. Recurrencias lineales

```

//Solves a linear homogeneous recurrence relation of degree "deg"
//of the form  $F(n) = a(d-1)*F(n-1) + a(d-2)*F(n-2) + \dots +$ 
 $\hookrightarrow a(1)*F(n-(d-1)) + a(0)*F(n-d)$ 
//with initial values  $F(0), F(1), \dots, F(d-1)$ 
//It finds the  $n$ th term of the recurrence,  $F(n)$ 
//The values of  $a[0, \dots, d]$  are in the array  $P[]$ 
lli solveRecurrence(const vector<lli> & P, const vector<lli> &
 $\hookrightarrow$  init, lli n){
    int deg = P.size();
    vector<lli> ans(deg), R(2*deg);
    ans[0] = 1;
    lli p = 1;
    for(lli v = n; v >= 1; p <= 1);
    do{
        int d = (n & p) != 0;
        fill(R.begin(), R.end(), 0);
        for(int i = 0; i < deg; i++)
            for(int j = 0; j < deg; j++)
                (R[i + j + d] += ans[i] * ans[j]) %= mod;
        for(int i = deg-1; i >= 0; i--)
            for(int j = 0; j < deg; j++)
                (R[i + j] += R[i + deg] * P[j]) %= mod;
        copy(R.begin(), R.begin() + deg, ans.begin());
    }while(p >= 1);
    lli nValue = 0;
    for(int i = 0; i < deg; i++)
        (nValue += ans[i] * init[i]) %= mod;
    return nValue;
}

```

3.12. Berlekamp-Massey

```

//Finds the shortest linear recurrence relation for the
//given init values. Only works for prime modulo.
vector<lli> BerlekampMassey(const vector<lli> & init){
    vector<lli> cur, ls;
    lli ld;
    for(int i = 0, m; i < init.size(); ++i){
        lli eval = 0;
        for(int j = 0; j < cur.size(); ++j)
            eval = (eval + init[i-j-1] * cur[j]) % mod;
    }
}

```

```

eval -= init[i];
if(eval < 0) eval += mod;
if(eval == 0) continue;
if(cur.empty()){
    cur.resize(i + 1);
    m = i;
    ld = eval;
}else{
    lli k = eval * inverse(ld, mod) % mod;
    vector<lli> c(i - m - 1);
    c.push_back(k);
    for(int j = 0; j < ls.size(); ++j)
        c.push_back((mod - ls[j]) * k % mod);
    if(c.size() < cur.size()) c.resize(cur.size());
    for(int j = 0; j < cur.size(); ++j){
        c[j] += cur[j];
        if(c[j] >= mod) c[j] -= mod;
    }
    if(i - m + ls.size() >= cur.size())
        ls = cur, m = i, ld = eval;
    cur = c;
}
}
if(cur.empty()) cur.push_back(0);
reverse(cur.begin(), cur.end());
return cur;
}

```

3.13. 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: ";
    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 << ",
        ↪ 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. Declaraciones previas

```
using lli = long long int;
using comp = complex<double>;
const 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();
    for(int i = 1, j = 0; i < n - 1; ++i){
        for(int k = n >> 1; (j ^= k) < k; k >>= 1);
        if(i < j) swap(X[i], X[j]);
    }
    vector<comp> wp(n>>1);
    for(int k = 1; k < n; k <<= 1){
        for(int j = 0; j < k; ++j)
            wp[j] = polar(1.0, PI * j / k * inv);
        for(int i = 0; i < n; i += k << 1){
            for(int j = 0; j < k; ++j){
                comp t = X[i + j + k] * wp[j];
                X[i + j + k] = X[i + j] - t;
                X[i + j] += t;
            }
        }
    }
    if(inv == -1)
        for(int i = 0; i < n; ++i)
            X[i] /= n;
}
```

4.3. FFT con raíces de la unidad en \mathbb{Z}_p (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;
}

lli powerMod(lli b, lli e, lli m){
    lli ans = 1;
    e %= m-1;
    if(e < 0) e += m-1;
    while(e){
        if(e & 1) ans = ans * b % m;
        e >>= 1;
        b = b * b % m;
    }
    return ans;
}

template<int prime, int gen>
void ntt(vector<int> & X, int inv){
    int n = X.size();
    for(int i = 1, j = 0; i < n - 1; ++i){
        for(int k = n >> 1; (j ^= k) < k; k >>= 1);
        if(i < j) swap(X[i], X[j]);
    }
    vector<lli> wp(n>>1, 1);
    for(int k = 1; k < n; k <<= 1){
        lli wk = powerMod(gen, inv * (prime - 1) / (k<<1), prime);
        for(int j = 1; j < k; ++j)
            wp[j] = wp[j - 1] * wk % prime;
        for(int i = 0; i < n; i += k << 1){
            for(int j = 0; j < k; ++j){
                int u = X[i + j], v = X[i + j + k] * wp[j] % prime;
                X[i + j] = u + v < prime ? u + v : u + v - prime;
                X[i + j + k] = u - v < 0 ? u - v + prime : u - v;
            }
        }
    }
}
```

```

if(inv == -1){
    lli nrev = inverse(n, prime);
    for(int i = 0; i < n; ++i)
        X[i] = X[i] * nrev % prime;
}
}

```

4.3.1. Valores para escoger el generador y el módulo

Generador (g)	Tamaño máxi- mo del arreglo (n)	Módulo p
3	2^{16}	$1 \times 2^{16} + 1 = 65537$
10	2^{18}	$3 \times 2^{18} + 1 = 786433$
3	2^{19}	$11 \times 2^{19} + 1 = 5767169$
3	2^{20}	$7 \times 2^{20} + 1 = 7340033$
3	2^{21}	$11 \times 2^{21} + 1 = 23068673$
3	2^{22}	$25 \times 2^{22} + 1 = 104857601$
3	2^{22}	$235 \times 2^{22} + 1 = 985661441$
26	2^{23}	$105 \times 2^{23} + 1 = 880803841$
3	2^{23}	$119 \times 2^{23} + 1 = 998244353$
11	2^{24}	$45 \times 2^{24} + 1 = 754974721$
3	2^{25}	$5 \times 2^{25} + 1 = 167772161$
3	2^{26}	$7 \times 2^{26} + 1 = 469762049$
31	2^{27}	$15 \times 2^{27} + 1 = 2013265921$

4.4. Multiplicación de polinomios (convolución lineal)

```

vector<comp> convolution(vector<comp> A, vector<comp> B){
    int sz = A.size() + B.size() - 1;
    int size = nearestPowerOfTwo(sz);
    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(sz);
    return A;
}

```

```

template<int prime, int gen>
vector<int> convolution(vector<int> A, vector<int> B){
    int sz = A.size() + B.size() - 1;
    int size = nearestPowerOfTwo(sz);
    A.resize(size), B.resize(size);
    ntt<prime, gen>(A, 1), ntt<prime, gen>(B, 1);
    for(int i = 0; i < size; i++)
        A[i] = (lli)A[i] * B[i] % prime;
    ntt<prime, gen>(A, -1);
    A.resize(sz);
    return A;
}

```

```
const int p = 7340033, g = 3; //default values for NTT
```

4.5. Aplicaciones

4.5.1. 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';
    X = convolution<p, g>(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();
}

```

4.5.2. Recíproco de un polinomio

```

vector<int> inversePolynomial(const vector<int> & A){
    vector<int> R(1, inverse(A[0], p));
    //R(x) = 2R(x)-A(x)R(x)^2
    while(R.size() < A.size()){
        int c = 2 * R.size();
        R.resize(c);
        vector<int> TR = R;
        TR.resize(2 * c);
        vector<int> TF(TR.size());
        for(int i = 0; i < c && i < A.size(); ++i)
            TF[i] = A[i];
        ntt<p, g>(TR, 1);
        ntt<p, g>(TF, 1);
        for(int i = 0; i < TR.size(); ++i)
            TR[i] = (1li)TR[i] * TR[i] % p * TF[i] % p;
        ntt<p, g>(TR, -1);
        for(int i = 0; i < c; ++i){
            R[i] = R[i] + R[i] - TR[i];
            if(R[i] < 0) R[i] += p;
            if(R[i] >= p) R[i] -= p;
        }
    }
    R.resize(A.size());
    return R;
}

```

4.5.3. Raíz cuadrada de un polinomio

```

const int inv2 = inverse(2, p);

vector<int> sqrtPolynomial(const vector<int> & A){
    int r0 = 1; //verify that r0^2 = A[0] mod p
    vector<int> R(1, r0);
    //R(x) = R(x)/2 + A(x)/(2R(x))
    while(R.size() < A.size()){
        int c = 2 * R.size();
        R.resize(c);
        vector<int> TF(c);
        for(int i = 0; i < c && i < A.size(); ++i)
            TF[i] = A[i];
        vector<int> IR = inversePolynomial(R);
        TF = convolution<p, g>(TF, IR);
        for(int i = 0; i < c; ++i){
            R[i] = R[i] + TF[i];
            if(R[i] >= p) R[i] -= p;
            R[i] = (1li)R[i] * inv2 % p;
        }
    }
    R.resize(A.size());
    return R;
}

