Competitive Programming Handbook

Autor: Felipe Rafael Focil Mendoza

Version 1.3.2

Fecha 10/05/2024

Respuestas del problema

Existen ciertos mensajes que regresara el juez.

Compilation Error (CE)

El código que enviaste no compila

Run Time Error (RTE)

El código que enviaste tiene algún error de lógica.

La principal causa es que estas tratando de acceder a un espacio de memoria que no existe.

Time Limit Exceeded (TLE)

Tu código es correcto, pero es demasiado lento, por lo cual deberás buscar una forma de reducir el tiempo de complejidad.

Memory Limit Exceeded (MLE)

El código que enviaste ocupa demasiada memoria

Wrong Answer (WA)

Tu código está mal.

Output Limit Exceeded (OLE)

Imprimiste más cosas de las esperadas.

Presentation Error

Tu código esta bien, pero lo imprimiste en el formato incorrecto

Accepted (AC)

¡Felicidades!!!!

¡Tú respuesta es correcta!!!

Levanta tu mano y pide al staff que te den un globo.

Comandos de Compilación desde Terminal

Considere X como la letra del problema, se recomienda nombrar el archivo como $X.cpp$,por ejemplo si se esta resolviendo el problema B, el nombre del archivo será $B.cpp$.

Linux

Compilar

```
g++ -o X X.cpp
```

Ejecutar

```
./X
```

Windows

Compilar

```
g++ -o X.exe X.cpp
```

Ejecutar

```
X.exe
```

Mac OS X

Compilar

```
g++ -o X X.cpp
```

Ejecutar

```
./X
```

Plantilla de código inicial

```cpp
#include <bits/stdc++.h>

#define FIN ios::ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define endl '\n'
#define forn(i,a,b) for(int i = int(a); i < int(b);i++)
#define pb push_back
#define all(v) v.begin(),v.end()
#define sz(a) int(a.size())
#define coutBool(a) cout << ((a) ? "YES" : "NO")<<endl
#define mp make_pair
#define fi first
#define se second
#define debug(x) cout<<#x<<" : "<<x<<endl

using namespace std;

using ll = long long;
using ull = unsigned long long;
using vi = vector<int>;
using vvi = vector<vi>;
using vl = vector<ll>;
using vvl = vector<vl>;
using vb = vector<bool>;
using vs = vector<string>;
using pi = pair<int, int>;
using u64 = uint64_t;

void _main() {
    //FIX ME
}

int main() {
    FIN;

    int t = 1;
    cin >> t; // COMMENT THIS ONLY HAS ONE CASE
    while (t--) {
        _main();
    }

    return 0;
}
```

Tamaños de variable

Tipo	Base 2	Base 10	Base 10
bool	0 a 1	0 a 1	1 * 10^0
char	-128 a 127	-128 a 127	2 * 10^2

Tipo	Base 2	Base 10	Base 10
`unsigned char`	0 a 255	0 a 255	2.55 * 10^2
`short`	-32,768 a 32,767	-32,768 a 32,767	3.28 * 10^4
`int`	-2^31 a (2^31 - 1)	-2,147,483,648 a 2,147,483,647	2.15 * 10^9
`long long`	-2^63 a (2^63 - 1)	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	9.22 * 10^18
`unsigned long long`	0 a (2^64 - 1)	0 a 18,446,744,073,709,551,615	1.84 * 10^19
`uint64_t` (Mayor precisión)	0 a (2^64 - 1)	0 a 18,446,744,073,709,551,615	1.84 * 10^19

Tipo	Dígitos de precisión
`float`	7
`double`	15
`long double`	19

Tiempo de Complejidad

n	Possible complexities
n ≤ 10	O(n!), O(n^7), O(n^6)
n ≤ 20	O(2^n * n), O(n^5)
n ≤ 80	O(n^4)
n ≤ 400	O(n^3)
n ≤ 7500	O(n^2)
n ≤ 7 * 10^4	O(n√n)
n ≤ 5 * 10^5	O(n log n)
n ≤ 5 * 10^6	O(n)
n ≤ 10^18	O(log^2 n), O(log n), O(1)

End of File

```
int x;
while(cin>>x){
}
```

Precision de decimales

```
double pi = 3.14159265358979323846;

cout << fixed << setprecision(5) << pi << endl;
```

Exponenciacion Binaria

La operación a^b en la gran mayoria de librerias se implementa en $O(b)$, sin embargo si expersa b en base 2 (binario) se puede realizar en $O(logb)$.

```
int binpow(int a, int b) {
    int r = 1;;
    while (b > 0) {
        if (b & 1)  r*= a;
        a *=a;
        b >>= 1;
    }
    return r;
}
```

Aritmética Modular

Cuando los casos de entrada son demasiados grandes, usualmente el problema te solicita que apliques aritmética modular sobre un número p, que generalmente $10^9 + 7$ (1000000007).

Por ejemplo, supongamos que tenemos la formula $a \times b \times c$; Realizar la siguiente formula, sería incorrecto.

```
r =  (a*b*c) % 1000000007;
```

Debido a que es una multiplicación, la cantidad de dígitos aumenta de forma que no se pueden almacenar, a pesar de que matemáticamente hablando sea correcto.

Operaciones mediante Aritemtica Modular

```
using u64 = uint64_t;

const u64 MOD = 1e9 + 7;

#define ADD(a, b) (((a % MOD) + (b % MOD)) % MOD)
#define SUBTRACT(a, b) (((a) - (b) + MOD) % MOD)
#define MULTIPLY(a, b) (((a) % MOD) * ((b) % MOD)) % MOD
#define DIVIDE(a, b) (MULTIPLY(a,binpow(b,MOD-2)))

u64 binpow(u64 a, u64 b) {
    u64 r = 1; a %= MOD;
    while (b > 0) {
        if (b & 1)  r = MULTIPLY(r, a);
        a = MULTIPLY(a, a);
        b >>= 1;
    }
    return r;
}

u64 modInverse(u64 a) {
    return binpow(a, MOD - 2);
}
```

Leer String

```
string s;
cin>>s;// Sin espacios
getline(cin,s); // Con espacios
```

Limpiar buffer

El usar cin independientemente del tipo de dato que se lea, guardar siempre en el buffer un salto de linea, el cual al querer usar un getline posteriormente nos puede generar problemas.

```
cin >> edad;
cin.ignore(100, '\n');
getline(cin, nombre);
```

Manipulacion de Bits

Operación	Descripción
&	Operador AND Devuelve un 1 en cada bit para los que ambos operandos sean 1.
|	Operador OR. Devuelve un 1 en cada bit para los que al menos uno de los operandos sea 1.

Operación	Descripción
`^`	Operador XOR. Devuelve un 1 en cada bit para los que uno de los operandos sea 1 pero no ambos.
`~`	Operador NOT. Devuelve el complemento a uno del operando.
`<<`	Operador de desplazamiento a la izquierda. Desplaza los bits del operando hacia la izquierda la cantidad especificada por el segundo operando.
`>>`	Operador de desplazamiento a la derecha. Desplaza los bits del operando hacia la derecha la cantidad especificada por el segundo operando, rellenando los bits vacíos con ceros o unos, dependiendo del tipo de dato y de la implementación del compilador.

```
unsigned int a = 0b1100;
unsigned int b = 0b1010;
c = a & b; // c = 0b1000
c = a | b; // c = 0b1110
c = a ^ b; // c = 0b0110
b = ~a; // b = 0b0011
a = 0b1100;
b = a << 2; // b = 0b110000
b = a >> 2; // b = 0b0011
```

Propiedades

- La operación AND a nivel de bits (`&`) puede utilizarse para comprobar si un número es par o impar. Si se realiza la operación `x & 1` y el resultado es 0, entonces el número `x` es par. Si el resultado es 1, entonces el número `x` es impar.
- La operación OR a nivel de bits (`|`) se utiliza a menudo para establecer uno o varios bits de un número en particular. Por ejemplo, si se quiere establecer el tercer bit (contando desde la derecha) de una variable `x` en 1, se puede utilizar la expresión `x |= (1 << 2)`, donde `<<` es el operador de desplazamiento a la izquierda y `2` es la posición del bit que se quiere establecer en 1.
- La operación XOR a nivel de bits (`^`) se utiliza a menudo para invertir el valor de un bit. Por ejemplo, si se quiere invertir el valor del tercer bit (contando desde la derecha) de una variable `x`, se puede utilizar la expresión `x ^= (1 << 2)`.
- La operación NOT a nivel de bits (`~`) puede utilizarse para obtener el complemento a uno de un número. Por ejemplo, si se quiere obtener el complemento a uno de la variable `x`, se puede utilizar la expresión `~x`.
- Las operaciones de desplazamiento a la izquierda (`<<`) y a la derecha (`>>`) se utilizan a menudo para multiplicar y dividir por potencias de 2, respectivamente. Por ejemplo, si se quiere multiplicar un número `x` por 8 (es decir, 2^3), se puede utilizar la expresión `x << 3`. Si se quiere dividir un número `x` por 4 (es decir, 2^2), se puede utilizar la expresión `x >> 2`.

- Las operaciones de bits son muy eficientes en términos de tiempo de ejecución, ya que se realizan directamente sobre los bits de los números, sin necesidad de utilizar algoritmos más complejos.
- Las operaciones AND (&) y OR (|) a nivel de bits se pueden utilizar para intercambiar los valores de dos variables a y b sin necesidad de utilizar una variable temporal. Por ejemplo, si se tiene a = 5 y b = 7, se puede intercambiar sus valores con la siguiente secuencia de operaciones:

```
a ^= b;
b ^= a;
a ^= b;
```

Al finalizar esta secuencia, a tendrá el valor original de b y b tendrá el valor original de a.

La operación XOR a nivel de bits (^) es asociativa, lo que significa que se puede realizar la operación entre tres o más valores en cualquier orden, y el resultado será el mismo. Por ejemplo, si se tienen a = 3, b = 5 y c = 7, se puede calcular a ^ b ^ c o a ^ c ^ b y el resultado será el mismo.

Bitset (STL)

```
    bitset<8> a;
    bitset<5> b = 8;
    // Establecer bits en posiciones específicas en 'a'
    a[0] = 1;
    a[2] = 1;
    a[5] = 1;
    // Contar los bits encendido en 'a'
    int count = a.count();
    // Comprobar si el bit en la posición '3' esta encendido
    bool isSet = a.test(3);
    // Establecer el bit en la posición '4' en '1'
    a.set(4, true);
    // Resetear el bit en la posición 2 a '0'
    a.reset(2);
    // Invertir el valor del bit en la posición '1'
    a.flip(1);

    string bitsetString = a.to_string();
    bitset<8> anotherBitset("10101010");
    bitset<8> c_and = a & b;
    bitset<8> c_or = a | b;
    bitset<8> c_xor = a ^ b;
    bitset<8> c_not = ~a;
```

Entero a Binario

```
string to_binary(int n) {
    bitset<8> bits(n);
```

```
    return bits.to_string();
}
```

Binario a Entero

```
int binary_to_int(string s) {
    return stoi(s, nullptr, 2);
}
```

Operaciones Basicas con bitset

Agrega operaciones basicas usando bitset, esto es recomendable solo en el caso que se necesite realizar operaciones con numero que sean mayores a 2^{64}, dado que un long long no soportaria este tipo de tamaño. En caso de que el numero a representar se menor a $2 * 2^{64} - 1$, se recomienda usar la funcion `.to_ullong`

```
#define MXSZ    64
#define bits    bitset<MXSZ>

//Suma
bits operator+(const bits &x, const bits &y)
{
    bits r;
    bool a, b, c = 0;
    forn(i,0,MXSZ){
        a = x[i] , b = y[i];
        r[i] = a ^ b ^ c;
        c  = (a & b) | (a & c) | (b & c);
    }
    return r;
}
// RESTA
bits operator-(const bits &x, const bits &y) {
    bits r;
    bool a, b, c = 0;
    for (int i = 0; i < MXSZ; i++) {
        a = x[i];
        b = y[i];
        r[i] = a ^ b ^ c;
        c = (!a & b) | (!a & c) | (b & c);
    }
    return r;
}

// MULTIPLICACION
bits operator*(const bits &x, const bits &y) {
    bits r,c;
    for (int i = 0; i < MXSZ; i++) {
        if (y[i]) {
```

```
            c = x << i;
            r = r + c;
        }
    }
    return r;
}

//Requerido para division
bool operator>=(const bits &x, const bits &y){
    for(int i = MXSZ - 1; i >= 0; i--)
       if(x[i] != y[i]) return (x[i] > y[i]);
    return 1;
}

// DIVISION
bits operator/(const bits &x, const bits &y) {
    bits r;
    bits c = x;
    for (int i = MXSZ - 1; i >= 0; i--) {
        if (c >> i >= y) {
            r[i] = 1;
            c = c - (y << i);
        }
    }
    return r;
}


// MODULO
bits operator%(const bits &x, const bits &y) {
    bits r = x;
    for (int i = MXSZ - 1; i >= 0; i--)
        if (r >> i >= y)  r = r - (y << i);
    return r;
}

// POTENCIA
bits pow(const bits &x, const bits &y) {
    bits r = 1;
    bits b = x;
    for (int i = 0; i < MXSZ; i++) {
        if (y[i]) r = r * b;
        b = b * b;
    }
    return r;
}
```

Código de Gray

El código Gray es un sistema numérico en el que los números consecutivos difieren en un solo bit.Cada número en el código Gray se representa en binario, donde un bit cambia su estado entre números consecutivos, lo que facilita la detección y corrección de errores.