```

4.5.4. Logaritmo y exponencial de un polinomio

```

vector<int> derivative(vector<int> A){
    for(int i = 0; i < A.size(); ++i)
        A[i] = (1li)A[i] * i % p;
    if(!A.empty()) A.erase(A.begin());
    return A;
}

vector<int> integral(vector<int> A){
    for(int i = 0; i < A.size(); ++i)
        A[i] = (1li)A[i] * (inverse(i+1, p)) % p;
    A.insert(A.begin(), 0);
    return A;
}

vector<int> logarithm(vector<int> A){

```

```

assert(A[0] == 1);
int n = A.size();
A = convolution<p, g>(derivative(A), inversePolynomial(A));
A.resize(n);
A = integral(A);
A.resize(n);
return A;
}

vector<int> exponential(const vector<int> & A){
    assert(A[0] == 0);
    //E(x) = E(x)(1-ln(E(x))+A(x))
    vector<int> E(1, 1);
    while(E.size() < A.size()){
        int c = 2*E.size();
        E.resize(c);
        vector<int> S = logarithm(E);
        for(int i = 0; i < c && i < A.size(); ++i){
            S[i] = A[i] - S[i];
            if(S[i] < 0) S[i] += p;
        }
        S[0] = 1;
        E = convolution<p, g>(E, S);
        E.resize(c);
    }
    E.resize(A.size());
    return E;
}

```

4.5.5. Cociente y residuo de dos polinomios

```

//returns Q(x), where A(x)=B(x)Q(x)+R(x)
vector<int> quotient(vector<int> A, vector<int> B){
    int n = A.size(), m = B.size();
    if(n < m) return vector<int>{0};
    reverse(A.begin(), A.end());
    reverse(B.begin(), B.end());
    A.resize(n-m+1), B.resize(n-m+1);
    A = convolution<p, g>(A, inversePolynomial(B));
    A.resize(n-m+1);
    reverse(A.begin(), A.end());
    return A;
}

```

```

//returns R(x), where A(x)=B(x)Q(x)+R(x)
vector<int> remainder(vector<int> A, const vector<int> & B){
    int n = A.size(), m = B.size();
    if(n >= m){
        vector<int> C = convolution<p, g>(quotient(A, B), B);
        A.resize(m-1);
        for(int i = 0; i < m-1; ++i){
            A[i] -= C[i];
            if(A[i] < 0) A[i] += p;
        }
    }
    return A;
}

```

4.5.6. Multievaluación rápida

```

//evaluates all the points in P(x), both the size of P and points
↪ must be the same
vector<int> multiEvaluate(const vector<int> & P, const vector<int>
↪ & points){
    int n = points.size();
    vector<vector<int>> prod(2*n - 1);
    function<void(int, int, int)> pre = [&](int v, int l, int r){
        if(l == r) prod[v] = vector<int>{(p - points[l]) % p, 1};
        else{
            int y = (l + r) / 2;
            int z = v + (y - l + 1) * 2;
            pre(v + 1, l, y);
            pre(z, y + 1, r);
            prod[v] = convolution<p, g>(prod[v + 1], prod[z]);
        }
    };
    pre(0, 0, n - 1);

    function<int(const vector<int>&, int)> eval = [&](const
↪ vector<int> & poly, int x0){
        int ans = 0;
        for(int i = (int)poly.size()-1; i >= 0; --i){
            ans = (lli)ans * x0 % p + poly[i];
            if(ans >= p) ans -= p;
        }
        return ans;
    };
}

```

```

};

vector<int> res(n);
function<void(int, int, int, vector<int>)> evaluate = [&](int v,
↪ int l, int r, vector<int> poly){
    poly = remainder(poly, prod[v]);
    if(poly.size() < 400){
        for(int i = 1; i <= r; ++i)
            res[i] = eval(poly, points[i]);
    }else{
        if(l == r)
            res[l] = poly[0];
        else{
            int y = (l + r) / 2;
            int z = v + (y - l + 1) * 2;
            evaluate(v + 1, l, y, poly);
            evaluate(z, y + 1, r, poly);
        }
    }
};
evaluate(0, 0, n - 1, P);
return res;
}

```

4.5.7. DFT con tamaño de vector arbitrario (algoritmo de Bluestein)

```

//it evaluates 1, w^2, w^4, ..., w^(2n-2) on the polynomial a(x)
//in this example we do a DFT with arbitrary size
vector<comp> bluestein(vector<comp> A){
    int n = A.size();
    int m = nearestPowerOfTwo(2*n-1);
    comp w = polar(1.0, PI / n), w1 = w, w2 = 1;
    vector<comp> p(m), q(m), b(n);
    for(int k = 0; k < n; ++k, w2 *= w1, w1 *= w*w){
        b[k] = w2;
        p[k] = A[k] * b[k];
        q[k] = (comp)1 / b[k];
        if(k) q[m-k] = q[k];
    }
    fft(p, 1), fft(q, 1);
    for(int i = 0; i < m; i++)
        p[i] *= q[i];
}

```

```

fft(p, -1);
for(int k = 0; k < n; ++k)
    A[k] = b[k] * p[k];
return A;
}

```

4.6. Convolución de dos vectores reales con solo dos FFT's

```

//A and B are real-valued vectors
//just do 2 fft's instead of 3
vector<comp> convolutionTrick(const vector<comp> & A, const
↪ vector<comp> & B){
    int sz = A.size() + B.size() - 1;
    int size = nearestPowerOfTwo(sz);
    vector<comp> C(size);
    comp I(0, 1);
    for(int i = 0; i < A.size() || i < B.size(); ++i){
        if(i < A.size()) C[i] += A[i];
        if(i < B.size()) C[i] += I*B[i];
    }
    fft(C, 1);
    vector<comp> D(size);
    for(int i = 0, j = 0; i < size; ++i){
        j = (size-1) & (size-i);
        D[i] = (conj(C[j]*C[j]) - C[i]*C[i]) * 0.25 * I;
    }
    fft(D, -1);
    D.resize(sz);
    return D;
}

```

4.7. Convolución con módulo arbitrario

```

//convolution with arbitrary modulo using only 4 fft's
vector<int> convolutionMod(const vector<int> & A, const
↪ vector<int> & B, int mod){
    int s = sqrt(mod);
    int sz = A.size() + B.size() - 1;
    int size = nearestPowerOfTwo(sz);
    vector<comp> a(size), b(size);
    for(int i = 0; i < A.size(); ++i)
        a[i] = comp(A[i] % s, A[i] / s);
}

```



```

for(int i = 0; i < B.size(); ++i)
    b[i] = comp(B[i] % s, B[i] / s);
fft(a, 1), fft(b, 1);
comp I(0, 1);
vector<comp> c(size), d(size);
for(int i = 0, j = 0; i < size; ++i){
    j = (size-1) & (size-i);
    comp e = (a[i] + conj(a[j])) * 0.5;
    comp f = (conj(a[j]) - a[i]) * 0.5 * I;
    comp g = (b[i] + conj(b[j])) * 0.5;
    comp h = (conj(b[j]) - b[i]) * 0.5 * I;
    c[i] = e * g + I * (e * h + f * g);
    d[i] = f * h;
}
fft(c, -1), fft(d, -1);
vector<int> D(sz);
for(int i = 0, j = 0; i < sz; ++i){
    j = (size-1) & (size-i);
    int p0 = (lli)round(real(c[i])) % mod;
    int p1 = (lli)round(imag(c[i])) % mod;
    int p2 = (lli)round(real(d[i])) % mod;
    D[i] = p0 + s*(p1 + (lli)p2*s % mod) % mod;
    if(D[i] >= mod) D[i] -= mod;
    if(D[i] < 0) D[i] += mod;
}
return D;
}

//convolution with arbitrary modulo using CRT
//slower but with no precision errors
const int a = 998244353, b = 985661441, c = 754974721;
const lli a_b = inverse(a, b), a_c = inverse(a, c), b_c =
    inverse(b, c);
vector<int> convolutionModCRT(const vector<int> & A, const
    vector<int> & B, int mod){
    vector<int> P = convolution<a, 3>(A, B);
    vector<int> Q = convolution<b, 3>(A, B);
    vector<int> R = convolution<c, 11>(A, B);
    vector<int> D(P.size());
    for(int i = 0; i < D.size(); ++i){
        int x1 = P[i] % a;
        if(x1 < 0) x1 += a;
        int x2 = a_b * (Q[i] - x1) % b;
        if(x2 < 0) x2 += b;

```

```

    int x3 = (a_c * (R[i] - x1) % c - x2) * b_c % c;
    if(x3 < 0) x3 += c;
    D[i] = x1 + a*(x2 + (lli)x3*b % mod) % mod;
    if(D[i] >= mod) D[i] -= mod;
    if(D[i] < 0) D[i] += mod;
}
return D;
}

```

4.8. Transformada rápida de Walsh–Hadamard

```

//Fast Walsh-Hadamard transform, works with any modulo p
//op: 0(OR), 1(AND), 2(XOR), A.size() must be power of 2
void fwt(vector<int> & A, int op, int inv){
    int n = A.size();
    for(int k = 1; k < n; k <= 1)
        for(int i = 0; i < n; i += k << 1)
            for(int j = 0; j < k; ++j){
                int u = A[i + j], v = A[i + j + k];
                int sum = u + v < p ? u + v : u + v - p;
                int rest = u - v < 0 ? u - v + p : u - v;
                if(inv == -1){
                    if(op == 0)
                        A[i + j + k] = rest ? p - rest : 0;
                    else if(op == 1)
                        A[i + j] = rest;
                    else if(op == 2)
                        A[i + j] = sum, A[i + j + k] = rest;
                }else{
                    if(op == 0)
                        A[i + j + k] = sum;
                    else if(op == 1)
                        A[i + j] = sum;
                    else if(op == 2)
                        A[i + j] = sum, A[i + j + k] = rest;
                }
            }
        }
    if(inv == -1 && op == 2){
        lli nrev = inverse(n, p);
        for(int i = 0; i < n; ++i)
            A[i] = A[i] * nrev % p;
    }
}