Para convertir un número binario a código Gray, si definimos X_2 de N bits como B_{N-1}, B_{N-2}, B_{N-3}, ..., B_0 y su código de Gray como G_{N-1}, G_{N-2}, G_{N-3}, ..., G_0, entonces:

$G_n = B_n \text{ para } n = N-1 \;\; G_i = G_{i+1} \oplus B_i \text{para } i = N-2, N-3, \ldots, 0$

Por ejemplo, si $10_2 = 1010$, donde $B_3 = 1$, $B_2 = 0$, $B_1 = 1$ y $B_0 = 0$, entonces:

$G_3 = B_3 = 1 \;\; G_2 = G_3 \oplus B_2 = 1 \oplus 0 = 1 \;\; G_1 = G_2 \oplus B_1 = 0 \oplus 1 = 1 \;\; G_0 = G_1 \oplus B_0 = 1 \oplus 0 = 1$

Es decir, convertir el número binario 1010 a código de Gray daría como resultado 1111, y $Gray(10) = 15$.

$Gray(n) = n \oplus (n \gg 1)$

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

Divisibilidad

Existen ciertas propiedades para saber si un número es divisible entre un numero k. Usualmente útil cuando los números son extremadamente grandes.

Número	Condición	Número	Condición
2	Si el último dígito es par	51	Si es divisible entre 3 y 17
3	Si la suma de todos los dígitos es divisible entre 3	54	Si es divisible entre 2, 3, y 9
4	Si los últimos 2 dígitos son divisibles entre 4	55	Si es divisible entre 5 y 11
5	Si el último dígito es 0 o 5	57	Si es divisible entre 3 y 19
6	Si es divisible entre 2 y 3	60	Si es divisible entre 2, 3, y 10
7	Si al restar el último dígito multiplicado por 2 del número sin el último dígito es divisible entre 7	63	Si es divisible entre 3, 7, y 9
8	Si los últimos 3 dígitos son divisibles entre 8	64	Si los últimos seis dígitos forman un número divisible entre 64
9	Si la suma de sus dígitos es divisible entre 9	65	Si es divisible entre 5 y 13
10	Si el último dígito es 0	66	Si es divisible entre 2, 3, y 11
11	Si la diferencia de la suma de los dígitos en posiciones pares e impares es divisible entre 11	70	Si es divisible entre 2, 5, y 7
12	Si es divisible entre 3 y 4	72	Si es divisible entre 3, 4, y 9

Número	Condición	Número	Condición
13	Si al restar el último dígito multiplicado por 9 del número sin el último dígito es divisible entre 13	75	Si es divisible entre 3 y 25
14	Si es divisible entre 2 y 7	77	Si es divisible entre 7
15	Si es divisible entre 3 y 5	78	Si es divisible entre 2, 3, y 13
16	Si los últimos cuatro dígitos son divisibles entre 16	81	Si es divisible entre 3 y 27
18	Si es divisible entre 2 y 9	84	Si es divisible entre 2, 3, 4, y 7
20	Si es divisible entre 4 y 5	85	Si es divisible entre 5 y 17
21	Si es divisible entre 3 y 7	87	Si es divisible entre 3 y 29
24	Si es divisible entre 3 y 8	90	Si es divisible entre 2, 3, y 9
25	Termina en 00 o es divisible entre 25, 50, o 75	91	Si es divisible entre 7 y 13
27	Si es divisible entre 3 y 9	93	Si es divisible entre 3 y 31
28	Si es divisible entre 4 y 7	96	Si es divisible entre 3, 4, y 8
30	Si es divisible entre 2, 3, y 5	98	Si es divisible entre 2, 7, y 14
32	Si los últimos cinco dígitos forman un número divisible entre 32	99	Si es divisible entre 3 y 11
33	Si es divisible entre 3 y 11	100	Si los últimos dos dígitos son 0
35	Si es divisible entre 5 y 7		
36	Si es divisible entre 4 y 9		
39	Si es divisible entre 3 y 13		
40	Si es divisible entre 4 y 10		
42	Si es divisible entre 2, 3, y 7		
45	Si es divisible entre 3 y 5		
48	Si es divisible entre 3 y 16		
50	Si es divisible entre 2 y 25		

Obtener la cantidad de divisores

Precomputar la cantidad de divisores en $O(n \log n)$

```
vi getDivisors(int n){
    vi ans(n+1,0);
    forn(i,1,n+1){
        for (int j = i; j <= n; j += i)
            ans[j]++;
    }
    return ans;
}
```

Números primos

Criba de Eratostenes

Obtener si cada número hasta n es primo. $O(n \log(\log(\sqrt(n))))$.

```
vector<bool> checkPrimes(int n){
    vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;

    for (int i = 3; i * i <= n; i += 2) {
        if (is_prime[i]) {
            for (int j = i * i; j <= n; j += 2 * i)
                is_prime[j] = false;
        }
    }

    return is_prime;
}
```

Obtener lista de números primos

```
vector<int> getPrimes(int n){
    vector<int> primes = {2};
    vector<bool> is_prime = checkPrimes(n);

    for(int i = 3; i <= n; i++){
        if(is_prime[i] && i % 2) primes.push_back(i);
    }

    return primes;
}
```

Existe una forma de obtener números primos mediante el minimo factor primo en $O(n)$ acambio de usar más memoria, sin embargo, esta optimización no es realmente muy importante, por lo que únicamente se tiene que usar si $N \leq 10^7$. Tambien esto nos puede permitir obtener la factorización de cualquier número $[2, N]$.

```
vector<int> getPrimesLinear(int n){
  vector<int> lp(n+1); //Minimun Prime Factor
  vector<int> pr; // Primes

  for (int i=2; i <= n; ++i) {
      if (lp[i] == 0) {
          lp[i] = i;
          pr.push_back(i);
      }
      for (int j = 0; i * pr[j] <= n; ++j) {
          lp[i * pr[j]] = pr[j];
          if (pr[j] == lp[i]) {
              break;
          }
      }
  }
  return pr;
}
```

Contar primos en un rango

Computar el número de primos en un rango

```
//O(n log(log(sqrt(n))))
vector<int> countPrimes(int n, vector<bool>& is_prime) {
    vector<int>count(n+1,0);
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) count[i] = count[i-1] + 1;
        else count[i] = count[i-1];
    }
    return count;
}

//O(1)
int countPrimes(int l, int r, vector<int>&count) {
    if(l == 1) return count[r];
    return count[r] - count[l-1];
}
```

Encontrar primos en un rango

Encontrar numeros primos en un rango $[L, R]$ ($R - L + 1 \approx 10^7$) dado que $R \leq 10^{12}$.

$O((R - L + 1) \log\log R + \sqrt(R) \log\log \sqrt(R))$

```
vector<char> segmentedSieve(long long L, long long R) {
    long long lim = sqrt(R);
    vector<char> mark(lim + 1, false);
```

```
    vector<long long> primes;
    for (long long i = 2; i <= lim; ++i) {
        if (!mark[i]) {
            primes.emplace_back(i);
            for (long long j = i * i; j <= lim; j += i)
                mark[j] = true;
        }
    }

    vector<char> isPrime(R - L + 1, true);
    for (long long i : primes)
        for (long long j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
            isPrime[j - L] = false;
    if (L == 1)
        isPrime[0] = false;
    return isPrime;
}
```

Existe una forma de hacerlo sin precomputar los números primos, sin embargo su tiempo de complejidad es peor. Sin embargo, en la practica sigue siendo bastante rápido.

```
vector<char> segmentedSieveNoPreGen(long long L, long long R) {
    vector<char> isPrime(R - L + 1, true);
    long long lim = sqrt(R);
    for (long long i = 2; i <= lim; ++i)
        for (long long j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
            isPrime[j - L] = false;
    if (L == 1)
        isPrime[0] = false;
    return isPrime;
}
```

Pruebas de Primalidad

Revisar si un número es primo en $O(\sqrt(n))$. Cabe mencionar que se considera la optimización que el único número primo par es 2, por lo que únicamente hay que revisar números impares.

```
bool isPrime(int x){
    if(x == 2) return true;
    else if(x % 2 == 0) return false;
    for(int i = 3; i * i  <= x; d+=2){
        if(x % i == 0) return false;
    }
    return true;
}
```

Existen algoritmos que se basan en comprobar de manera probabilistica si un número es primo basado en el Teorema Debil de Fermat (Para más detalles revisar la sección de la Función Toetenica de Euler), cuanto mayor sea la cantidad de iteraciones menor es la probabilidad de equivocarse.

La prueba de Fermat falla con los Números Charmichael ($a^{n-1} \equiv 1 \bmod n$ se cumple para todo a coprimo con n, por ejemplo $561 = 3 \times 11 \times 17$), sin embargo en la practica se sigue usando ya que es bastante rápida y los números de Charmichael son raros.

```
bool isPrimeFermat(int n, int iter=5) {
    if (n < 4)
        return n == 2 || n == 3;

    for (int i = 0; i < iter; i++) {
        int a = 2 + rand() % (n - 3);
        if (binpow(a, n - 1, n) != 1)
            return false;
    }
    return true;
}
```

Miller desmostro que el algoritmo se puede volver determinista si se prueban todas bases $a \leq 2ln(n)^2$, siendo que para los números de 64 bit,basta con chechar las primeras 12 bases primas.

```
using u64 = uint64_t;
using u128 = __uint128_t;

bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
}

bool MillerRabin(u64 n) {
    if (n < 2)
        return false;

    int r = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
```

```
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}
```

Factorización de enteros

Los algoritmos para factorizar un número primo suelen ser bastantes lentos, por lo que se recomiendo aplicar una prueba de primalidad antes de realizar cualquiera de estos algoritmos.

Factorización de Rueda. $O(\sqrt{n}))$. Esto se le tiene que agregar el calculo de numeros primos, con una Criba de Erastotenes, por ejemplo.

```
vector<long long> primes;

vector<long long> factorize(long long n) {
    vector<long long> factorization;
    for (long long d : primes) {
        if (d * d > n)
            break;
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}
```

Variación que devuelve un map donde la clave es la base y el valor el exponente.

```
vector<long long> primes;

map<long long, long long> factorize(long long n) {
    map<long long,long long> factorization;
    for (long long d : primes) {
        if (d * d > n)
            break;
        while (n % d == 0) {
            factorization[d]++;
            n /= d;
        }
    }
    if (n > 1)
        factorization[n]++;
```

```
    return factorization;
}
```

Un numero impar se puede escribir $n = pq$ como la diferencia de cuadrados $n = a^2 - b^2$.

$n = (\frac{p+q}{2})^2 - (\frac{p-q}{2})^2$.

Estos valores p y q se pueden obtener con el metodo de factorización de Fermat. $O(|p-q|)$ cabe mencionar que si los números son demasiado grande o la distancia entre p y q , este algritmo es extremadamente lento.

```
pair<int,int> fermat(int n) {
    int a = ceil(sqrt(n));
    int b2 = a*a - n;
    int b = round(sqrt(b2));
    while (b * b != b2) {
        a = a + 1;
        b2 = a*a - n;
        b = round(sqrt(b2));
    }
    return {a-b,n/(a-b)};
}
```

Busqueda Binaria

Iterativa

```
int binarySearch(vector<int> v, int val) {
    int l = 0, r = v.size()-1;

    while (l <= r) {
        int mid = (l + (r - l)) / 2;

        if (v[mid] == val) return mid;
        else if (v[mid] < val)
            l = mid + 1;
        else
            r = mid - 1;

    }
    return -1;
}
```

Busqueda Binaria STL

```
bool exist(vector<int> v, int val){
    return binary_search(v.begin(),v.end(),val);
}

// Busqueda Binaria
int binarySearch(const vector<int> &v, int val) {
    auto it = lower_bound(v.begin(), v.end(), val);
    if (it != v.end() && *it == val)
        return distance(v.begin(), it);
    return -1;
}

// Encontrar el primer valor que sea menor
int lowerBound(const vector<int> &v, int val) {
    auto it = lower_bound(v.begin(), v.end(), val);
    if (it == v.begin()) return -1;
    return distance(v.begin(), prev(it));
}

//Ecnontrar el primer varor que sea
int upperBound(const vector<int> &v, int val) {
    auto it = upper_bound(v.begin(), v.end(), val);
    if (it == v.end()) return -1;
    return distance(v.begin(), it);
}

int main(){
    vi v = {10, 20, 30, 40, 50};
    int x;
    cin >> x;

    int index = upperBound(v, x);
    cout << index;

    return 0;
}
```

Por Intervalos

```
int busquedaBinaria(vector<int> &v, int l, int r, int val) {
    int res = -1;
    while (l <= r) {
        int mid = (l + (r - l)) / 2;
        if (v[mid] < clave) {
            res = mid;
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
```

```
    return res;
}
```

Busqueda Ternaria

Dada una función $f(x)$ que es unimodal en un rango $[l, r]$. Una función unimodal se refiere a uno de los siguientes comportamientos:

- La función incrementa estrictamente al inicio, alcanza un máximo (en único punto o en un rango) y decrementa estrictamente.
- La función decrementa estrictamente al inicio, alcanza un mínimo e incrementa estrictamente.

Encontrar el máximo de la función $f(x)$ en el intervalo $[l, r]$.

```
double ternary_search(double l, double r) {
    double eps = 1e-9;//set the error limit here
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1);//evaluates the function at m1
        double f2 = f(m2); //evaluates the function at m2
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l);//return the maximum of f(x) in [l, r]
}
```

STL

Vectores

```
void showVector(vector<int>&v){
    for(int i = 0; i < v.size();i++){
        cout<<v[i]<<" ";
    }
}

int main(){
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);

    v.erase(b.begin());
```

```
    v.insert(v.begin(),3);

    sort(v.begin(),v.end());
    showVector(v);
    v.clear();
}
```

Listas Ligadas

```
void showList(list<int>&lista){
    for(auto &x : lista){
        cout<<x<<" ";
    }
}

int main(){

    list<int>lista;

    lista.push_back(10);
    lista.push_back(20);
    lista.push_front(30);
    lista.push_front(40);

    cout<<lista.front()<<" "<<lista.back();
    lista.reverse();
    lista.pop_back();
    lista.sort();
    lista.pop_front();

    showList(lista);

    return 0;
}
```

Colas

```
void showQueue(queue<int>cola){
    queue<int> q = cola;
    while(!q.empty()){
        cout<<q.front()<<;
        q.pop();
    }
}

int main(){

    queue<int>cola;
```

```
    cola.push(10);
    cola.push(20);
    cola.push(30);

    cout<<cola.size()<<cola.front()<<.cola.back();
    showQueue(cola);

    return 0;
}
```

Pila

```
int main(){
    stack<int>pila;

    pila.push(10);
    pila.push(20);
    pila.push(30);

    while(!pila.empty()){
        cout<<pila.top();
        pila.pop();
    }

}
```

Deque

```
void showDequeAndSorted(deque<int> &d) {
    for (int i : d) {
        cout << i << " ";
    }
    cout << endl;
}

int main() {
    deque<int> d = {10, 20, 50, 12, -10, -20, 50, -30, -15};

    d.push_front(100);
    d.push_back(-5);

    showDequeAndSorted(d);

    d.pop_front();
    d.pop_back();

    cout<<d[0]<<endl;
```

```
    return 0;
}
```

Set

```
void showSet(unordered_set<string>hash){
    for(auto &x : hash){
        cout<<x<<" ";
    }

}

int main(){

    unordered_set<string>hash;

    hash.insert("apple");
    hash.insert("banana");
    hash.insert("cherry");

    if(hash.find("apple") == hash.end() ){
        cout<<"False\n";
    }else{
        cout<<"True\n";
    }

    showSet(hash);

    return 0;
}
```

Map

```
void showMap(unordered_map<string,int>&hash){
     for(auto &x : hash){
        cout<<x.first<<":"<<x.second<<endl;
    }
}

int main(){
    unordered_map<string,int>hash;

    hash["apple"] = 1;
    hash["banana"] = 1;
    hash["cherry"] = 1;
    string key = "orange";
    if(hash.find(key) == hash.end()){
        hash[key] = 1;
    }else{
```

```
        hash[key]++;
    }

    showMap(hash);
    return 0;
}
```

Cola de Prioridad

```
void showPq(priority_queue<int> p){
    priority_queue<int>pq = p;
    while(!pq.empty()){
        cout<<pq.top()<<"\n";
        pq.pop();
    }
}
void showMinPq(priority_queue<int> p){
    priority_queue<int>pq = p;
    while(!pq.empty()){
        cout<<-pq.top()<<"\n";
        pq.pop();
    }
}

int main(){
    priority_queue<int>min,max;
    //Otra alternaitiva para el obtener el valor minimo sin la necesidad de
convertirlos a negativo.
    priority_queue<int, vector<int>, greater<int>> min_pq;

    max.push(10);
    max.push(20);
    max.push(50);
    max.push(12);


    min.push(-10);
    min.push(-20);
    min.push(50);
    min.push(-30);
    min.push(-15);

    showPq(max);
    showMinPq(min);
}
```

Heap

```
#define pb push_back
#define sz(a) int(a.size())

template<typename T>
class Heap {
public:
    vector<T> data;
    void push(T v) {
        data.pb(v);
        heapify_up(sz(data) - 1);
    }
    T top() { return data[0]; }
    bool empty() { return data.empty(); }

    void pop() {
        swap(data[0], data[sz(data) - 1]);
        data.pop_back();
        heapify_down(0);
    }

private:
    void heapify_up(int i) {
        if (i == 0) return;

        int p = parent(i);
        if (!isSorted(data[p], data[i])) {
            swap(data[p], data[i]);
            heapify_up(p);
        }
    }

    void heapify_down(int i) {
        int l = 2 * i + 1;
        int r = l + 1;
        int j = i;
        if (l < sz(data) && isSorted(data[l], data[j])) j = l;
        if (r < sz(data) && isSorted(data[r], data[j])) j = r;

        if (j != i) {
            swap(data[i], data[j]);
            heapify_down(j);
        }
    }
    int parent(int i) { return (i - 1) / 2; }
    bool isSorted(T& a, T& b) {//FIX ME
        return a > b;
    }
};
```

Grafos

Grafo No Ponderado

Plantilla

```
template<typename T>
class Graph {
public:

    using vT = vector<T>;
    using vvT = vector<vT>;
    vvT adj;

    Graph(int n) {
        this->n = n;
        adj.resize(n);
    }

    Graph(int n, vvi& mat) {
        this->n = n;
        adj = mat;
    }

    void add(int u, int v, bool undirected = false) {
        adj[u].push_back(v);
        if (undirected)
            adj[v].push_back(u);
    }
};

int main(){
    Graph<int>grafoDirigido(4);
    grafoDirigido.add(0,1);
    grafoDirigido.add(0,3);
    grafoDirigido.add(1,2);
    grafoDirigido.add(2,3);

    Graph<int>grafoNoDirigido(4);
    grafoNoDirigido.add(0,1,true);
    grafoNoDirigido.add(0,3,true);
    grafoNoDirigido.add(1,2,true);
    grafoNoDirigido.add(2,3,true);
}
```

Busqueda en Profundidad (DFS)

```
vb vsisted(n,false);
void dfs(T u, vb& visited) {
    cout << u << " ";
    for (auto &v : adj[u]) {
        if (!visited[v]) {
```

```
            visited[v] = true;
            dfs(v, visited);
        }
    }
}
```

Busqueda en Anchura (BFS)

```
vb vsisted(n,false);
void bfs(T u, vb& visited) {
    queue<T> q;
    q.push(u);
    visited[u] = true;

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " ";

        for (T v : adj[current]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

Componentes Conexas

Una componente conexa es un grupo de nodos en un grafo donde cada nodo puede alcanzar cualquier otro nodo del mismo grupo de caminos del grafo.

```
void dfs(int i, vT & c, vb& u) {
        u[i] = 1;
        c.pb(i);
        for (T v : adj[i]) {
            if (!u[v])
                dfs(v, c, u);
        }
    }
    vvT findComponents() {
        vb visited(n, false);
        vvT components;
        forn(i, 0, n) {
            if (!visited[i]) {
                vT component;
                dfs(i, component, visited);
                components.pb(component);
```

```
            }
        }
        return components;
    }
```

Puentes

Un puente en un grafo es una arista cuya eliminación dividiría el grafo en dos o más componentes conexas adicionales.

```
void findBridges() {
        int timer = 0;
        vb visited(n, false);
        vT tin(n, -1), low(n, -1);
        for (int i = 0; i < n; ++i) {
            if (!visited[i])
                dfs(i, -1, timer, visited, tin, low);
        }
    }

    void dfs(int u, int p, int& timer, vb& vis, vT& tin, vT& low) {
        vis[u] = true;
        tin[u] = low[u] = timer++;
        for (int v : adj[u]) {
            if (v == p) continue;
            if (vis[v]) low[u] = min(low[u], tin[v]);
            else {
                dfs(v, u, timer, vis, tin, low);
                low[u] = min(low[u], low[v]);
                if (low[v] > tin[u]) IS_BRIDGE(u, v);
            }
        }
    }

    //Process the bridge, for example print it
    void IS_BRIDGE(int u, int v) {
        cout << u << " " << v << endl;
    }
```

Puntos de articulación

Un punto de articulación en un grafo es un nodo cuya eliminación dividiría el grafo en dos o más componentes conexas adicionales.

```
 void findArticulationsPoints() {
        int timer = 0;
        vb visited(n, false);
        vT tin(n, -1), low(n, -1);
        forn(i, 0, n) {
```

```
            if (!visited[i])
                dfs(i, -1, timer, visited, tin, low);
        }
    }

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

    //Process Articulation Point
    void IS_CUTPOINT(int u) {
        cout << u << endl;
    }
```

Encontrar ciclos

Un ciclo en un grafo es una secuencia de aristas que comienza y termina en el mismo nodo, permitiendo recorrer el grafo y volver al punto de partida.

```
vT findCycle() {
    vi color(n, 0);
    vT parent(n, -1),cycle;

    int start = -1;

    forn(v, 0, n) {
        if (color[v] == 0 && dfs(v, color, parent, start)) break;
    }

    if (start != -1) {
        cycle.pb(start);
        for (int i = parent[start]; i != start; i = parent[i])
            cycle.pb(i);
        cycle.pb(start);
        reverse(all(cycle));
    }

    return cycle;
```

```
}

bool dfs(int v, vi& c, vT& p, int& s) {
    c[v] = 1;
    for (int u : adj[v]) {
        if (c[u] == 0) {
            p[u] = v;
            if (dfs(u, c, p, s)) return true;
        }
        else if (c[u] == 1 && p[v] != u) {
            s = u;
            return true;
        }
    }
    c[v] = 2;
    return false;
}
```

Bipartito

Un grafo bipartito es un tipo de grafo cuyos vértices pueden dividirse en dos conjuntos disjuntos de tal manera que todas las aristas del grafo conectan un vértice de un conjunto con uno del otro conjunto. Es decir puedes pintar los nodos con dos colores, de tal manera que no haya aristas que conecten nodos del mismo color.

```
bool isBipartite() {
    vi side(n, -1);
    bool ans = true;
    queue<int> q;
    forn(i, 0, n) {
        if (side[i] == -1) {
            q.push(i);
            side[i] = 0;
            while (!q.empty()) {
                int v = q.front();
                q.pop();
                for (int u : adj[v]) {
                    if (side[u] == -1) {
                        side[u] = side[v] ^ 1;
                        q.push(u);
                    }
                    else {
                            ans &= side[u] != side[v];
                        if (!ans) return ans;
                    }
                }
            }
        }
    }
    return ans;
}
```

Camino Euleriano

Un camino euleriano en un grafo es un recorrido que pasa por cada arista exactamente una vez y visita cada vértice al menos una vez. Si el camino comienza y termina en el mismo vértice, se llama ciclo euleriano. Se debe usar matrices de adyacencencia;

```
Graph(int n, vvT& mat) {
    this->n = n;
    adj = mat;
}

vi eulerianPath() {
    // Assuming adj is adjacency matrix
    vi deg(n);
    vi path;
    forn(i, 0, n) {
        for (int j : adj[i])
            deg[i]++;
    }

    int start = 0;
    while (start < n && !deg[start])
        start++;

    if (start == n) return path;

    int v1 = -1, v2 = -1;
    bool bad = false;

    forn(i, 0, n) {
        if (deg[i] & 1) {
            if (v1 == -1) v1 = i;
            else if (v2 == -1) v2 = i;
            else bad = true;
        }
    }

    if (v1 != -1)
        ++adj[v1][v2], ++adj[v2][v1];

    stack<int>st;

    st.push(start);
    while (!st.empty()) {
        int v = st.top();
        int i;
        for (i = 0; i < n; ++i)
            if (adj[v][i]) break;

        if (i == n) {
```

```
            path.pb(v);
            st.pop();
        }
        else {
            --adj[v][i];
            --adj[i][v];
            st.push(i);
        }
    }

    if (v1 != -1) {
        for (size_t i = 0; i + 1 < sz(path); ++i) {
            if ((path[i] == v1 && path[i + 1] == v2) ||
                (path[i] == v2 && path[i + 1] == v1)) {
                vi path2;
                for (size_t j = i + 1; j < sz(path); ++j)
                    path2.pb(path[j]);
                for (size_t j = 1; j <= i; ++j)
                    path2.pb(path[j]);
                path = path2;
                break;
            }
        }
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (adj[i][j])
                bad = true;
        }
    }

    if (bad) {
        vi dummy;
        return dummy;
    }

    return path;
}
```

Oredenamiento Topologico

Ordenar los nodos de un grafo dirigido de manera que ningún nodo preceda a sus nodos dependientes,garantiza que se siga un orden coherente y sin conflictos en situaciones en las que el orden es esencial.