```

5. Geometría

5.1. Estructura point

```
ld eps = 1e-9, inf = numeric_limits<ld>::max();
```

```
bool geq(ld a, ld b){return a-b >= -eps;}    //a >= b
bool leq(ld a, ld b){return b-a >= -eps;}    //a <= b
bool ge(ld a, ld b){return a-b > eps;}       //a > b
bool le(ld a, ld b){return b-a > eps;}       //a < b
bool eq(ld a, ld b){return abs(a-b) <= eps;} //a == b
bool neq(ld a, ld b){return abs(a-b) > eps;} //a != b

struct point{
    ld x, y;
    point(): x(0), y(0){}
    point(ld x, ld y): x(x), y(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 ld & k) const{return point(x * k, y * k);}

    point operator/(const ld & 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 ld & p){*this = *this * p; return *this;}

    point operator/=(const ld & p){*this = *this / p; return *this;}

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

```
        return p + ((*this) - p).rotate(angle);
    }
    point perpendicular() const{
        return point(-y, x);
    }

    ld dot(const point & p) const{
        return x * p.x + y * p.y;
    }
    ld cross(const point & p) const{
        return x * p.y - y * p.x;
    }
    ld norm() const{
        return x * x + y * y;
    }
    long double length() const{
        return sqrtl(x * x + y * y);
    }

    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 eq(x, p.x) && eq(y, p.y);
    }
    bool operator!=(const point & p) const{
        return !(*this == p);
    }
    bool operator<(const point & p) const{
        if(eq(x, p.x)) return le(y, p.y);
        return le(x, p.x);
    }
    bool operator>(const point & p) const{
        if(eq(x, p.x)) return ge(y, p.y);
        return ge(x, p.x);
    }
}
```

```

};

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

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

int sgn(ld x){
    if(ge(x, 0)) return 1;
    if(le(x, 0)) return -1;
    return 0;
}

```

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 eq((p - a).cross(v), 0);
}

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
    ld det = v1.cross(v2);
    if(eq(det, 0)){
        if(eq((a2 - a1).cross(v1), 0)){

```

```

            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
    ld 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;
    ld det = v.cross(v2);
    if(eq(det, 0)){
        if(eq((c - a).cross(v), 0)){
            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
    }
}
return sgn(v2.cross(a - c)) != sgn(v2.cross(b - c)); //1:
    ↪ single point, 0: no point
}
}

```

5.2.5. Distancia punto-recta

```

ld 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

```

ld distancePointCircle(const point & p, const point & c, ld r){
    //point p, center c, radius r
    return max((ld)0, (p - c).length() - r);
}

```

5.3.2. Proyección punto exterior a círculo

```

point projectionPointCircle(const point & p, const point & c, ld
    ↪ 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, ld r){
    //point p (outside the circle), center c, radius r
    point v = (p - c).normalize() * r;
    ld 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, ld r){
    //line a+tv, center c, radius r
    ld A = v.dot(v);
    ld B = (a - c).dot(v);
    ld C = (a - c).dot(a - c) - r * r;
    ld D = B*B - A*C;
    if(eq(D, 0)) 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);
        ld t1 = (-B + D) / A;
        ld t2 = (-B - D) / A;
        return {a + v * t1, a + v * t2};
    }
}

```

5.3.5. Centro y radio a través de tres puntos

```

pair<point, ld> 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());
    ld r = (c - m).length();

```

```
    return {c, r};
}
```

5.3.6. Intersección de círculos

```
vector<point> intersectionCircles(const point & c1, ld r1, const
↪ point & c2, ld r2){
    //circle 1 with center c1 and radius r1
    //circle 2 with center c2 and radius r2
    ld A = 2*r1*(c2.y - c1.y);
    ld B = 2*r1*(c2.x - c1.x);
    ld C = (c1 - c2).dot(c1 - c2) + r1*r1 - r2*r2;
    ld D = A*A + B*B - C*C;
    if(eq(D, 0)) return {c1 + point(B, A) * r1 / C};
    else if(le(D, 0)) return {};
    else{
        D = sqrt(D);
        ld cos1 = (B*C + A*D) / (A*A + B*B);
        ld sin1 = (A*C - B*D) / (A*A + B*B);
        ld cos2 = (B*C - A*D) / (A*A + B*B);
        ld sin2 = (A*C + B*D) / (A*A + B*B);
        return {c1 + point(cos1, sin1) * r1, c1 + point(cos2, sin2) *
↪ r1};
    }
}
```

5.3.7. Contención de círculos

```
int circleInsideCircle(const point & c1, ld r1, const point & c2,
↪ ld 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
    ld l = r1 - r2 - (c1 - c2).length();
    return (ge(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
}
```

```
int circleOutsideCircle(const point & c1, ld r1, const point & c2,
↪ ld 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
```

```
    ld l = (c1 - c2).length() - (r1 + r2);
    return (ge(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
}
```

```
int pointInCircle(const point & c, ld r, const point & p){
    //test if point p is inside the circle with center c and radius
    ↪ r
    //returns "0" if it's outside, "-1" if it's in the perimeter,
    ↪ "1" if it's inside
    ld l = (p - c).length() - r;
    return (le(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
}
```

5.3.8. Tangentes

```
vector<vector<point>> commonExteriorTangents(const point & c1, ld
↪ r1, const point & c2, ld 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(eq(r1, r2))
            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, ld
↪ r1, const point & c2, ld 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}};
}
}

```

5.3.9. Smallest enclosing circle

```

pair<point, ld> mec2(vector<point> & S, const point & a, const
    ↪ point & b, int n){
    ld hi = inf, lo = -hi;
    for(int i = 0; i < n; ++i){
        ld si = (b - a).cross(S[i] - a);
        if(eq(si, 0)) continue;
        point m = getCircle(a, b, S[i]).first;
        ld cr = (b - a).cross(m - a);
        if(le(si, 0)) hi = min(hi, cr);
        else lo = max(lo, cr);
    }
    ld v = (ge(lo, 0) ? lo : le(hi, 0) ? hi : 0);
    point c = (a + b) / 2 + (b - a).perpendicular() * v / (b -
        ↪ a).norm();
    return {c, (a - c).norm()};
}

```

```

pair<point, ld> mec(vector<point> & S, const point & a, int n){
    random_shuffle(S.begin(), S.begin() + n);
    point b = S[0], c = (a + b) / 2;
    ld r = (a - c).norm();
    for(int i = 1; i < n; ++i){
        if(ge((S[i] - c).norm(), r)){
            tie(c, r) = (n == S.size() ? mec(S, S[i], i) : mec2(S, a,
                ↪ S[i], i));
        }
    }
    return {c, r};
}

```

```

pair<point, ld> smallestEnclosingCircle(vector<point> S){
    assert(!S.empty());

```

```

    auto r = mec(S, S[0], S.size());
    return {r.first, sqrt(r.second)};
}

```

5.4. Polígonos

5.4.1. Perímetro y área de un polígono

```

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

```

```

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

```

5.4.2. Envoltente 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 && leq((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 && leq((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. Verificar si un punto pertenece a un polígono convexo $O(\log n)$

```

//point in convex polygon in log(n)
//first do preprocess: seg=process(P),
//then for each query call pointInConvexPolygon(seg, p - P[0])
vector<point> process(vector<point> & P){
    int n = P.size();
    rotate(P.begin(), min_element(P.begin(), P.end()), P.end());
    vector<point> seg(n - 1);
    for(int i = 0; i < n - 1; ++i)
        seg[i] = P[i + 1] - P[0];
    return seg;
}

```

```

bool pointInConvexPolygon(vector<point> & seg, const point & p){
    int n = seg.size();
    if(neq(seg[0].cross(p), 0) && sgn(seg[0].cross(p)) !=
        ↪ sgn(seg[0].cross(seg[n - 1])))
        return false;
    if(neq(seg[n - 1].cross(p), 0) && sgn(seg[n - 1].cross(p)) !=
        ↪ sgn(seg[n - 1].cross(seg[0])))
        return false;
    if(eq(seg[0].cross(p), 0))
        return geq(seg[0].length(), p.length());
    int l = 0, r = n - 1;
    while(r - l > 1){
        int m = l + ((r - l) >> 1);
        if(geq(seg[m].cross(p), 0)) l = m;
        else r = m;
    }
    return eq(abs(seg[l].cross(seg[l + 1])), abs((p -
        ↪ seg[l]).cross(p - seg[l + 1])) + abs(p.cross(seg[l])) +
        ↪ abs(p.cross(seg[l + 1])));
}

```

5.4.6. Cortar un polígono con una recta

```

bool lineCutsPolygon(vector<point> & P, const point & a, const
    ↪ point & v){
    //line a+tv, polygon P
    int n = P.size();
    for(int i = 0, first = 0; i <= n; ++i){

```

```

    int side = sgn(v.cross(P[i%n]-a));
    if(!side) continue;
    if(!first) first = side;
    else if(side != first) return true;
}
return false;
}

vector<vector<point>> cutPolygon(vector<point> & P, const point &
↪ a, const point & v){
    //line a+tv, polygon P
    int n = P.size();
    if(!lineCutsPolygon(P, a, v)) return {P};
    int idx = 0;
    vector<vector<point>> ans(2);
    for(int i = 0; i < n; ++i){
        if(intersectLineSegmentInfo(a, v, P[i], P[(i+1)%n])){
            point p = intersectLines(a, v, P[i], P[(i+1)%n] - P[i]);
            if(P[i] == p) continue;
            ans[idx].push_back(P[i]);
            ans[1-idx].push_back(p);
            ans[idx].push_back(p);
            idx = 1-idx;
        }else{
            ans[idx].push_back(P[i]);
        }
    }
    return ans;
}

```

5.4.7. Centroide de un polígono

```

point centroid(vector<point> & P){
    point num;
    ld den = 0;
    int n = P.size();
    for(int i = 0; i < n; ++i){
        ld cross = P[i].cross(P[(i + 1) % n]);
        num += (P[i] + P[(i + 1) % n]) * cross;
        den += cross;
    }
    return num / (3 * den);
}

```

5.4.8. 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(ge(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 && ge(f(i, i+1, j+1), f(i, i+1, j)))
            ans.emplace_back(i, ++j);
    }
    return ans;
}

```

5.4.9. Diámetro y ancho

```

pair<ld, ld> 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]);};
    ld diameter = 0;
    ld width = inf;
    while(ge(dot(0, k), 0)) k = (k+1) % n;
    for(int i = 0; i < n; ++i){
        while(ge(cross(i, k), 0)) 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);
}

```

5.4.10. Smallest enclosing rectangle

```

pair<ld, ld> 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]);};
ld perimeter = inf, area = inf;
for(int i = 0, j = 0, k = 0, m = 0; i < n; ++i){
    while(ge(dot(i, j), 0)) j = (j+1) % n;
    if(!i) k = j;
    while(ge(cross(i, k), 0)) k = (k+1) % n;
    if(!i) m = k;
    while(le(dot(i, m), 0)) m = (m+1) % n;
    //pairs: (i, k) , (j, m)
    point v = P[(i+1)%n] - P[i];
    ld h = distancePointLine(P[i], v, P[k]);
    ld 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;
    ld ans = inf;
    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, -inf});
        auto upper = S.upper_bound({P[i].x + ans + eps, -inf});
        for(auto it = lower; it != upper; ++it){
            ld 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;
        ld th;
        node *l, *r;
    }*root;

    vector<pair<ld, 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).dot(p - aux[i].second);
        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<ld, node*>> que;

void k_nn(node *t, point p, int k){
    if(!t)
        return;
    ld d = (p - t->p).length();
    if(que.size() < k)
        que.push({ d, t });
    else if(ge(que.top().first, d)){
        que.pop();
        que.push({ d, t });
    }
    if(!t->l && !t->r)
        return;
    if(le(d, t->th)){
        k_nn(t->l, p, k);
        if(leq(t->th - d, que.top().first))
            k_nn(t->r, p, k);
    }else{
        k_nn(t->r, p, k);
        if(leq(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]);
        ld x = (A[(pa + 1) % na] - A[pa]).cross(B[(pb + 1) % nb] -
            ↪ B[pb]);
        if(leq(x, 0)) pb++;
        if(geq(x, 0)) pa++;
    }

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

    return M;
}

```

5.8. Triangulación de Delaunay

//Delaunay triangulation in $O(n \log n)$

```

const point inf_pt(inf, inf);

struct QuadEdge{
    point origin;
    QuadEdge* rot = nullptr;
    QuadEdge* onext = nullptr;
    bool used = false;
    QuadEdge* rev() const{return rot->rot;}
    QuadEdge* lnext() const{return rot->rev()->onext->rot;}
    QuadEdge* oprev() const{return rot->onext->rot;}
    point dest() const{return rev()->origin;}
};

```

```

QuadEdge* make_edge(const point & from, const point & to){
    QuadEdge* e1 = new QuadEdge;
    QuadEdge* e2 = new QuadEdge;
    QuadEdge* e3 = new QuadEdge;
    QuadEdge* e4 = new QuadEdge;
    e1->origin = from;
    e2->origin = to;
    e3->origin = e4->origin = inf_pt;
    e1->rot = e3;

```

```

    e2->rot = e4;
    e3->rot = e2;
    e4->rot = e1;
    e1->onext = e1;
    e2->onext = e2;
    e3->onext = e4;
    e4->onext = e3;
    return e1;
}

void splice(QuadEdge* a, QuadEdge* b){
    swap(a->onext->rot->onext, b->onext->rot->onext);
    swap(a->onext, b->onext);
}

void delete_edge(QuadEdge* e){
    splice(e, e->oprev());
    splice(e->rev(), e->rev()->oprev());
    delete e->rot;
    delete e->rev()->rot;
    delete e;
    delete e->rev();
}

QuadEdge* connect(QuadEdge* a, QuadEdge* b){
    QuadEdge* e = make_edge(a->dest(), b->origin);
    splice(e, a->lnext());
    splice(e->rev(), b);
    return e;
}

bool left_of(const point & p, QuadEdge* e){
    return ge((e->origin - p).cross(e->dest() - p), 0);
}

bool right_of(const point & p, QuadEdge* e){
    return le((e->origin - p).cross(e->dest() - p), 0);
}

ld det3(ld a1, ld a2, ld a3, ld b1, ld b2, ld b3, ld c1, ld c2, ld
↪ c3) {
    return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1 * b3) + a3
↪ * (b1 * c2 - c1 * b2);
}

```

```

bool in_circle(const point & a, const point & b, const point & c,
↪ const point & d) {
    ld det = -det3(b.x, b.y, b.norm(), c.x, c.y, c.norm(), d.x, d.y,
↪ d.norm());
    det += det3(a.x, a.y, a.norm(), c.x, c.y, c.norm(), d.x, d.y,
↪ d.norm());
    det -= det3(a.x, a.y, a.norm(), b.x, b.y, b.norm(), d.x, d.y,
↪ d.norm());
    det += det3(a.x, a.y, a.norm(), b.x, b.y, b.norm(), c.x, c.y,
↪ c.norm());
    return ge(det, 0);
}

pair<QuadEdge*, QuadEdge*> build_tr(int l, int r, vector<point> &
↪ P){
    if(r - l + 1 == 2){
        QuadEdge* res = make_edge(P[l], P[r]);
        return make_pair(res, res->rev());
    }
    if(r - l + 1 == 3){
        QuadEdge *a = make_edge(P[l], P[l + 1]), *b = make_edge(P[l +
↪ 1], P[r]);
        splice(a->rev(), b);
        int sg = sgn((P[l + 1] - P[l]).cross(P[r] - P[l]));
        if(sg == 0)
            return make_pair(a, b->rev());
        QuadEdge* c = connect(b, a);
        if(sg == 1)
            return make_pair(a, b->rev());
        else
            return make_pair(c->rev(), c);
    }
    int mid = (l + r) / 2;
    QuadEdge *ldo, *ldi, *rdo, *rdi;
    tie(ldo, ldi) = build_tr(l, mid, P);
    tie(rdi, rdo) = build_tr(mid + 1, r, P);
    while(true){
        if(left_of(rdi->origin, ldi)){
            ldi = ldi->lnext();
            continue;
        }
        if(right_of(ldi->origin, rdi)){
            rdi = rdi->rev()->onext;
        }
    }
}

```

```

    continue;
}
break;
}
QuadEdge* basel = connect(rdi->rev(), ldi);
auto valid = [&basel](QuadEdge* e){return right_of(e->dest(),
↪ basel);};
if(ldi->origin == ldo->origin)
    ldo = basel->rev();
if(rdi->origin == rdo->origin)
    rdo = basel;
while(true){
    QuadEdge* lcand = basel->rev()->onext;
    if(valid(lcand)){
        while(in_circle(basel->dest(), basel->origin, lcand->dest(),
↪ lcand->onext->dest())){
            QuadEdge* t = lcand->onext;
            delete_edge(lcand);
            lcand = t;
        }
    }
    QuadEdge* rcand = basel->oprev();
    if(valid(rcand)){
        while(in_circle(basel->dest(), basel->origin, rcand->dest(),
↪ rcand->oprev()->dest())){
            QuadEdge* t = rcand->oprev();
            delete_edge(rcand);
            rcand = t;
        }
    }
    if(!valid(lcand) && !valid(rcand))
        break;
    if(!valid(lcand) || (valid(rcand) && in_circle(lcand->dest(),
↪ lcand->origin, rcand->origin, rcand->dest())))
        basel = connect(rcand, basel->rev());
    else
        basel = connect(basel->rev(), lcand->rev());
}
return make_pair(ldo, rdo);
}

```

```

vector<tuple<point, point, point>> delaunay(vector<point> & P){
    sort(P.begin(), P.end());
    auto res = build_tr(0, (int)P.size() - 1, P);

```

```

QuadEdge* e = res.first;
vector<QuadEdge*> edges = {e};
while(1e((e->dest() - e->onext->dest()).cross(e->origin -
↪ e->onext->dest()), 0))
    e = e->onext;
auto add = [&P, &e, &edges]() {
    QuadEdge* curr = e;
    do{
        curr->used = true;
        P.push_back(curr->origin);
        edges.push_back(curr->rev());
        curr = curr->lnext();
    }while(curr != e);
};
add();
P.clear();
int kek = 0;
while(kek < (int)edges.size())
    if(!(e = edges[kek++])->used)
        add();
vector<tuple<point, point, point>> ans;
for(int i = 0; i < (int)P.size(); i += 3){
    ans.push_back(make_tuple(P[i], P[i + 1], P[i + 2]));
}
return ans;
}