```
vT topologicalSort() {
    vi degree(n, 0);
    vT topoSort;
    queue<int> q;
    forn(u, 0, n) {
```

```
        for (T v : adj[u]) degree[v]++;
    }

    forn(u, 0, n){
        if (degree[u] == 0) q.push(u);
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        topoSort.pb(u);
        for (int v : adj[u]) {
            degree[v]--;
            if (degree[v] == 0) q.push(v);
        }
    }
    //No existe
    if (topoSort.size() != n)
        topoSort.clear();

    return topoSort;
}
```

Grafos ponderados

Plantilla

```
template<typename T, typename W>
class Graph {
public:
    int n;
    using vT = vector<T>;
    using vvT = vector<vT>;
    using vW = vector<W>;
    using vvW = vector<vW>;
    using pTW = pair<T, W>;
    using vpTW = vector<pTW>;
    using vvpTW = vector<vpTW >;

    vvpTW adj;

    Graph(int n) {
        this->n = n;
        adj.resize(n);
    }

    void add(T u, T v, W w, bool directed = false) {
        adj[u].pb({ v,w });
        if (directed) adj[v].pb({ u,w });
    }
private:
```

```
    const W INF = numeric_limits<W>::max();
};

int main(){
    Graph<int, int> g(5);
    g.add(0, 1, 2);
    g.add(0, 2, 5);
    g.add(1, 2, 1);
    g.add(1, 3, 6);
    g.add(2, 3, 3);
    g.add(2, 4, 8);

    vi dist = g.dikkstra(0);
}
```

Camino Más Corto

Dijkstra

El algoritmo de Dijkstra se utiliza para encontrar el camino más corto desde un nodo de inicio a todos los demás nodos en un grafo con pesos no negativos.

```
vW dijkstra(T s) {
    vW d(n, INF);
    vT p(n, -1);
    d[s] = 0;

    priority_queue<pTW, vpTW, greater<pTW>> pq;
    pq.push({ 0, s });

    while (!pq.empty()) {
        T u = pq.top().se;
        W du = pq.top().fi;
        pq.pop();

        if (du != d[u]) continue;
        for (auto& child : adj[u]) {
            T v = child.fi;
            W w = child.se;
            if (d[u] + w < d[v]) {
                d[v] = d[u] + w;
                p[v] = u;
                pq.push({ d[v], v });
            }
        }
    }
    return d;
}
```

Bellman-Ford

El algoritmo de Bellman-Ford se utiliza para encontrar las distancias del camino más corto desde un nodo de inicio a todos los demás nodos en un grafo ponderado, incluso cuando el grafo contiene aristas con pesos negativos.

```
vW bellmanFord(T s) {
    vW d(n, INF);
    d[s] = 0;
    for (;;) {
        bool any = false;
        forn(u, 0, n) {
            for (auto& child : adj[u]) {
                T v = child.fi;
                W w = child.se;

                if (d[u] < INF) {
                    if (d[v] > d[u] + w) {
                        d[v] = d[u] + w;
                        any = true;
                    }
                }
            }
        }
        if (!any)break;
    }
    return d;
}
```

Obtener el camino más corto

```
vT bellmanFordPath(T s) {
    vW d(n, INF);
    vT p(n, -1);
    d[s] = 0;
    for (;;) {
        bool any = false;
        forn(u, 0, n) {
            for (auto& child : adj[u]) {
                T v = child.fi;
                W w = child.se;

                if (d[u] < INF && d[v] > d[u] + w) {
                    d[v] = d[u] + w;
                    p[v] = u;
                    any = true;
                }
            }
        }
        if (!any) break;
        }
    vT path;
```

```
    for (T v = 0; v < n; ++v) {
        if (p[v] != -1) {
            T current = v;
            stack<T> temp;
            temp.push(v);
            while (current != s) {
                current = p[current];
                temp.push(current);
            }
            while (!temp.empty()) {
                path.pb(temp.top());
                temp.pop();
            }
        }
    }
    return path;
}
```

Modificcion con ciclos negativos

```
void bellmanFordNegativeCycles(T s) {
    vW d(n, INF);
    d[s] = 0;
    vT p(n, -1);
    W x;
    forn(i, 0, n) {
        x = -1;
        forn(u, 0, n) {
            for (auto& child : adj[u]) {
                T v = child.fi;
                W w = child.se;
                if (d[u] < INF && d[v] > d[u] + w) {
                    d[v] = max(-INF, d[u] + w);
                    p[v] = u;
                    x = v;
                }
            }
        }
    }

    if (x == -1)
        cout << "No negative cycle from " << s;
    else {
        T y = x;
        forn(i, 0, n) y = p[y];

        vT path;
        for (T cur = y;; cur = p[cur]) {
            path.pb(cur);
            if (cur == y && sz(path) > 1) break;
        }
        reverse(all(path));
```

```
        cout << "Negative cycle: ";
        for (T u : path) cout << u << " ";
    }
}
```

BFS/0-1

Cuando los pesos son 0 o 1, es más eficiente utilizar una modificación de la BFS

```
vT bfs01(T s) {
    vW d(n, INF);
    d[s] = 0;
    deque<T> q;
    q.push_front(s);

    while (!q.empty()) {
        int v = q.front();
        q.pop_front();
        for (auto child : adj[v]) {
            int u = child.fi;
            int w = child.se;
            if (d[v] + w < d[u]) {
                d[u] = d[v] + w;
                if (w == 1) q.pb(u);
                else q.push_front(u);
            }
        }
    }
    return d;
}
```

Floyd-Warshall

El algoritmo de Floyd-Warshall es un algoritmo de búsqueda de caminos más cortos que encuentra las distancias mínimas entre todos los pares de nodos en un grafo ponderado y dirigido o no dirigido, lo que lo hace adecuado para detectar ciclos negativos y para resolver problemas como la búsqueda de rutas óptimas

```
vvW floydWarshall() {
    vvW d(n, vector<W>(n, INF));

    for (int u = 0; u < n; ++u) {
        for (auto& child : adj[u]) {
            T v = child.fi;
            W w = child.se;
            d[u][v] = w;
        }
        d[u][u] = 0;
    }
    forn(k, 0, n) {
```

```
        forn(i, 0, n) {
            forn(j, 0, n) {
                if (d[i][k] < INF && d[k][j] < INF)
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }
    return d;
}
```

Minimum Spanning Trees

Un Minimum Spanning Tree (MST) es el árbol que conecta todos los nodos de un grafo con el menor costo posible, donde el costo se calcula sumando los pesos de las aristas seleccionadas.

Kruskal

Kruskal es un algoritmo para encontrar el árbol de expansión mínima en un grafo ponderado no dirigido. Este algoritmo selecciona las aristas más pequeñas y las agrega al árbol de expansión, evitando la formación de ciclos.

Kruskal es un algoritmo que ayuda a encontrar la forma más barata de conectar todos los nodos en un conjunto de nodos. Imagina que tienes ciudades y quieres construir carreteras para conectarlas de la manera más eficiente.

```
vvpTW kruskal() {
    W cost = 0;
    vvpTW result(n);
    vT parent(n), rank(n);
    for (T i = 0; i < n; ++i) make_set(i, parent, rank);
    vector<pair<W, pair<T, T>>> edges;
    for (T u = 0; u < n; ++u) {
        for (auto& child : adj[u]) {
            T v = child.first;
            W w = child.second;
            edges.push_back({ w, {u, v} });
        }
    }

    sort(all(edges));

    for (auto& edge : edges) {
        T u = edge.second.first;
        T v = edge.second.second;
        W w = edge.first;
        if (find_set(u, parent) != find_set(v, parent)) {
            cost += w;
            result[u].pb({ v, w });
            union_sets(u, v, parent, rank);
        }
    }
```

```
    return result;
}

void make_set(T v, vT& parent, vT& rank) {
    parent[v] = v;
    rank[v] = 0;
}

T find_set(T v, vT& parent) {
    if (v == parent[v]) return v;
    return parent[v] = find_set(parent[v], parent);
}

void union_sets(T a, T b, vT& parent, vT& rank) {
    a = find_set(a, parent);
    b = find_set(b, parent);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b]) rank[a]++;
    }
}
```

Disjoint Set

Un conjunto disjunto es una estructura que agrupa elementos en diferentes conjuntos, donde cada conjunto no comparte elementos con los otros conjuntos.

Plantilla

```
template<typename T>
class DisjointSet {
public:
    using vT = vector<T>;
    using vvT = vector<vT>;

    int n;
    vT parent, rank, size;
    enum Type { BASIC, SIZE, RANK };
    Type type;

    DisjointSet(int n, Type type = BASIC) {
        this->n = n;
        this->type = type;
        parent.resize(n);
        if (type == SIZE) size.resize(n, 1);
        if (type == RANK) rank.resize(n, 0);
        for (int i = 0; i < n; ++i) make(i);
    }
```

```
    void make(T v) {
        parent[v] = v;
    }

    T find(T v) {
        if (v == parent[v]) return v;
        return parent[v] = find(parent[v]);
    }

    void unionSet(T a, T b) {
        a = find(a);
        b = find(b);
        if (a != b) parent[b] = a;
    }

    bool sameGroup(T a, T b) {
        return find(a) == find(b);
    }

    vvT groups() {
        vvT group(n), ans;
        forn(i, 0, n)
            group[find(i)].pb(i);

        for (auto& g : group)
            if (!g.empty()) ans.pb(g);

        return ans;
    }
};

int main(){
    DisjointSet<int> ds(5);

    ds.unionSet(0, 1);
    ds.unionSet(2, 3);

    cout << (ds.sameGroup(0, 1) ? "Yes" : "No") << endl;
    cout << (ds.sameGroup(0, 2) ? "Yes" : "No") << endl;

    auto groups = ds.groups();
    for (auto& g : groups) {
        for (auto& e : g)  cout << e << " ";
        cout << endl;
    }
}
```

Existen optimizaciones que busca unir conjuntos basados en el rango o tamaño del conjunto.

Union By Rank

```
void unionByRank(T a, T b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (rank[a] < rank[b]) swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b]) rank[a]++;
    }

}

int main(){
    DisjointSet<int> ds(5, DisjointSet<int>::RANK);

    ds.unionByRank(0, 1);
    ds.unionByRank(2, 3);
}
```

Union By Size

```
void unionBySize(T a, T b) {
        a = find(a);
        b = find(b);
        if (a != b) {
            if (size[a] < size[b]) swap(a, b);
            parent[b] = a;
            size[a] += size[b];
        }
}

int sizeOfGroup(T a) {
    return size[find(a)];
}

int main(){
    DisjointSet<int> ds(5, DisjointSet<int>::SIZE);
    ds.unionBySize(0, 1);
    ds.unionBySize(2, 3);

    cout << (ds.sameGroup(0, 1) ? "Yes" : "No") << endl;
    cout << (ds.sameGroup(0, 2) ? "Yes" : "No") << endl;

    cout << ds.sizeOfGroup(0) << endl;
}
```

Lowest Common Ancestor (Version no generica)

```
class Grafo {
public:
    int n;
    vector<vector<int>> grafo;
    Grafo(int n) {
        this->n = n;
        grafo.resize(n);
    }

    void add(int u, int v) {
        grafo[u].push_back(v);
        grafo[v].push_back(u);
    }

    int encontrarLCA(int nodo1, int nodo2) {
        vector<int> padre(n, -1);
        vector<int> nivel(n, -1);
        set<int> visitado;

        dfs(nodo1, padre, nivel, visitado, 0);

        int lca = obtenerLCA(nodo1, nodo2, padre, nivel);
        return lca;
    }

    void dfs(int nodo, vector<int>& padre, vector<int>& nivel, set<int>& visitado,
int nivel_actual) {
        visitado.insert(nodo);
        nivel[nodo] = nivel_actual;

        for (int vecino : grafo[nodo]) {
            if (visitado.find(vecino) == visitado.end()) { //
                padre[vecino] = nodo;
                dfs(vecino, padre, nivel, visitado, nivel_actual + 1);
            }
        }
    }

    int obtenerLCA(int nodo1, int nodo2, vector<int>& padre, vector<int>& nivel) {
        if (nivel[nodo1] < nivel[nodo2]) swap(nodo1, nodo2);

        while (nivel[nodo1] > nivel[nodo2]) nodo1 = padre[nodo1];

        if (nodo1 == nodo2)return nodo1;


        while (padre[nodo1] != padre[nodo2]) {
            nodo1 = padre[nodo1];
            nodo2 = padre[nodo2];
        }

        return padre[nodo1];
    }
```

```
};

int main() {
    Grafo grafo(6);
    grafo.add(0, 1);
    grafo.add(0, 2);
    grafo.add(1, 3);
    grafo.add(1, 4);
    grafo.add(2, 5);

    int nodo1 = 3;
    int nodo2 = 4;
    int lca = grafo.encontrarLCA(nodo1, nodo2);

    cout << "Lowest Common Ancestor (LCA) de " << nodo1 << " y " << nodo2 << " es
" << lca << endl;

    return 0;
}
```

Segment Tree

Un Segment Tree es una estructura de datos que permite hacer consultas eficientes sobre intervalos en un arreglo, como hallar la suma o el máximo en un rango dado.

Plantilla

```
template<typename T>
class SegmentTree {
public:

    vector<T> data;
    vector<T> st;
    int n;

    SegmentTree(vector<T>& a) {
        n = sz(a);
        data = a;
        st.resize(4 * n);
        build(1, 0, n - 1);
    }

    T query(int l, int r) {
        return query(1, 0, n - 1, l, r);
    }

    void update(int pos, T val) {
        update(1, 0, n - 1, pos, val);
    }
```

```
    T operation(T& a, T& b) {
        //Modifica esta funcion de acuerdo el tipo de ST
        return a+b;
    }

private:

    void build(int v, int l, int r) {
        if (l == r) {
            st[v] = data[l];
            return;
        }
        int m = mid(l, r);
        build(lc(v), l, m);
        build(rc(v), m + 1, r);