```

6. Grafos

6.1. Disjoint Set

```

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. Definiciones

```

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;
    }
};

```

```

struct path{
    int cost = inf;
    deque<int> vertices;
    int size = 1;
    int prev = -1;
};

```

```

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 u = i;
        for(int j = 0; j < paths[i].size; j++){
            paths[i].vertices.push_front(u);
            u = paths[u].prev;
        }
    }
}

```

6.3. 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;
}

```

6.4. Dijkstra

```

vector<path> dijkstra(int start){
    priority_queue<edge, vector<edge>, greater<edge>> cola;

```

```

vector<path> paths(V);
cola.emplace(start, 0);
paths[start].cost = 0;
while(!cola.empty()){
    int u = cola.top().dest; cola.pop();
    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].prev = u;
            paths[v].size = paths[u].size + 1;
        }else if(nuevo < paths[v].cost){
            paths[v].prev = u;
            paths[v].size = paths[u].size + 1;
            cola.emplace(v, nuevo);
            paths[v].cost = nuevo;
        }
    }
}
buildPaths(paths);
return paths;
}

```

6.5. Bellman Ford

```

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].prev = 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].prev = u;
        paths[v].size = paths[u].size + 1;
        paths[v].cost = nuevo;
    }
}
}
buildPaths(paths);
return paths;
}

```

6.6. 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;
}

```

6.7. 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;
}

```

6.8. 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;
}

```

6.9. 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;
}

```

6.10. 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 {};
}

```

6.11. 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;
}

```

6.12. 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);
}

```


6.13. 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;
}
```

6.14. 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;
}
```

6.15. 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;
}

```

6.16. 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));
        }
    }
    return 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;
}

```

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

```

8.6. Hungariano

```

//Given a m*n cost matrix (m<=n), it finds a maximum cost
⇨ assignment.
//The actual assignment is in the vector returned.
//To find the minimum, negate the values.
template<typename T>
pair<T, vector<int>> hungarian(const vector<vector<T>> & a){
    int m = a.size(), n = a[0].size();
    assert(m <= n);
    vector<int> x(m, -1), y(n, -1);

```

```

vector<T> px(m, numeric_limits<T>::min()), py(n, 0);
for(int u = 0; u < m; ++u)
    for(int v = 0; v < n; ++v)
        px[u] = max(px[u], a[u][v]);
for(int u = 0, p, q; u < m; ){
    vector<int> s(m + 1, u), t(n, -1);
    for(p = q = 0; p <= q && x[u] < 0; ++p){
        for(int k = s[p], v = 0; v < n && x[u] < 0; ++v){
            if(px[k] + py[v] == a[k][v] && t[v] < 0){
                s[++q] = y[v], t[v] = k;
                if(s[q] < 0)
                    for(p = v; p >= 0; v = p)
                        y[v] = k = t[v], p = x[k], x[k] = v;
            }
        }
    }
    if(x[u] < 0){
        T delta = numeric_limits<T>::max();
        for(int i = 0; i <= q; ++i)
            for(int v = 0; v < n; ++v)
                if(t[v] < 0)
                    delta = min(delta, px[s[i]] + py[v] - a[s[i]][v]);
        for(int i = 0; i <= q; ++i)
            px[s[i]] -= delta;
        for(int v = 0; v < n; ++v)
            py[v] += (t[v] < 0 ? 0 : delta);
    }else{
        ++u;
    }
}
T cost = 0;
for(int u = 0; u < m; ++u)
    cost += a[u][x[u]];
return {cost, x};
}

```

9. Estructuras de datos

9.1. Segment Tree

9.1.1. Minimalistic: Point updates, range queries

```

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

    //build from an array in O(n)
    SegmentTree(int N, vector<T> & arr): N(N){
        ST.resize(N << 1);
        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];
    }

    //single element update in [l, r]
    void update(int l, int r, T value){
        l += N, r += N;
        for(int i = l; i <= r; ++i)
            ST[i] = value;
        l >>= 1, r >>= 1;
        while(l >= 1){
            for(int i = r; i >= l; --i)
                ST[i] = ST[i << 1] + ST[i << 1 | 1];
            l >>= 1, r >>= 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. Dynamic: Range updates and range queries

```

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

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

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

    T sum_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 sum;
    }
};

```

```

        else return left->sum_query(start, end) +
        ↪ right->sum_query(start, end);
    }

    void add_range(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->add_range(start, end, dif);
            right->add_range(start, end, dif);
            sum = left->sum + right->sum;
        }
    }

    void add_pos(int i, T sum){
        add_range(i, i, sum);
    }
};

```

9.1.3. Static: Range updates and range queries

```

template<typename T>
struct SegmentTreeEst{
    int size;
    vector<T> sum, lazy;

    void rec(int pos, int l, int r, vector<T> & arr){
        if(l == r) sum[pos] = arr[l];
        else{
            int half = l + ((r - l) >> 1);
            rec(2*pos+1, l, half, arr);
            rec(2*pos+2, half+1, r, arr);
            sum[pos] = sum[2*pos+1] + sum[2*pos+2];
        }
    }

    SegmentTreeEst(int n, vector<T> & arr): size(n){
        int h = ceil(log2(n));
        sum.resize((1 << (h + 1)) - 1);
    }
};

```

```

    lazy.resize((1 << (h + 1)) - 1);
    rec(0, 0, n - 1, arr);
}

void propagate(int pos, int l, int r, T dif){
    sum[pos] += (r - l + 1) * dif;
    if(l != r){
        lazy[2*pos+1] += dif;
        lazy[2*pos+2] += dif;
    }
}

T sum_query_rec(int start, int end, int pos, int l, int r){
    if(lazy[pos] != 0){
        propagate(pos, l, r, lazy[pos]);
        lazy[pos] = 0;
    }
    if(end < l || r < start) return 0;
    if(start <= l && r <= end) return sum[pos];
    else{
        int half = l + ((r - l) >> 1);
        return sum_query_rec(start, end, 2*pos+1, l, half) +
            sum_query_rec(start, end, 2*pos+2, half+1, r);
    }
}

T sum_query(int start, int end){
    return sum_query_rec(start, end, 0, 0, size - 1);
}

void add_range_rec(int start, int end, int pos, int l, int r, T
    ↪ dif){
    if(lazy[pos] != 0){
        propagate(pos, l, r, lazy[pos]);
        lazy[pos] = 0;
    }
    if(end < l || r < start) return;
    if(start <= l && r <= end) propagate(pos, l, r, dif);
    else{
        int half = l + ((r - l) >> 1);
        add_range_rec(start, end, 2*pos+1, l, half, dif);
        add_range_rec(start, end, 2*pos+2, half+1, r, dif);
        sum[pos] = sum[2*pos+1] + sum[2*pos+2];
    }
}

```

```

}

void add_range(int start, int end, T dif){
    add_range_rec(start, end, 0, 0, size - 1, dif);
}

void add_pos(int i, T sum){
    add_range(i, i, sum);
}

};

```

9.1.4. Persistent: Point updates, range queries

```

template<typename T>
struct StPer{
    StPer *left, *right;
    int l, r;
    T sum;

    StPer(int start, int end): left(NULL), right(NULL), l(start),
        ↪ r(end), sum(0){
        if(l != r){
            int half = l + ((r - l) >> 1);
            left = new StPer(l, half);
            right = new StPer(half+1, r);
        }
    }

    StPer(int start, int end, T val): left(NULL), right(NULL),
        ↪ l(start), r(end), sum(val){}

    StPer(int start, int end, StPer* left, StPer* right):
        ↪ left(left), right(right), l(start), r(end){
        sum = left->sum + right->sum;
    }

    T sum_query(int start, int end){
        if(end < l || r < start) return 0;
        if(start <= l && r <= end) return sum;
        else return left->sum_query(start, end) +
            ↪ right->sum_query(start, end);
    }

    StPer* update(int pos, T val){
        if(l == r) return new StPer(l, r, sum + val);
    }
}

```



```

    int half = 1 + ((r - 1) >> 1);
    if(pos <= half) return new StPer(l, r, left->update(pos, val),
        ↪ right);
    return new StPer(l, r, left, right->update(pos, val));
}
};

```

9.2. Fenwick Tree

```

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

    //build from array in O(n), indexed in 0
    FenwickTree(int N, vector<T> & arr): N(N){
        bit.resize(N);
        for(int i = 0; i < N; ++i){
            bit[i] += arr[i];
            if((i | (i + 1)) < N)
                bit[i | (i + 1)] += bit[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 * S + 1; i <= c_r * S - 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);
    }

    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;}

    inline int nodeHeight(AVLNode<T> *& pos){return pos ?
        pos->height : 0;}

    inline void update(AVLNode<T> *& pos){
        if(!pos) return;
        pos->height = 1 + max(nodeHeight(pos->left),
            nodeHeight(pos->right));
        pos->size = 1 + nodeSize(pos->left) + nodeSize(pos->right);
    }
}

```

```

}

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

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

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

void updateBalance(AVLNode<T> *& pos){
    if(!pos) return;
    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);
        update(pos), 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);
        }
    }
    update(pos), 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){
    assert(0 <= i && i < nodeSize(root));
    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);
    update(pos);
}