        T a = st[lc(v)];
        T b = st[rc(v)];

        st[v] = operation(a, b);
    }

    T query(int v, int l, int r, int ql, int qr) {
        if (l > qr || r < ql)
            return 0;

        if (l >= ql && r <= qr)
            return st[v];

        int m = mid(l, r);
        T a = query(lc(v), l, m, ql, qr);
        T b = query(rc(v), m + 1, r, ql, qr);

        return operation(a, b);
    }

    void update(int v, int l, int r, int pos, T val) {
        if (l == r) { st[v] = val; return; }
        int m = mid(l, r);
        if (pos <= m)  update(lc(v), l, m, pos, val);
        else  update(rc(v), m + 1, r, pos, val);

        T a = st[lc(v)];
        T b = st[rc(v)];

        st[v] = operation(a, b);
    }

    int find(int v, int l, int r, T& val) {
        if (l == r) {
            if (st[v] == val) return l;
            else return -1;
        }
```

```
        int m = mid(l, r);
        int leftIndex = find(lc(v), l, m, val);

        if (leftIndex != -1) return leftIndex;
        else return find(rc(v), m + 1, r, val);
    }

    int lc(int v) { return 2 * v; }
    int rc(int v) { return 2 * v + 1; }
    int mid(int l, int r) { return (l + r) / 2; }
};

int main(){
    vector<int> a = { 1, 2, 3, 4, 5 };
    SegmentTree<int> st(a);

    cout << st.query(0, 2) << endl;
    cout << st.query(1, 3) << endl;
    cout << st.query(2, 4) << endl;
    cout << st.query(3, 4) << endl;
    cout << st.query(0, 4) << endl;

    st.update(0, 100);
    cout << st.query(0, 4) << endl;
}
```

Algunos ejemplos de modificaciones de Segment Tree

```
// Guarda la suma en rangos
T operation(T& a, T& b) {
        return a+b;
}

// Guarda el valor maximo en rangos
T operation(T& a, T& b) {
        return max(a,b);
}

// Guarda el valor XOR en rangos
T operation(T& a, T& b) {
        return a^b;
}
```

Funciones de busqueda

Regresar el indice del arbol que contenga exactamente ese valor

```
int find(T& val) {
    return find(1, 0, n - 1, val);
```

```
}

int find(int v, int l, int r, T& val) {
    if (l == r) {
        if (st[v] == val) return l;
        else return -1;
    }
    int m = mid(l, r);
    int leftIndex = find(lc(v), l, m, val);

    if (leftIndex != -1) return leftIndex;
    else return find(rc(v), m + 1, r, val);
}
```

Encontrar el primer valor que sea mayor o igual en el arbol

```
int lower_bound(T& val) {
    return lower_bound(1, 0, n - 1, val);
}
int lower_bound(int v, int l, int r, T& val) {
    if (l == r) {
        if (st[v] >= val) return l;
        else return -1;
    }
    int m = mid(l, r);
    if (st[lc(v)] >= val)
         return lower_bound(lc(v), l, m, val);
    else
        return lower_bound(rc(v), m + 1, r, val);
}
```

Encontrar el primer valor que sea estrictamente mayor en el arbol

```
int upper_bound(T& val) {
    return upper_bound(1, 0, n - 1, val);
}

int upper_bound(int v, int l, int r, T& val) {
    if (l == r) {
        if (st[v] > val) return l;
        else return -1;
    }
    int m = mid(l, r);
    if (st[lc(v)] > val)
        return upper_bound(lc(v), l, m, val);
    else
        return upper_bound(rc(v), m + 1, r, val);
}
```

Busqueda en tableros

```
struct Celda {
    int x, y, distancia;
    Celda(int x, int y, int distancia): x(x), y(y), distancia(distancia) {}
};

// Función para validar si una celda es válida
bool esCeldaValida(vector<vector<char>>& grid, int x, int y) {
    // Verificamos si la celda está dentro de la matriz
    if (x < 0 || x >= grid.size() || y < 0 || y >= grid[0].size()) {
        return false;
    }

    // Verificamos si la celda no está bloqueada
    if (grid[x][y] == '#') {
        return false;
    }

    // Si la celda está dentro de la matriz y no está bloqueada, es válida
    return true;
}

// Función para buscar el camino desde la posición inicial hasta la final en la
matriz
int buscarCamino(vector<vector<char>>& grid, int filaInicio, int colInicio, int
filaFin, int colFin) {
    // Vector para almacenar si una celda ha sido visitada o no
    vector<vector<bool>> visitado(grid.size(), vector<bool>(grid[0].size(),
false));

    // Cola para almacenar las celdas a visitar
    queue<Celda> cola;
    cola.push(Celda(filaInicio, colInicio, 0));
    visitado[filaInicio][colInicio] = true;

    // Movimientos válidos
    int dx[] = {-1, 0, 1, 0};
    int dy[] = {0, 1, 0, -1};

    // Búsqueda en anchura
    while (!cola.empty()) {
        Celda actual = cola.front();
        cola.pop();

        // Si hemos llegado a la celda final, devolvemos la distancia
        if (actual.x == filaFin && actual.y == colFin) {
            return actual.distancia;
        }

        // Visitamos las celdas adyacentes
```

```
        for (int i = 0; i < 4; i++) {
            int siguienteX = actual.x + dx[i];
            int siguienteY = actual.y + dy[i];

            // Verificamos si la celda adyacente es válida y no ha sido visitada
            if (esCeldaValida(grid, siguienteX, siguienteY) &&
!visitado[siguienteX][siguienteY]) {
                cola.push(Celda(siguienteX, siguienteY, actual.distancia + 1));
                visitado[siguienteX][siguienteY] = true;
            }
        }
    }

    // Si no se puede llegar a la celda final, devolvemos -1
    return -1;
}

int main() {
    // Matriz de ejemplo
    vector<vector<char>> grid = {{'.', '.', '.', 'I', '.', '#'},
                                 {'.', '.', '#', '#', '#', '.'},
                                 {'.', '.', '.', '.', '#', '.'},
                                 {'.', '.', '#', '.', '.', '.'},
                                 {'.', '#', '.', '.', '#', '.'},
                                 {'.', '.', '.', 'F', '.', '.'}};

    // Buscamos el camino desde la posición inicial hasta la final
    int filaInicio = 0, colInicio = 3
    int filaFin = 5, colFin = 3;
    int distancia = buscarCamino(grid, filaInicio, colInicio, filaFin, colFin);

    // Imprimimos la distancia
    if (distancia == -1) {
        cout << "No se puede llegar a la celda final" << endl;
    } else {
        cout << "La distancia desde la posición inicial hasta la final es: " <<
distancia << endl;
    }

    return 0;

}
```

Arboles Binarios

A partir de una cadena preorden

```
struct Nodo {
    int valor;
```

```
    Nodo* izq;
    Nodo* der;

    Nodo(int val) {
        valor = val;
        izq = nullptr;
        der = nullptr;
    }
};

class ArbolBinario {
private:
    Nodo* raiz;

    Nodo* insertar(Nodo* actual, int valor) {
        if (actual == nullptr) {
            return new Nodo(valor);
        }
        if (valor < actual->valor) {
            actual->izq = insertar(actual->izq, valor);
        }
        else {
            actual->der = insertar(actual->der, valor);
        }
        return actual;
    }

public:
    ArbolBinario() {
        raiz = nullptr;
    }

    void construirArbol(string cadena) {
        stack<Nodo*> pila;
        int i = 0;
        while (i < cadena.length()) {
            if (cadena[i] == ',' || cadena[i] == '{') {
                i++;
                continue;
            }
            else if (cadena[i] == '#') {
                if (!pila.empty()) {
                    Nodo* nodoActual = pila.top();
                    pila.pop();
                    if (!pila.empty()) {
                        Nodo* nodoPadre = pila.top();
                        pila.pop();
                        nodoPadre->der = nodoActual;
                    }
                    else {
                        raiz = nodoActual;
                    }
                }
            }
```

```
            else {
                int valor = 0;
                while (i < cadena.length() && cadena[i] != ',' && cadena[i] !=
'}') {
                    valor = valor * 10 + (cadena[i] - '0');
                    i++;
                }
                i--;
                Nodo* nodoNuevo = new Nodo(valor);
                pila.push(nodoNuevo);
            }
            i++;
        }
    }

    void imprimirArbol(Nodo* nodo) {
        if (nodo != nullptr) {
            cout << nodo->valor << " ";
            imprimirArbol(nodo->izq);
            imprimirArbol(nodo->der);
        }
    }

    void imprimir() {
        imprimirArbol(raiz);
        cout << endl;
    }
};

int main() {
    ArbolBinario arbol;
    string cadena = "{1,2,4,#,#,5,#,11,#,#,3,6,#,#,7,8,9,#,#,10,#,#,#}";



    arbol.construirArbol(cadena);
    arbol.imprimir();

    return 0;
}
```

De busqueda

```
struct Nodo {
    int valor;
    Nodo* izq;
    Nodo* der;

    Nodo(int val) {
        valor = val;
        izq = nullptr;
```

```
        der = nullptr;
    }
};

class ArbolBinario {
private:
    Nodo* raiz;

    Nodo* insertar(Nodo* actual, int valor) {
        if (actual == nullptr) {
            return new Nodo(valor);
        }
        if (valor < actual->valor) {
            actual->izq = insertar(actual->izq, valor);
        }
        else {
            actual->der = insertar(actual->der, valor);
        }
        return actual;
    }

    void inorder(Nodo* actual) {
        if (actual == nullptr) {
            return;
        }
        inorder(actual->izq);
        cout << actual->valor << " ";
        inorder(actual->der);
    }

    void preorder(Nodo* actual) {
        if (actual == nullptr) {
            return;
        }
        cout << actual->valor << " ";
        preorder(actual->izq);
        preorder(actual->der);
    }

    void postorder(Nodo* actual) {
        if (actual == nullptr) {
            return;
        }
        postorder(actual->izq);
        postorder(actual->der);
        cout << actual->valor << " ";
    }

    void levelorder(Nodo* actual) {
        if (actual == nullptr) {
            return;
        }

        queue<Nodo*> cola;
```

```
        cola.push(actual);

        while (!cola.empty()) {
            Nodo* actual = cola.front();
            cola.pop();
            cout << actual->valor << " ";
            if (actual->izq != nullptr) {
                cola.push(actual->izq);
            }
            if (actual->der != nullptr) {
                cola.push(actual->der);
            }
        }
    }

    Nodo* eliminar(Nodo* actual, int valor) {
    if (actual == nullptr) {
        return nullptr;
    }

    if (valor < actual->valor) {
        actual->izq = eliminar(actual->izq, valor);
    } else if (valor > actual->valor) {
        actual->der = eliminar(actual->der, valor);
    } else {
        // Caso 1: el nodo no tiene hijos
        if (actual->izq == nullptr && actual->der == nullptr) {
            delete actual;
            return nullptr;
        }
        // Caso 2: el nodo tiene un solo hijo
        else if (actual->izq == nullptr) {
            Nodo* temp = actual->der;
            delete actual;
            return temp;
        } else if (actual->der == nullptr) {
            Nodo* temp = actual->izq;
            delete actual;
            return temp;
        }
        // Caso 3: el nodo tiene dos hijos
        else {
            Nodo* sucesor = encontrarSucesor(actual->der);
            actual->valor = sucesor->valor;
            actual->der = eliminar(actual->der, sucesor->valor);
        }
    }
    return actual;
}

    Nodo* encontrarSucesor(Nodo* actual) {
    while (actual->izq != nullptr) {
        actual = actual->izq;
    }
```

```
        return actual;
    }

    public:
        ArbolBinario() {
            raiz = nullptr;
        }

        void insertar(int valor) {
            raiz = insertar(raiz, valor);
        }

        void eliminar(int valor) {
            raiz = eliminar(raiz, valor);
        }

        void inorder() {
            inorder(raiz);
        }

        void preorder() {
            preorder(raiz);
        }

        void postorder() {
            postorder(raiz);
        }

        void levelorder() {
            levelorder(raiz);
        }


    };

    int main() {
        ArbolBinario arbol;

        cout << "Inorder: ";
        arbol.inorder();
        cout << endl;

        cout << "Preorder: ";
        arbol.preorder();
        cout << endl;

        cout << "Postorder: ";
        arbol.postorder();
        cout << endl;

        cout << "Levelorder: ";
        arbol.levelorder();
        cout << endl;
```

```
    return 0;
}
```

Metodos

Función	Descripción
`insertar`	Inserta un nuevo nodo con el valor especificado en el árbol binario. Si el árbol está vacío, crea un nuevo nodo y lo establece como raíz. Si el valor es menor que el valor del nodo actual, se inserta en el subárbol izquierdo, de lo contrario en el subárbol derecho.
`eliminar`	Elimina un nodo con el valor especificado del árbol binario. Si el nodo tiene hijos, se reemplaza por su sucesor inmediato (el nodo más pequeño en el subárbol derecho) o su predecesor inmediato (el nodo más grande en el subárbol izquierdo). Si el nodo no tiene hijos, simplemente se elimina. Si el valor no se encuentra en el árbol, no se realiza ninguna acción.
`inorder`	Imprime los nodos del árbol en orden ascendente, primero los del subárbol izquierdo, luego la raíz y finalmente los del subárbol derecho.
`preorder`	Imprime los nodos del árbol en preorden, primero la raíz, luego los del subárbol izquierdo y finalmente los del subárbol derecho.
`postorder`	Imprime los nodos del árbol en postorden, primero los del subárbol izquierdo, luego los del subárbol derecho y finalmente la raíz.
`levelorder`	Imprime los nodos del árbol por niveles, comenzando por la raíz y luego imprimiendo los nodos del nivel 1, nivel 2, y así sucesivamente.

AVL

```
struct Nodo {
    int valor;
    Nodo* izquierda;
    Nodo* derecha;
    int altura;
};



struct ArbolAVL {

    Nodo* raiz = NULL;

    int altura(Nodo* nodo) {
        if (nodo == NULL) {
            return 0;
        }
        return nodo->altura;
```

```
    }

    int obtenerBalance(Nodo* nodo) {
        if (nodo == NULL) {
            return 0;
        }
        return altura(nodo->izquierda) - altura(nodo->derecha);
    }

    Nodo* nuevoNodo(int valor) {
        Nodo* nodo = new Nodo;
        nodo->valor = valor;
        nodo->izquierda = NULL;
        nodo->derecha = NULL;
        nodo->altura = 1;
        return nodo;
    }

    Nodo* rotarDerecha(Nodo* nodo) {
        Nodo* nodoIzquierda = nodo->izquierda;
        Nodo* nodoDerecha = nodoIzquierda->derecha;
        nodoIzquierda->derecha = nodo;
        nodo->izquierda = nodoDerecha;
        nodo->altura = max(altura(nodo->izquierda), altura(nodo->derecha)) + 1;
        nodoIzquierda->altura = max(altura(nodoIzquierda->izquierda),
altura(nodoIzquierda->derecha)) + 1;
        return nodoIzquierda;
    }

    Nodo* rotarIzquierda(Nodo* nodo) {
        Nodo* nodoDerecha = nodo->derecha;
        Nodo* nodoIzquierda = nodoDerecha->izquierda;
        nodoDerecha->izquierda = nodo;
        nodo->derecha = nodoIzquierda;
        nodo->altura = max(altura(nodo->izquierda), altura(nodo->derecha)) + 1;
        nodoDerecha->altura = max(altura(nodoDerecha->izquierda),
altura(nodoDerecha->derecha)) + 1;
        return nodoDerecha;
    }

    Nodo* insertar(Nodo* nodo, int valor) {
        if (nodo == NULL) {
            return nuevoNodo(valor);
        }
        if (valor < nodo->valor) {
            nodo->izquierda = insertar(nodo->izquierda, valor);
        }
        else if (valor > nodo->valor) {
            nodo->derecha = insertar(nodo->derecha, valor);
        }
        else {
            return nodo;
        }
        nodo->altura = 1 + max(altura(nodo->izquierda), altura(nodo->derecha));
```

```
        int balance = obtenerBalance(nodo);
        if (balance > 1 && valor < nodo->izquierda->valor) {
            return rotarDerecha(nodo);
        }
        if (balance < -1 && valor > nodo->derecha->valor) {
            return rotarIzquierda(nodo);
        }
        if (balance > 1 && valor > nodo->izquierda->valor) {
            nodo->izquierda = rotarIzquierda(nodo->izquierda);
            return rotarDerecha(nodo);
        }
        if (balance < -1 && valor < nodo->derecha->valor) {
            nodo->derecha = rotarDerecha(nodo->derecha);
            return rotarIzquierda(nodo);
        }
        return nodo;
    }

    void preorden(Nodo* nodo) {
        if (nodo != NULL) {
            cout << nodo->valor << " ";
            preorden(nodo->izquierda);
            preorden(nodo->derecha);
        }
    }

    void push(int val) {
        raiz = insertar(raiz, val);
    }
};



int main() {

    ArbolAVL* avl = new ArbolAVL();

    avl->push(10);
    avl->push(20);
    avl->push(30);
    avl->push(40);
    avl->push(50);
    avl->push(25);


    cout << "Preorden del arbol AVL construido es: ";
    avl->preorden(avl->raiz);
    cout << endl;

    return 0;
}
```

Metodos

Función	Descripción
`Nodo* nuevoNodo(int valor)`	Crea un nuevo nodo con el valor dado y apuntadores nulos a izquierda y derecha.
`Nodo* rotarDerecha(Nodo* nodo)`	Realiza una rotación hacia la derecha en el subárbol cuya raíz es el nodo dado, actualiza las alturas de los nodos y retorna la nueva raíz.
`Nodo* rotarIzquierda(Nodo* nodo)`	Realiza una rotación hacia la izquierda en el subárbol cuya raíz es el nodo dado, actualiza las alturas de los nodos y retorna la nueva raíz.
`Nodo* insertar(Nodo* nodo, int valor)`	Inserta un nuevo nodo con el valor dado en el árbol con raíz en el nodo dado, realiza rotaciones si es necesario para balancear el árbol y retorna la nueva raíz del árbol.
`void preorden(Nodo* nodo)`	Recorre el árbol en preorden, imprimiendo los valores de los nodos.
`void push(int val)`	Inserta un nuevo nodo con el valor dado en el árbol, utilizando la función `insertar`.

Trie

```
const int ALPHABET_SIZE = 26; // Tamaño del alfabeto (en este caso, letras
minúsculas del inglés)

class Trie {
    struct TrieNode {
        bool es_fin;
        TrieNode* hijos[ALPHABET_SIZE];
        TrieNode() {
            es_fin = false;
            memset(hijos, 0, sizeof(hijos));
        }
    };
    TrieNode* raiz;
public:
    Trie() {
        raiz = new TrieNode();
    }
    void insertar(string palabra) {
        TrieNode* actual = raiz;
        for (char c : palabra) {
            int idx = c - 'a';
            if (!actual->hijos[idx]) {
                actual->hijos[idx] = new TrieNode();
            }
```

```
            actual = actual->hijos[idx];
        }
        actual->es_fin = true;
    }
    bool buscar(string palabra) {
        TrieNode* actual = raiz;
        for (char c : palabra) {
            int idx = c - 'a';
            if (!actual->hijos[idx]) {
                return false;
            }
            actual = actual->hijos[idx];
        }
        return actual->es_fin;
    }
    void borrar(string palabra) {
        borrar_util(raiz, palabra, 0);
    }
    void imprimir() {
        string palabra;
        imprimir_util(raiz, palabra);
    }
    void limpiar() {
        limpiar_util(raiz);
        raiz = new TrieNode();
    }
private:
    void borrar_util(TrieNode* nodo, string palabra, int nivel) {
        if (!nodo) {
            return;
        }
        if (nivel == palabra.length()) {
            nodo->es_fin = false;
            return;
        }
        int idx = palabra[nivel] - 'a';
        borrar_util(nodo->hijos[idx], palabra, nivel + 1);
        if (es_vacio(nodo->hijos[idx]) && !nodo->es_fin) {
            delete nodo->hijos[idx];
            nodo->hijos[idx] = nullptr;
        }
    }
    void imprimir_util(TrieNode* nodo, string& palabra) {
        if (nodo->es_fin) {
            cout << palabra << endl;
        }
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            if (nodo->hijos[i]) {
                palabra.push_back('a' + i);
                imprimir_util(nodo->hijos[i], palabra);
                palabra.pop_back();
            }
        }
    }
```

```
    void limpiar_util(TrieNode* nodo) {
        if (!nodo) {
            return;
        }
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            limpiar_util(nodo->hijos[i]);
        }
        delete nodo;
    }
    bool es_vacio(TrieNode* nodo) {
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            if (nodo->hijos[i]) {
                return false;
            }
        }
        return true;
    }
};
```

Metodos

Método	Descripción
`Trie()`	Constructor que crea un Trie vacío.
`void insertar(string palabra)`	Inserta una palabra en el Trie.
`bool buscar(string palabra)`	Busca una palabra en el Trie. Devuelve `true` si la palabra está en el Trie y `false` en caso contrario.
`void borrar(string palabra)`	Elimina una palabra del Trie.
`void imprimir()`	Imprime todas las palabras almacenadas en el Trie.
`void limpiar()`	Elimina todas las palabras del Trie, dejándolo vacío.

Permutaciones

3. **Permutaciones**:

Las permutaciones representan el número de formas en que se pueden ordenar (k) elementos de un conjunto de (n) elementos, teniendo en cuenta el orden.

- Fórmula de permutaciones sin repetición: (P(n, k) = \frac{{n!}}{{(n-k)!}})
- Fórmula de permutaciones con repetición: (P(n_1, n_2, \ldots, n_k) = \frac{{n!}}{{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}}), donde (n_1, n_2, \ldots, n_k) son el número de elementos idénticos en cada categoría.

```
unsigned long long factorial(int n) {
    unsigned long long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

// Función para calcular el número de permutaciones de un conjunto
unsigned long long calcularPermutaciones(int n) {
    return factorial(n);
}


vector<vector<int>> getPermutations(const vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> currentPerm = nums;

    do {
        result.push_back(currentPerm);
    } while (next_permutation(currentPerm.begin(), currentPerm.end()));

    return result;
}
```

Combinaciones

Las combinaciones representan el número de formas en que se pueden seleccionar (k) elementos de un conjunto de (n) elementos, sin tener en cuenta el orden.

- Fórmula: (C(n, k) = \binom{n}{k} = \frac{{n!}}{{k!(n-k)!}}), donde (n!) representa el factorial de (n).

```
unsigned long long calcularCombinaciones(int n, int k) {
    unsigned long long numerador = factorial(n);
    unsigned long long denominador = factorial(k) * factorial(n - k);
    return numerador / denominador;
}

void getCombinationsUtil(const vector<int>& nums, int startIndex, int k,
vector<int>& currentComb,
                         vector<vector<int>>& result) {
    if (k == 0) {
        result.push_back(currentComb);
        return;
    }

    for (int i = startIndex; i <= nums.size() - k; i++) {
        currentComb.push_back(nums[i]);
```

```
        getCombinationsUtil(nums, i + 1, k - 1, currentComb, result);
        currentComb.pop_back();
    }
}

vector<vector<int>> getCombinations(const vector<int>& nums, int k) {
    vector<vector<int>> result;
    vector<int> currentComb;
    getCombinationsUtil(nums, 0, k, currentComb, result);
    return result;
}
```

Algebra Lineal

Eliminación Gausseana

```
vector<vector<double>> eliminacionGaussiana(vector<vector<double>>& M) {
    int n = M.size();
    int m = M[0].size();

    for (int i = 0; i < n; i++) {
        // Encontrar el pivote máximo en la columna actual
        double maximo = abs(M[i][i]);
        int filaMaximo = i;
        for (int k = i + 1; k < n; k++) {
            if (abs(M[k][i]) > maximo) {
                maximo = abs(M[k][i]);
                filaMaximo = k;
            }
        }

        // Intercambiar filas si es necesario para tener el pivote máximo en la
fila actual
        if (filaMaximo != i) {
            for (int j = 0; j < m; j++) {
                swap(M[i][j], M[filaMaximo][j]);
            }
        }

        // Hacer ceros debajo del pivote
        for (int k = i + 1; k < n; k++) {
            double factor = -M[k][i] / M[i][i];
            for (int j = i; j < m; j++) {
                M[k][j] += factor * M[i][j];
            }
        }
    }
    return M;
}
```

```
int main() {
    vector<vector<double>> matriz = {
        {2, 1, -1, 8},
        {-3, -1, 2, -11},
        {-2, 1, 2, -3}
    };

    vector<vector<double>> M = eliminacionGaussiana(matriz);

    return 0;
}
```

Validar Solución

```
bool tieneSolucion(vector<vector<double>>& matriz) {
    int filas = matriz.size();
    int columnas = matriz[0].size();
    for (int i = 0; i < filas; i++) {
        bool filaCeros = true;
        for (int j = 0; j < columnas - 1; j++) {
            if (matriz[i][j] != 0) {
                filaCeros = false;
                break;
            }
        }
        if (filaCeros && matriz[i][columnas - 1] != 0) return false;
    }
    return true;
}
```

Obtener soluciones

```
vector<double> obtenerSoluciones(vector<vector<double>>& matriz) {
    int n = matriz.size();
    int m = matriz[0].size();
    vector<double> solucion(n, 0.0);

    for (int i = n - 1; i >= 0; i--) {
        double suma = 0.0;

        for (int j = i + 1; j < m - 1; j++) {
            suma += matriz[i][j] * solucion[j];
        }

        solucion[i] = (matriz[i][m - 1] - suma) / matriz[i][i];
    }
```

```
    return solucion;
}
```

Transformación Lineal

```
vector<vector<double>> multiplicarMatrices(const vector<vector<double>>& matrizA,
const vector<vector<double>>& matrizB) {
    int filasA = matrizA.size();
    int columnasA = matrizA[0].size();
    int filasB = matrizB.size();
    int columnasB = matrizB[0].size();

    if (columnasA != filasB) {//ERROR
        return vector<vector<double>>(); // Devolver una matriz vacía
    }

    vector<vector<double>> resultado(filasA, vector<double>(columnasB, 0.0));

    for (int i = 0; i < filasA; i++) {
        for (int j = 0; j < columnasB; j++) {
            for (int k = 0; k < columnasA; k++) {
                resultado[i][j] += matrizA[i][k] * matrizB[k][j];
            }
        }
    }

    return resultado;
}

int main() {
    vector<vector<double>> matrizA = {
        {2, 3, 1},
        {4, 5, 2},
        {1,0,1}
    };

    vector<vector<double>> matrizB = {
        {1},
        {3},
        {2}
    };

    vector<vector<double>> resultado = multiplicarMatrices(matrizA, matrizB);

    if (!resultado.empty()) {
        cout << "Resultado de la multiplicación:" << endl;
        for (int i = 0; i < resultado.size(); i++) {
            for (int j = 0; j < resultado[i].size(); j++) {
                cout << resultado[i][j] << " ";
            }
```