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

```

template<typename T>
struct TreapNode{
    TreapNode<T> *left, *right;
    T value;
    int key, size;

    //fields for queries
    bool rev;
    T sum, add;

    TreapNode(T value = 0): value(value), key(rand()), size(1),
        ↪ left(NULL), right(NULL), sum(value), add(0), rev(false){}
};

template<typename T>
struct Treap{
    TreapNode<T> *root;

    Treap(): root(NULL) {}

    inline int nodeSize(TreapNode<T>* t){return t ? t->size: 0;}

    inline T nodeSum(TreapNode<T>* t){return t ? t->sum : 0;}

    inline void update(TreapNode<T>* &t){
        if(!t) return;
        t->size = 1 + nodeSize(t->left) + nodeSize(t->right);
        t->sum = t->value; //reset node fields
        push(t->left), push(t->right); //push changes to child nodes
        t->sum = t->value + nodeSum(t->left) + nodeSum(t->right);
        ↪ //combine(left,t,t), combine(t,right,t)
    }

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

    void merge(TreapNode<T>* &t, TreapNode<T>* t1, TreapNode<T>*
        ↪ 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;
        update(t);
    }

    void split(TreapNode<T>* t, T & x, TreapNode<T>* &t1,
        ↪ TreapNode<T>* &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;
        update(t);
    }

    void insert(TreapNode<T>* &t, TreapNode<T>* 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);
        update(t);
    }

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

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

    void insert(T & x){insert(root, new TreapNode<T>(x));}

```

```

void erase(T & x){erase(root, x);}

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

T kth(int i){
    assert(0 <= i && i < nodeSize(root));
    TreapNode<T> *t = root;
    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;
}

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

//OPERATIONS FOR IMPLICIT TREAP
inline void push(TreapNode<T>* t){
    if(!t) return;
    //add in range example
    if(t->add){
        t->value += t->add;
        t->sum += t->add * nodeSize(t);
        if(t->left) t->left->add += t->add;
        if(t->right) t->right->add += t->add;

        t->add = 0;
    }
    //reverse range example
    if(t->rev){
        swap(t->left, t->right);
        if(t->left) t->left->rev ^= true;
        if(t->right) t->right->rev ^= true;
        t->rev = false;
    }
}

void split2(TreapNode<T>* t, int i, TreapNode<T>* &t1,
    ↪ TreapNode<T>* &t2){
    if(!t)
        return void(t1 = t2 = NULL);
    push(t);
    int curr = nodeSize(t->left);
    if(i <= curr)
        split2(t->left, i, t1, t->left), t2 = t;
    else
        split2(t->right, i - curr - 1, t->right, t2), t1 = t;
    update(t);
}

inline int aleatorio(){
    return (rand() << 15) + rand();
}

void merge2(TreapNode<T>* &t, TreapNode<T>* t1, TreapNode<T>*
    ↪ t2){
    push(t1), push(t2);
    if(!t1) t = t2;
    else if(!t2) t = t1;
    else if(aleatorio() % (nodeSize(t1) + nodeSize(t2)) <
        ↪ nodeSize(t1))
        merge2(t1->right, t1->right, t2), t = t1;
    else
        merge2(t2->left, t1, t2->left), t = t2;
    update(t);
}

//insert the element "x" at position "i"
void insert_at(T & x, int i){
    if(i > nodeSize(root)) return;

```

```

    TreapNode<T> *t1 = NULL, *t2 = NULL;
    split2(root, i, t1, t2);
    merge2(root, t1, new TreapNode<T>(x));
    merge2(root, root, t2);
}

//delete element at position "i"
void erase_at(int i){
    if(i >= nodeSize(root)) return;
    TreapNode<T> *t1 = NULL, *t2 = NULL, *t3 = NULL;
    split2(root, i, t1, t2);
    split2(t2, 1, t2, t3);
    merge2(root, t1, t3);
}

void update_at(TreapNode<T>* t, T & x, int i){
    push(t);
    assert(0 <= i && i < nodeSize(t));
    int curr = nodeSize(t->left);
    if(i == curr)
        t->value = x;
    else if(i < curr)
        update_at(t->left, x, i);
    else
        update_at(t->right, x, i - curr - 1);
    update(t);
}

T nth(TreapNode<T>* t, int i){
    push(t);
    assert(0 <= i && i < nodeSize(t));
    int curr = nodeSize(t->left);
    if(i == curr)
        return t->value;
    else if(i < curr)
        return nth(t->left, i);
    else
        return nth(t->right, i - curr - 1);
}

//update value of element at position "i" with "x"
void update_at(T & x, int i){update_at(root, x, i);}

//ith element

```

```

T nth(int i){return nth(root, i);}

//add "val" in [l, r]
void add_update(T & val, int l, int r){
    TreapNode<T> *t1 = NULL, *t2 = NULL, *t3 = NULL;
    split2(root, l, t1, t2);
    split2(t2, r - l + 1, t2, t3);
    t2->add += val;
    merge2(root, t1, t2);
    merge2(root, root, t3);
}

//reverse [l, r]
void reverse_update(int l, int r){
    TreapNode<T> *t1 = NULL, *t2 = NULL, *t3 = NULL;
    split2(root, l, t1, t2);
    split2(t2, r - l + 1, t2, t3);
    t2->rev ^= true;
    merge2(root, t1, t2);
    merge2(root, root, t3);
}

//rotate [l, r] k times to the right
void rotate_update(int k, int l, int r){
    TreapNode<T> *t1 = NULL, *t2 = NULL, *t3 = NULL, *t4 = NULL;
    split2(root, l, t1, t2);
    split2(t2, r - l + 1, t2, t3);
    k %= nodeSize(t2);
    split2(t2, nodeSize(t2) - k, t2, t4);
    merge2(root, t1, t4);
    merge2(root, root, t2);
    merge2(root, root, t3);
}

//sum query in [l, r]
T sum_query(int l, int r){
    TreapNode<T> *t1 = NULL, *t2 = NULL, *t3 = NULL;
    split2(root, l, t1, t2);
    split2(t2, r - l + 1, t2, t3);
    T ans = nodeSum(t2);
    merge2(root, t1, t2);
    merge2(root, root, t3);
    return ans;
}

```

```

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

void inorder(){inorder(root);}
};

```

9.6. Sparse table

9.6.1. Normal

```

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))]); //put the function accordingly
    }

    T sum(int l, int r){ //non-idempotent functions
        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;
    }
};

```

```

    }
}

return ans;
}

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

```

9.6.2. Disjoint

```

//build on O(n log n), queries in O(1) for any operation
template<typename T>
struct DisjointSparseTable{
    vector<vector<T>> left, right;
    int K, N;

    DisjointSparseTable(vector<T> & arr){
        N = arr.size();
        K = log2(N) + 2;
        left.assign(K + 1, vector<T>(N));
        right.assign(K + 1, vector<T>(N));
        for(int j = 0; (1 << j) <= N; ++j){
            int mask = (1 << j) - 1;
            T acum = 0; //neutral element of your operation
            for(int i = 0; i < N; ++i){
                acum += arr[i]; //your operation
                left[j][i] = acum;
                if((i & mask) == mask) acum = 0; //neutral element of your
                ↪ operation
            }
            acum = 0; //neutral element of your operation
            for(int i = N-1; i >= 0; --i){
                acum += arr[i]; //your operation
                right[j][i] = acum;
                if((i & mask) == 0) acum = 0; //neutral element of your
                ↪ operation
            }
        }
    }
};

```



```

T query(int l, int r){
    if(l == r) return left[0][l];
    int i = 31 - __builtin_clz(l^r);
    return left[i][r] + right[i][l]; //your operation
}
};

```

9.7. Wavelet Tree

```

struct WaveletTree{
    int lo, hi;
    WaveletTree *left, *right;
    vector<int> freq;
    vector<int> pref; //just use this if you want sums

    //queries indexed in base 1, complexity for all queries:
    ↪ O(log(max_element))
    //build from [from, to) with non-negative values in range [x, y]
    //you can use vector iterators or array pointers
    WaveletTree(vector<int>::iterator from, vector<int>::iterator
    ↪ to, int x, int y): lo(x), hi(y){
        if(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);
        pref.reserve(to - from + 1);
        pref.push_back(0);
        for(auto it = from; it != to; ++it){
            freq.push_back(freq.back() + f(*it));
            pref.push_back(pref.back() + *it);
        }
        if(hi != lo){
            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);
    }

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

```

9.8. Ordered Set C++

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template<typename T>

```

```
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
↪ tree_order_statistics_node_update>;
```

```
int main(){
    int t, n, m;
    ordered_set<int> 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(m);
            }
        }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;
}
```

9.9. Splay Tree

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> lengths;
    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++;
    lengths.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;
}

```

11.8. Contar número de unos en binario en un rango

```
//count the number of 1's in the i-th bit of all
//representations in binary of numbers in [1,n]
lli count(lli n, int i){
    if(n <= 0) return 0ll;
    lli ans = ((n + 1) >> (i + 1)) << i;
    ans += max(((n + 1) & ((1ll << (i + 1)) - 1)) - (1ll << i),
        ↪ 0ll);
    return ans;
}
```