```
            cout << endl;
        }
    }

    return 0;
}
```

Exponenciación de Matrices

```
typedef vector<vector<double>> Matrix;

Matrix matrixMultiply(const Matrix& A, const Matrix& B) {
    int n = A.size();
    int m = B[0].size();
    int p = B.size();
    Matrix result(n,vector<double>(m, 0.0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            for (int k = 0; k < p; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

Matrix matrixPower(const Matrix& A, int n) {
    if (n == 0) {
        int size = A.size();
        Matrix identity(size,vector<double>(size, 0.0));
        for (int i = 0; i < size; i++) {
            identity[i][i] = 1.0;
        }
        return identity;
    } else if (n % 2 == 0) {
        Matrix halfPower = matrixPower(A, n / 2);
        return matrixMultiply(halfPower, halfPower);
    } else {
        Matrix halfPower = matrixPower(A, (n - 1) / 2);
        Matrix squaredHalfPower = matrixMultiply(halfPower, halfPower);
        return matrixMultiply(A, squaredHalfPower);
    }
}

int main() {
    Matrix A = {{1, 2}, {3, 4}};

    Matrix result = matrixPower(A, 3);
```

```
    for (const auto& row : result) {
        for (double value : row) {
            cout << value << " ";
        }cout << std::endl;
    }

    return 0;
}
```

Algoritmo de Mo

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int block_size;

// Estructura de datos que almacena los valores y permite realizar las operaciones
struct DataStructure {
    vector<int> data;

    void add(int idx) {
        // TODO: Implementar la lógica para agregar un valor en el índice idx
    }

    void remove(int idx) {
        // TODO: Implementar la lógica para eliminar un valor en el índice idx
    }

    int get_answer() {
        // TODO: Implementar la lógica para calcular la respuesta actual de la
estructura de datos
        return 0; // Cambiar 0 por el valor correcto
    }
};

vector<int> mo_s_algorithm(vector<Query> queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());

    DataStructure ds;

    int cur_l = 0;
    int cur_r = -1;

    for (Query q : queries) {
        while (cur_l > q.l) {
            cur_l--;
```

```
            ds.add(cur_l);
        }
        while (cur_r < q.r) {
            cur_r++;
            ds.add(cur_r);
        }
        while (cur_l < q.l) {
            ds.remove(cur_l);
            cur_l++;
        }
        while (cur_r > q.r) {
            ds.remove(cur_r);
            cur_r--;
        }
        answers[q.idx] = ds.get_answer();
    }
    return answers;
}

int main() {
    // Ejemplo de uso del algoritmo de Mo's
    block_size = 3; // Tamaño del bloque, ajusta según tus necesidades

    vector<Query> queries = {
        {0, 4, 0},
        {2, 6, 1},
        {1, 5, 2},
    };

    vector<int> results = mo_s_algorithm(queries);

    for (int answer : results) {
        cout << answer << " ";
    }

    return 0;
}
```

Algoritmo de Meet in the Middle

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

// Función que implementa la parte "izquierda" del algoritmo
void generateSubsetsLeft(vector<int>& nums, int index, int sum, map<int, int>&
leftSumCounts) {
    if (index == nums.size() / 2) {
```

```
        leftSumCounts[sum]++;
        return;
    }

    // Incluir el elemento actual en el subconjunto
    generateSubsetsLeft(nums, index + 1, sum + nums[index], leftSumCounts);

    // No incluir el elemento actual en el subconjunto
    generateSubsetsLeft(nums, index + 1, sum, leftSumCounts);
}

// Función que implementa la parte "derecha" del algoritmo
int countSubsetsRight(vector<int>& nums, int index, int sum, map<int, int>&
leftSumCounts) {
    if (index == nums.size()) {
        int complement = -sum; // Calcula el complemento necesario para alcanzar
el objetivo
        if (leftSumCounts.find(complement) != leftSumCounts.end()) {
            return leftSumCounts[complement];
        }
        return 0;
    }

    // Incluir el elemento actual en el subconjunto
    int includeCount = countSubsetsRight(nums, index + 1, sum + nums[index],
leftSumCounts);

    // No incluir el elemento actual en el subconjunto
    int excludeCount = countSubsetsRight(nums, index + 1, sum, leftSumCounts);

    return includeCount + excludeCount;
}

int main() {
    vector<int> nums = {1, 2, 3, 4, 5};
    int targetSum = 6;

    map<int, int> leftSumCounts;
    generateSubsetsLeft(nums, 0, 0, leftSumCounts);

    int totalSubsets = countSubsetsRight(nums, nums.size() / 2, 0, leftSumCounts);

    cout << "Número de subconjuntos con suma igual a " << targetSum << ": " <<
totalSubsets << endl;

    return 0;
}
```

Transformada Rapida de Fourier

```
void fft(vector<complex<double>>& a, bool invertir) {
    int n = a.size();
    if (n <= 1) return;

    vector<complex<double>> a0(n / 2), a1(n / 2);
    for (int i = 0, j = 0; i < n; i += 2, ++j) {
        a0[j] = a[i];
        a1[j] = a[i + 1];
    }

    fft(a0, invertir);
    fft(a1, invertir);

    double ang = 2 * M_PI / n * (invertir ? -1 : 1);
    complex<double> w(1), wn(cos(ang), sin(ang));
    for (int i = 0; i < n / 2; ++i) {
        complex<double> t = w * a1[i];
        a[i] = a0[i] + t;
        a[i + n / 2] = a0[i] - t;
        if (invertir) {
            a[i] /= 2;
            a[i + n / 2] /= 2;
        }
        w *= wn;
    }
}

// Función para multiplicar dos polinomios utilizando FFT
vector<int> multiplicar_polinomios(const vector<int>& polinomio1, const
vector<int>& polinomio2) {
    int n = 1;
    while (n < polinomio1.size() + polinomio2.size()) {
        n <<= 1;
    }

    vector<complex<double>> a(n), b(n);
    for (int i = 0; i < polinomio1.size(); ++i) {
        a[i] = polinomio1[i];
    }
    for (int i = 0; i < polinomio2.size(); ++i) {
        b[i] = polinomio2[i];
    }

    fft(a, false);
    fft(b, false);

    for (int i = 0; i < n; ++i) {
        a[i] *= b[i];
    }

    fft(a, true);

    vector<int> resultado(n);
```

```
    for (int i = 0; i < n; ++i) {
        resultado[i] = round(a[i].real());
    }

    return resultado;
}

// Función para evaluar un polinomio en un punto específico
int evaluar_polinomio(const vector<int>& polinomio, int x) {
    int resultado = 0;
    int n = polinomio.size();
    for (int i = n - 1; i >= 0; --i) {
        resultado = resultado * x + polinomio[i];
    }
    return resultado;
}

int main() {
    vector<int> polinomio1 = {1, 2, 3};  // Representa x^2 + 2x + 3
    vector<int> polinomio2 = {4, 5};     // Representa 4x + 5

    vector<int> resultado = multiplicar_polinomios(polinomio1, polinomio2);

    for (int coeficiente : resultado) {
        cout << coeficiente << " ";
    }
    cout << endl;

    int x = 2;
    int resultado_evaluacion = evaluar_polinomio(resultado, x);
    cout << "El polinomio evaluado en x=" << x << " es igual a " <<
resultado_evaluacion << endl;

    return 0;
}
```

Algoritmos de Strings

Knuth-Morris-Pratt KMP

Busqueda de patrones en String en O(n + m)

```
vector<int> computeLPSArray(const string& pattern) {
    int m = pattern.length();
    vector<int> lps(m, 0);
    int len = 0;

    for (int i = 1; i < m; i++) {
        while (len > 0 && pattern[i] != pattern[len]) len = lps[len - 1];
        if (pattern[i] == pattern[len]) len++;
```

```
            lps[i] = len;
        }

        return lps;
    }

    vector<int> KMPSearch(const string& text, const string& pattern) {
        vector<int> indices, lps = computeLPSArray(pattern);
        int n = text.length(), m = pattern.length(), i = 0, j = 0;

        while (i < n) {
            if (pattern[j] == text[i]) i++, j++;
            if (j == m) indices.push_back(i - j), j = lps[j - 1];
            else if (i < n && pattern[j] != text[i]) j = j ? lps[j - 1] : i++;
        }

        return indices;
    }
```

Algoritmo de Manacher

Devuelve el substring palindromo más grande

```
    string manacher(const string& s) {
        string T = "^#";
        for (char c : s) {
            T += c;
            T += '#';
        }
        T += '$';

        int n = T.size();
        vector<int> P(n, 0);
        int C = 0, R = 0;

        for (int i = 1; i < n - 1; i++) {
            int i_mirror = 2 * C - i;
            if (R > i) {
                P[i] = min(R - i, P[i_mirror]);
            }

            while (T[i + 1 + P[i]] == T[i - 1 - P[i]]) {
                P[i]++;
            }

            if (i + P[i] > R) {
                C = i;
                R = i + P[i];
            }
        }
```

```
    int maxLen = 0;
    int centerIndex = 0;
    for (int i = 1; i < n - 1; i++) {
        if (P[i] > maxLen) {
            maxLen = P[i];
            centerIndex = i;
        }
    }

    int start = (centerIndex - maxLen) / 2;
    return s.substr(start, maxLen);
}
```

Longest Common Subsequence

```
string findLCS(const string& X, const string& Y) {
    int m = X.length();
    int n = Y.length();

    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    int lcs_length = dp[m][n];
    string lcs(lcs_length, ' ');

    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs[--lcs_length] = X[i - 1];
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    return lcs;
}
```

Fenwick Tree

Un Fenwick Tree (también conocido como Binary Indexed Tree o BIT) es una estructura de datos que se utiliza para realizar consultas y actualizaciones eficientes en una secuencia de valores numéricos, como sumas acumulativas.

```
class FenwickTree {
private:
    std::vector<int> tree;
    int size;

public:
    FenwickTree(int n) {
        size = n;
        tree.assign(n + 1, 0);
    }

    // Añade 'delta' al elemento en la posición 'index'
    void update(int index, int delta) {
        index++; // Aumentar en 1 porque BIT es 1-indexado
        while (index <= size) {
            tree[index] += delta;
            index += index & -index; // Avanzar al siguiente elemento en el árbol
        }
    }

    // Devuelve la suma acumulativa de los primeros 'index' elementos
    int query(int index) {
        index++; // Aumentar en 1 porque BIT es 1-indexado
        int result = 0;
        while (index > 0) {
            result += tree[index];
            index -= index & -index; // Retroceder al elemento anterior en el
árbol
        }
        return result;
    }
};

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5, 6, 7};
    int n = arr.size();

    FenwickTree fenwick(n);

    // Construir el Fenwick Tree
    for (int i = 0; i < n; i++) {
        fenwick.update(i, arr[i]);
    }

    // Consultar la suma acumulativa de los primeros k elementos
```

```
    int k = 4;
    int sum = fenwick.query(k);
    std::cout << "La suma acumulativa de los primeros " << k << " elementos es: "
<< sum << std::endl;

    // Actualizar el valor en la posición 2 a 10
    fenwick.update(2, 10);

    // Consultar la suma acumulativa de los primeros k elementos después de la
actualización
    int newSum = fenwick.query(k);
    std::cout << "La suma acumulativa de los primeros " << k << " elementos
después de la actualización es: " << newSum << std::endl;

    return 0;
}
```

Spanning Tree

Un Sparse Table es una estructura de datos que permite realizar consultas eficientes en un rango de valores en un arreglo, especialmente para operaciones idempotentes como mínimo, máximo, suma, etc.

Método	Descripción
`SparseTable(values, op)`	Constructor que inicializa la tabla con un arreglo y una operación.
`query(l, r)`	Realiza una consulta en el rango [l, r] según la operación especificada.
`gcd(a, b)`	Función auxiliar para calcular el máximo común divisor.
`sumQuery(l, r)`	Realiza una consulta de suma en el rango [l, r].
`multQuery(l, r)`	Realiza una consulta de multiplicación en el rango [l, r].
`calcLog2(len)`	Calcula el logaritmo base 2 de un número entero `len`.