11.9. Números aleatorios en C++11

```
//Random number generation in C++11
mt19937_64
↪ rng(chrono::steady_clock::now().time_since_epoch().count());

//Generate a random integer in [a, b], you can also use long long
↪ int
int aleatorio_int(int a, int b){
    uniform_int_distribution<int> dist(a, b);
    return dist(rng);
}

//Generate a random double in [a, b], you can also use long double
double aleatorio_double(double a, double b){
    uniform_real_distribution<double> dist(a, b);
    return dist(rng);
}
```

12. Fórmulas y notas

12.1. Números de Stirling del primer tipo

$\begin{bmatrix} n \\ k \end{bmatrix}$ representa el número de permutaciones de n elementos en exactamente k ciclos disjuntos.

$$\begin{aligned} \begin{bmatrix} 0 \\ 0 \end{bmatrix} &= 1 \\ \begin{bmatrix} 0 \\ n \end{bmatrix} &= \begin{bmatrix} n \\ 0 \end{bmatrix} = 0, & \quad n > 0 \\ \begin{bmatrix} n \\ k \end{bmatrix} &= (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}, & \quad k > 0 \\ \sum_{k=0}^n \begin{bmatrix} n \\ k \end{bmatrix} &= n! \\ \sum_{k=0}^{\infty} \begin{bmatrix} n \\ k \end{bmatrix} x^k &= \prod_{k=0}^{n-1} (x+k) \end{aligned}$$

12.2. Números de Stirling del segundo tipo

$\{n \atop k\}$ representa el número de formas de particionar un conjunto de n objetos distinguibles en k subconjuntos no vacíos.

$$\begin{aligned} \{0 \atop 0\} &= 1 \\ \{0 \atop n\} &= \{n \atop 0\} = 0, & \quad n > 0 \\ \{n \atop k\} &= k \{n-1 \atop k\} + \{n-1 \atop k-1\}, & \quad k > 0 \\ &= \sum_{j=0}^k \frac{j^n}{j!} \cdot \frac{(-1)^{k-j}}{(k-j)!} \end{aligned}$$

12.3. Números de Euler

$\langle n \atop k \rangle$ representa el número de permutaciones de 1 a n en donde exactamente k números son mayores que el número anterior, es decir, cuántas

permutaciones tienen k “ascensos”.

$$\begin{aligned}\langle 1 \rangle_0 &= 1 \\ \langle n \rangle_k &= (n-k) \langle n-1 \rangle_{k-1} + (k+1) \langle n-1 \rangle_k, \quad n \geq 2 \\ &= \sum_{j=0}^k (-1)^j \binom{n+1}{j} (k+1-j)^n \\ \sum_{k=0}^{n-1} \langle n \rangle_k &= n!\end{aligned}$$

12.4. Números de Catalan

$$\begin{aligned}C_0 &= 1 \\ C_n &= \frac{1}{n+1} \binom{2n}{n} = \sum_{j=0}^{n-1} C_j C_{n-1-j} \\ \sum_{n=0}^{\infty} C_n x^n &= \frac{1 - \sqrt{1-4x}}{2x}\end{aligned}$$

12.5. Números de Bell

B_n representa el número de formas de particionar un conjunto de n elementos.

$$\begin{aligned}B_n &= \sum_{k=0}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\} = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k \\ \sum_{k=0}^{\infty} \frac{B_k}{k!} x^k &= e^{e^x - 1}\end{aligned}$$

12.6. Números de Bernoulli

$$\begin{aligned}B_0^+ &= 1 \\ B_n^+ &= 1 - \sum_{k=0}^{n-1} \binom{n}{k} \frac{B_k^+}{n-k+1} \\ \sum_{m=0}^{\infty} \frac{B_m^+ x^m}{m!} &= \frac{x}{1-e^{-x}} = \frac{1}{\frac{1}{1!} - \frac{x}{2!} + \frac{x^2}{3!} - \frac{x^3}{4!} + \dots}\end{aligned}$$

12.7. Fórmula de Faulhaber

$$S_p(n) = \sum_{k=1}^n k^p = \frac{1}{p+1} \sum_{k=0}^p \binom{p+1}{k} B_k^+ n^{p+1-k}$$

12.8. Función Beta

$$\begin{aligned}B(x, y) &= \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} = 2 \int_0^{\pi/2} \sin^{2x-1}(\theta) \cos^{2y-1}(\theta) d\theta \\ &= \int_0^1 t^{x-1} (1-t)^{y-1} dt = \int_0^{\infty} \frac{t^{x-1}}{(1+t)^{x+y}} dt\end{aligned}$$

12.9. Función zeta de Riemann

La siguiente fórmula converge rápido para valores pequeños de n ($n \approx 20$):

$$\begin{aligned}\zeta(s) &\approx \frac{1}{d_0(1-2^{1-s})} \sum_{k=1}^n \frac{(-1)^{k-1} d_k}{k^s} \\ d_k &= \sum_{j=k}^n \frac{4^j}{n+j} \binom{n+j}{2j}\end{aligned}$$

12.10. Funciones generadoras

$$\sum_{n=0}^{\infty} \left(\sum_{k=0}^n a_k \right) x^n = \frac{1}{1-x} \sum_{n=0}^{\infty} a_n x^n$$

$$\sum_{n=0}^{\infty} \binom{n+k-1}{k-1} x^n = \frac{1}{(1-x)^k}$$

$$\sum_{n=0}^{\infty} p_n x^n = \frac{1}{\prod_{k=1}^{\infty} (1-x^k)} = \frac{1}{\sum_{n=-\infty}^{\infty} x^{\frac{1}{2}n(3n+1)}}$$

$$\sum_{n=0}^{\infty} n^k x^n = \frac{\sum_{i=0}^{k-1} \langle k \rangle_i x^{i+1}}{(1-x)^{k+1}}, \quad k \geq 1$$

12.11. Números armónicos

$$H_n = \sum_{k=1}^n \frac{1}{k} \approx \ln(n) + \gamma + \frac{1}{2n} - \frac{1}{12n^2}$$

$$\gamma \approx 0.577215664901532860606512$$

12.12. Aproximación de Stirling

$$\ln(n!) \approx n \ln(n) - n + \frac{1}{2} \ln(2\pi n)$$

$$\# \text{ de dígitos de } n! = 1 + \left\lfloor n \log \left(\frac{n}{e} \right) + \frac{1}{2} \log(2\pi n) \right\rfloor \quad (n \geq 30)$$

12.13. Ternas pitagóricas

- Una terna de enteros positivos (a, b, c) es pitagórica si $a^2 + b^2 = c^2$. Además es primitiva si $\gcd(a, b, c) = 1$.

- Generador de ternas primitivas:

$$a = m^2 - n^2$$

$$b = 2mn$$

$$c = m^2 + n^2$$

donde $n \geq 1$, $m > n$, $\gcd(m, n) = 1$ y m, n tienen distinta paridad.

- Árbol de ternas pitagóricas primitivas: al multiplicar cualquiera de estas matrices:

$$\begin{pmatrix} 1 & -2 & 2 \\ 2 & -1 & 2 \\ 2 & -2 & 3 \end{pmatrix}, \quad \begin{pmatrix} -1 & 2 & 2 \\ -2 & 1 & 2 \\ -2 & 2 & 3 \end{pmatrix}, \quad \begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 3 \end{pmatrix}$$

por una terna primitiva \mathbf{v}^T , obtenemos otra terna primitiva diferente. En particular, si empezamos con $\mathbf{v} = (3, 4, 5)$, podremos generar todas las ternas primitivas.

12.14. Árbol de Stern–Brocot

Todos los racionales positivos se pueden representar como un árbol binario de búsqueda completo infinito con raíz $\frac{1}{1}$.

- Dado un racional $q = [a_0; a_1, a_2, \dots, a_k]$ donde $a_k \neq 1$, sus hijos serán $[a_0; a_1, a_2, \dots, a_k + 1]$ y $[a_0; a_1, a_2, \dots, a_k - 1, 2]$, y su padre será $[a_0; a_1, a_2, \dots, a_k - 1]$.
- Para hallar el camino de la raíz $\frac{1}{1}$ a un racional q , se usa búsqueda binaria iniciando con $L = \frac{0}{1}$ y $R = \frac{1}{0}$. Para hallar M se supone que $L = \frac{a}{b}$ y $R = \frac{c}{d}$, entonces $M = \frac{a+c}{b+d}$.

12.15. Combinatoria

- Principio de las casillas: al colocar n objetos en k lugares hay al menos $\lceil \frac{n}{k} \rceil$ objetos en un mismo lugar.
- Número de funciones: sean A y B conjuntos con $m = |A|$ y $n = |B|$. Sea $f : A \rightarrow B$:

- Si $m \leq n$, entonces hay $m! \binom{n}{m}$ funciones inyectivas f .
 - Si $m = n$, entonces hay $n!$ funciones biyectivas f .
 - Si $m \geq n$, entonces hay $n! \left\{ \begin{smallmatrix} m \\ n \end{smallmatrix} \right\}$ funciones suprayectivas f .
- Barras y estrellas: ¿cuántas soluciones en los enteros no negativos tiene la ecuación $\sum_{i=1}^k x_i = n$? Tiene $\binom{n+k-1}{k-1}$ soluciones.
- ¿Cuántas soluciones en los enteros positivos tiene la ecuación $\sum_{i=1}^k x_i = n$? Tiene $\binom{n-1}{k-1}$ soluciones.
- Desordenamientos: $a_0 = 1$, $a_1 = 0$, $a_n = (n-1)(a_{n-1} + a_{n-2}) = na_{n-1} + (-1)^n$.
- Sea $f(x)$ una función. Sea $g_n(x) = xg'_{n-1}(x)$ con $g_0(x) = f(x)$. Entonces $g_n(x) = \sum_{k=0}^n \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} x^k f^{(k)}(x)$.
- Supongamos que tenemos $m+1$ puntos: $(0, y_0), (1, y_1), \dots, (m, y_m)$. Entonces el polinomio $P(x)$ de grado m que pasa por todos ellos es:

$$P(x) = \left[\prod_{i=0}^m (x-i) \right] (-1)^m \sum_{i=0}^m \frac{y_i (-1)^i}{(x-i)i!(m-i)!}$$

- Sea a_0, a_1, \dots una recurrencia lineal homogénea de grado d dada por $a_n = \sum_{i=1}^d b_i a_{n-i}$ para $n \geq d$ con términos iniciales a_0, a_1, \dots, a_{d-1} . Sean $A(x)$ y $B(x)$ las funciones generadoras de las sucesiones a_n y b_n respectivamente, entonces se cumple que $A(x) = \frac{A_0(x)}{1-B(x)}$, donde
- $$A_0(x) = \sum_{i=0}^{d-1} \left[a_i - \sum_{j=0}^{i-1} a_j b_{i-j} \right] x^i.$$
- Si queremos obtener otra recurrencia c_n tal que $c_n = a_{kn}$, las raíces del polinomio característico de c_n se obtienen al elevar todas las raíces del polinomio característico de a_n a la k -ésima potencia; y sus términos iniciales serán $a_0, a_k, \dots, a_{k(d-1)}$.

12.16. Grafos

- Sea d_n el número de grafos con n vértices etiquetados: $d_n = 2^{\binom{n}{2}}$.
- Sea c_n el número de grafos conexos con n vértices etiquetados. Tenemos la recurrencia: $c_1 = 1$ y $d_n = \sum_{k=1}^n \binom{n-1}{k-1} c_k d_{n-k}$. También se cumple, usando funciones generadoras exponenciales, que $C(x) = 1 + \ln(D(x))$.
- Sea t_n el número de torneos fuertemente conexos en n nodos etiquetados. Tenemos la recurrencia $t_1 = 1$ y $d_n = \sum_{k=1}^n \binom{n}{k} t_k d_{n-k}$. Usando funciones generadoras exponenciales, tenemos que $T(x) = 1 - \frac{1}{D(x)}$.
- Número de spanning trees en un grafo completo con n vértices etiquetados: n^{n-2} .
- Número de bosques etiquetados con n vértices y k componentes conexas: kn^{n-k-1} .
- Para un grafo no dirigido simple G con n vértices etiquetados de 1 a n , sea $Q = D - A$, donde D es la matriz diagonal de los grados de cada nodo de G y A es la matriz de adyacencia de G . Entonces el número de spanning trees de G es igual a cualquier cofactor de Q .

12.17. Teoría de números

$$(f * e)(n) = f(n)$$

$$(\varphi * \mathbf{1})(n) = n$$

$$(\mu * \mathbf{1})(n) = e(n)$$

$$\varphi(n^k) = n^{k-1}\varphi(n)$$

$$\sum_{\substack{k=1 \\ \gcd(k,n)=1}}^n k = \frac{n\varphi(n)}{2} \quad , \quad n \geq 2$$

$$\sum_{k=1}^n \text{lcm}(k, n) = \frac{n}{2} + \frac{n}{2} \sum_{d|n} d\varphi(d) = \frac{n}{2} + \frac{n}{2} \prod_{p^a|n} \frac{p^{2a+1} + 1}{p + 1}$$

$$\sum_{k=1}^n \gcd(k, n) = \sum_{d|n} d\varphi\left(\frac{n}{d}\right) = \prod_{p^a|n} p^{a-1}(1 + (a+1)(p-1))$$

- Suma de dos cuadrados: sea $\chi_4(n)$ una función multiplicativa igual a 1 si $n \equiv 1 \pmod{4}$, -1 si $n \equiv 3 \pmod{4}$ y cero en otro caso. Entonces, el número de soluciones enteras (a, b) de la ecuación $a^2 + b^2 = n$ es $4(\chi_4 * \mathbf{1})(n) = 4 \sum_{d|n} \chi_4(d)$.

- Teorema de Lucas:

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{k_i} \pmod{p}$$

$$m = \sum_{i=0}^k m_i p^i \quad , \quad n = \sum_{i=0}^k n_i p^i$$

$$0 \leq m_i, n_i < p$$

- Sean $a, b, c \in \mathbb{Z}$ con $a \neq 0$ y $b \neq 0$. La ecuación $ax + by = c$ tiene como soluciones:

$$x = \frac{x_0 c - b k}{d}$$

$$y = \frac{y_0 c + a k}{d}$$

para toda $k \in \mathbb{Z}$ si y solo si $d|c$, donde $ax_0 + by_0 = \gcd(a, b) = d$ (Euclides extendido). Si a y b tienen el mismo signo, hay exactamente $\max\left(\left\lfloor \frac{x_0 c}{|b|} \right\rfloor + \left\lfloor \frac{y_0 c}{|a|} \right\rfloor + 1, 0\right)$ soluciones no negativas. Si tienen el signo distinto, hay infinitas soluciones no negativas.

- Dada una función aritmética f con $f(1) \neq 0$, existe otra función aritmética g tal que $(f * g)(n) = e(n)$, dada por:

$$g(1) = \frac{1}{f(1)}$$

$$g(n) = -\frac{1}{f(1)} \sum_{d|n, d < n} f\left(\frac{n}{d}\right) g(d) \quad , \quad n > 1$$

- Sean $h(n) = \sum_{k=1}^n f\left(\left\lfloor \frac{n}{k} \right\rfloor\right) g(k)$, $G(n) = \sum_{k=1}^n g(k)$ y $m = \lfloor \sqrt{n} \rfloor$, entonces:

$$h(n) = \sum_{k=1}^{\lfloor n/m \rfloor} f\left(\left\lfloor \frac{n}{k} \right\rfloor\right) g(k) + \sum_{k=1}^{m-1} \left(G\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - G\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right) \right) f(k)$$

- Sean $F(n) = \sum_{k=1}^n f(k)$, $G(n) = \sum_{k=1}^n g(k)$, $h(n) = (f * g)(n) = \sum_{d|n} f(d)g\left(\frac{n}{d}\right)$ y $H(n) = \sum_{k=1}^n h(k)$, entonces:

$$H(n) = \sum_{k=1}^n f(k)G\left(\left\lfloor \frac{n}{k} \right\rfloor\right)$$

- Sean $\Phi_p(n) = \sum_{k=1}^n k^p \varphi(k)$ y $M_p(n) = \sum_{k=1}^n k^p \mu(k)$. Aplicando lo anterior, podemos calcular $\Phi_p(n)$ y $M_p(n)$ con complejidad $O(n^{2/3})$ si precalculamos con fuerza bruta los primeros $\lfloor n^{2/3} \rfloor$ valores, y para

los demás, usamos las siguientes recurrencias (DP con `map`):

producto de algunas potencias de dos y distintos primos de Fermat.

$$\Phi_p(n) = S_{p+1}(n) - \sum_{k=2}^{\lfloor n/m \rfloor} k^p \Phi_p\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - \sum_{k=1}^{m-1} \left(S_p\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - S_p\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right) \right) \Phi_p(k)$$

$$M_p(n) = 1 - \sum_{k=2}^{\lfloor n/m \rfloor} k^p M_p\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - \sum_{k=1}^{m-1} \left(S_p\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - S_p\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right) \right) M_p(k)$$

- En general, si queremos hallar $F(n)$ y existe una función mágica $g(n)$ tal que $G(n)$ y $H(n)$ se puedan calcular en $O(1)$, entonces:

$$F(n) = \frac{1}{g(1)} \left[H(n) - \sum_{k=2}^{\lfloor n/m \rfloor} g(k) F\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - \sum_{k=1}^{m-1} \left(G\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - G\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right) \right) F(k) \right]$$

12.18. Primos

$10^2 + 1, 10^3 + 9, 10^4 + 7, 10^5 + 3, 10^6 + 3, 10^7 + 19, 10^8 + 7, 10^9 + 7, 10^{10} + 19, 10^{11} + 3, 10^{12} + 39, 10^{13} + 37, 10^{14} + 31, 10^{15} + 37, 10^{16} + 61, 10^{17} + 3, 10^{18} + 3.$

$10^2 - 3, 10^3 - 3, 10^4 - 27, 10^5 - 9, 10^6 - 17, 10^7 - 9, 10^8 - 11, 10^9 - 63, 10^{10} - 33, 10^{11} - 23, 10^{12} - 11, 10^{13} - 29, 10^{14} - 27, 10^{15} - 11, 10^{16} - 63, 10^{17} - 3, 10^{18} - 11.$

12.19. Números primos de Mersenne

Números primos de la forma $M_p = 2^p - 1$ con p primo. Todos los números perfectos pares son de la forma $2^{p-1}M_p$ y viceversa.

Los primeros 47 valores de p son: 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917, 20996011, 24036583, 25964951, 30402457, 32582657, 37156667, 42643801, 43112609.

12.20. Números primos de Fermat

Números primos de la forma $F_p = 2^{2^p} + 1$, solo se conocen cinco: 3, 5, 17, 257, 65537. Un polígono de n lados es construible si y solo si n es el