```
class SparseTable {
public:
    enum Operation {
        MIN,
        MAX,
        SUM,
        MULT,
        GCD
    };

    SparseTable(vector<long>& values, Operation op) : op(op) {
        init(values);
    }

    long query(int l, int r) {
        if (op == Operation::MIN) {
```

```
            return query(l, r, [](long a, long b) { return min(a, b); });
        } else if (op == Operation::MAX) {
            return query(l, r, [](long a, long b) { return max(a, b); });
        } else if (op == Operation::GCD) {
            return query(l, r, [this](long a, long b) { return gcd(a, b); });
        }

        if (op == Operation::SUM) {
            return sumQuery(l, r);
        } else {
            return multQuery(l, r);
        }
    }

private:
    int n;
    int P;
    vector<vector<long>> dp;
    vector<vector<int>> it;
    Operation op;

    void init(vector<long>& v) {
        n = v.size();
        P = static_cast<int>(log2(n));
        dp = vector<vector<long>>(P + 1, vector<long>(n));
        it = vector<vector<int>>(P + 1, vector<int>(n));

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

        for (int i = 1; i <= P; i++) {
            for (int j = 0; j + (1 << i) <= n; j++) {
                long leftInterval = dp[i - 1][j];
                long rightInterval = dp[i - 1][j + (1 << (i - 1))];
                if (op == Operation::MIN) {
                    dp[i][j] = min(leftInterval, rightInterval);
                    if (leftInterval <= rightInterval) {
                        it[i][j] = it[i - 1][j];
                    } else {
                        it[i][j] = it[i - 1][j + (1 << (i - 1))];
                    }
                } else if (op == Operation::MAX) {
                    dp[i][j] = max(leftInterval, rightInterval);
                    if (leftInterval >= rightInterval) {
                        it[i][j] = it[i - 1][j];
                    } else {
                        it[i][j] = it[i - 1][j + (1 << (i - 1))];
                    }
                } else if (op == Operation::SUM) {
                    dp[i][j] = leftInterval + rightInterval;
                } else if (op == Operation::MULT) {
                    dp[i][j] = leftInterval * rightInterval;
```

```
                } else if (op == Operation::GCD) {
                    dp[i][j] = gcd(leftInterval, rightInterval);
                }
            }
        }
    }

    long gcd(long a, long b) {
        while (b != 0) {
            long temp = b;
            b = a % b;
            a = temp;
        }
        return abs(a);
    }

    long sumQuery(int l, int r) {
        long sum = 0;
        for (int p = calcLog2(r - l + 1); l <= r; p = calcLog2(r - l + 1)) {
            sum += dp[p][l];
            l += (1 << p);
        }
        return sum;
    }

    long multQuery(int l, int r) {
        long result = 1;
        for (int p = calcLog2(r - l + 1); l <= r; p = calcLog2(r - l + 1)) {
            result *= dp[p][l];
            l += (1 << p);
        }
        return result;
    }

    long query(int l, int r, function<long(long, long)> fn) {
        int len = r - l + 1;
        int p = calcLog2(len);
        return fn(dp[p][l], dp[p][r - (1 << p) + 1]);
    }

    int calcLog2(int len) {
        return static_cast<int>(log2(len));
    }
};

int main() {
    vector<long> values = {4, 4, 4, 4, 4, 4};
    SparseTable sparseTable(values, SparseTable::Operation::SUM);
    cout << sparseTable.query(0, values.size() - 1) << endl;
    return 0;
}
```

Knapsack

```
struct Item {
    int weight;
    int value;
};

// Función para resolver el problema de la mochila
int knapsack(vector<Item>& items, int capacity) {
    int n = items.size();
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            if (items[i - 1].weight <= w) {
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - items[i - 1].weight] +
items[i - 1].value);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][capacity];
}

int main() {
    int capacity = 10; // Capacidad de la mochila
    vector<Item> items = {
        {2, 6},
        {2, 10},
        {3, 12},
        {4, 13},
        {5, 15}
    };

    int max_value = knapsack(items, capacity);
    cout << "El valor máximo que se puede obtener es: " << max_value << endl;

    return 0;
}
```

Rootaear un arbol desde el centro

```
#include <iostream>
#include <vector>
```

```cpp
using namespace std;

vector<int> treeCenters(vector<vector<int>>& g) {
    int n = g.size();
    vector<int> degree(n, 0);
    vector<int> leaves;

    for (int i = 0; i < n; i++) {
        degree[i] = g[i].size();
        if (degree[i] == 0 || degree[i] == 1) {
            leaves.push_back(i);
            degree[i] = 0;
        }
    }

    int count = leaves.size();

    while (count < n) {
        vector<int> new_leaves;
        for (int node : leaves) {
            for (int neighbor : g[node]) {
                degree[neighbor] = degree[neighbor] - 1;
                if (degree[neighbor] == 1) {
                    new_leaves.push_back(neighbor);
                }
                degree[node] = 0;
            }
        }
        count += new_leaves.size();
        leaves = new_leaves;
    }

    return leaves; // Centros (center) del árbol
}

void dfs(vector<vector<int>>& g, vector<bool>& visited, vector<vector<int>>&
rootedTree, int node) {
    if (!visited[node]) {
        visited[node] = true;
        for (int neighbor : g[node]) {
            if (!visited[neighbor]) {
                rootedTree[node].push_back(neighbor);
                dfs(g, visited, rootedTree, neighbor);
            }
        }
    }
}

int main() {
    // Definir el grafo no dirigido como una lista de adyacencia
    int n = 6;
    vector<vector<int>> graph(n);

    // Agregar las conexiones del grafo
```

```
    graph[0].push_back(1);
    graph[1].push_back(0);
    graph[1].push_back(2);
    graph[2].push_back(1);
    graph[2].push_back(3);
    graph[3].push_back(2);
    graph[3].push_back(4);
    graph[4].push_back(3);
    graph[4].push_back(5);
    graph[5].push_back(4);

    vector<int> centers = treeCenters(graph);

    cout << "Centro(s) del árbol: ";
    for (int center : centers) {
        cout << center << " ";
    }
    cout << endl;

    // Elegir el primer centro como raíz
    int root = centers[0];

    vector<vector<int>> rootedTree(n);
    vector<bool> visited(n, false);

    dfs(graph, visited, rootedTree, root);

    cout << "Grafo enraizado desde el centro:" << endl;
    for (int i = 0; i < n; i++) {
        cout << i << ": ";
        for (int neighbor : rootedTree[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Identificar dos arboles isomorficos

```
using namespace std;

// Función para realizar un recorrido DFS desde el nodo raíz central
void DFS(int u, int parent, vector<vector<int>>& graph, vector<int>& code) {
    code.push_back(1);  // Etiqueta de apertura
    for (int v : graph[u]) {
        if (v != parent) {
            DFS(v, u, graph, code);
```

```
        }
    }
    code.push_back(0);  // Etiqueta de cierre
}

// Función para generar el código AHU de un grafo en forma de lista de adyacencia
vector<int> AHU_Code(vector<vector<int>>& graph, int root) {
    vector<int> code;
    DFS(root, -1, graph, code);
    return code;
}

// Función para verificar si dos códigos AHU son iguales
bool areCodesEqual(const vector<int>& code1, const vector<int>& code2) {
    return code1 == code2;
}

int main() {
    // Definir los grafos como listas de adyacencia
    vector<vector<int>> graph1 = {{1, 2}, {0, 3}, {0, 3}, {1, 2}};
    vector<vector<int>> graph2 = {{1, 2, 3}, {0, 4}, {0, 4}, {0, 4}, {1, 2, 3}};

    // Supongamos que el nodo raíz central es el nodo 0 en ambos grafos
    int root = 0;

    // Generar códigos AHU para ambos grafos
    vector<int> code1 = AHU_Code(graph1, root);
    vector<int> code2 = AHU_Code(graph2, root);

    // Verificar si los códigos son iguales
    if (areCodesEqual(code1, code2)) {
        cout << "Los grafos son isomorfos." << endl;
    } else {
        cout << "Los grafos no son isomorfos." << endl;
    }

    return 0;
}
```

Lowest Common Ancestor

```
class LCA {
public:
    LCA(vector<vector<int>>& graph, int root) : N(graph.size()), parent(N,
vector<int>(MAX_DEPTH, -1)), depth(N, 0) {
        dfs(graph, root, -1, 0);
        precompute();
    }
```

```
    int findLCA(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);
        for (int i = MAX_DEPTH - 1; i >= 0; i--) {
            if (depth[u] - (1 << i) >= depth[v]) u = parent[u][i];
        }
        if (u == v) return u;
        for (int i = MAX_DEPTH - 1; i >= 0; i--) {
            if (parent[u][i] != parent[v][i]) {
                u = parent[u][i];
                v = parent[v][i];
            }
        }
        return parent[u][0];
    }

private:
    const int MAX_DEPTH = 16; // Ajusta este valor según tus necesidades
    int N;
    vector<vector<int>> parent;
    vector<int> depth;

    void dfs(vector<vector<int>>& graph, int u, int par, int d) {
        parent[u][0] = par;
        depth[u] = d;
        for (int v : graph[u]) {
            if (v != par) dfs(graph, v, u, d + 1);
        }
    }

    void precompute() {
        for (int j = 1; j < MAX_DEPTH; j++) {
            for (int i = 0; i < N; i++) {
                if (parent[i][j - 1] != -1) {
                    parent[i][j] = parent[parent[i][j - 1]][j - 1];
                }
            }
        }
    }
};

int main() {
    int N = 9;  // Número de nodos en el árbol
    vector<vector<int>> graph(N);

    // Definir las aristas del árbol
    graph[0] = {1, 2};
    graph[1] = {3, 4};
    graph[2] = {5};
    graph[4] = {6, 7};
    graph[6] = {8};

    LCA lca(graph, 0);

    int u = 5;
```

```
    int v = 7;

    int lcaNode = lca.findLCA(u, v);
    cout << "El LCA de " << u << " y " << v << " es el nodo " << lcaNode << endl;

    return 0;
}
```

Algoritmo de Trajan para Componentes Fuertemente Conexos

```
#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>

using namespace std;

void tarjan(int u, vector<vector<int>>& grafo, vector<int>& ids, vector<int>&
bajos, stack<int>& pila, vector<bool>& enPila, int& idActual, vector<vector<int>>&
componentes) {
    ids[u] = bajos[u] = idActual++;
    pila.push(u);
    enPila[u] = true;

    for (int v : grafo[u]) {
        if (ids[v] == -1) {
            tarjan(v, grafo, ids, bajos, pila, enPila, idActual, componentes);
        }
        if (enPila[v]) {
            bajos[u] = min(bajos[u], bajos[v]);
        }
    }

    if (ids[u] == bajos[u]) {
        vector<int> componente;
        while (true) {
            int nodo = pila.top();
            pila.pop();
            enPila[nodo] = false;
            componente.push_back(nodo);
            if (nodo == u) {
                break;
            }
        }
        componentes.push_back(componente);
    }
}
```

```
vector<vector<int>> encontrarComponentesFuertementeConexos(vector<vector<int>>&
grafo) {
    int n = grafo.size();
    vector<int> ids(n, -1);
    vector<int> bajos(n, -1);
    vector<bool> enPila(n, false);
    stack<int> pila;
    int idActual = 0;
    vector<vector<int>> componentes;

    for (int i = 0; i < n; i++) {
        if (ids[i] == -1) {
            tarjan(i, grafo, ids, bajos, pila, enPila, idActual, componentes);
        }
    }

    return componentes;
}

int main() {
    int V = 5;
    vector<vector<int>> grafo(V);

    grafo[0].push_back(1);
    grafo[1].push_back(2);
    grafo[2].push_back(0);

    grafo[3].push_back(4);

    vector<vector<int>> componentes =
encontrarComponentesFuertementeConexos(grafo);

    cout << "Componentes fuertemente conexos:" << endl;
    for (const vector<int>& componente : componentes) {
        for (int nodo : componente) {
            cout << nodo << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Problema del Agente Viajero DP O($n^2 2^n$)

```
const int INF = INT_MAX;

// Función para calcular el costo mínimo y el camino óptimo
int tsp(vector<vector<int>>& graph, vector<vector<int>>& dp, int mask, int pos,
int n) {
```

```
    if (mask == (1 << n) - 1) {
        return graph[pos][0]; // Todos los nodos han sido visitados
    }

    if (dp[mask][pos] != -1) {
        return dp[mask][pos]; // Ya calculado antes
    }

    int minCost = INF;

    for (int next = 0; next < n; next++) {
        if ((mask & (1 << next)) == 0) { // Si el siguiente nodo no ha sido
visitado
            int newCost = graph[pos][next] + tsp(graph, dp, mask | (1 << next),
next, n);
            minCost = min(minCost, newCost);
        }
    }

    return dp[mask][pos] = minCost;
}

// Función para reconstruir el camino óptimo
vector<int> reconstructPath(vector<vector<int>>& dp, vector<vector<int>>& graph,
int n) {
    vector<int> path;
    int mask = 1;
    int pos = 0;

    for (int i = 0; i < n - 1; i++) {
        int next = -1;
        for (int j = 1; j < n; j++) {
            if ((mask & (1 << j)) == 0 && (next == -1 || dp[mask][pos] -
graph[pos][next] == dp[mask | (1 << j)][j])) {
                next = j;
            }
        }
        path.push_back(next);
        mask |= (1 << next);
        pos = next;
    }

    return path;
}

int main() {
    int n = 4;
    vector<vector<int>> graph = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };
```

```
    vector<vector<int>> dp(1 << n, vector<int>(n, -1));

    int minCost = tsp(graph, dp, 1, 0, n);
    vector<int> optimalPath = reconstructPath(dp, graph, n);

    cout << "Costo mínimo del TSP: " << minCost << endl;
    cout << "Camino óptimo: 0 ";
    for (int node : optimalPath) {
        cout << node << " ";
    }
    cout << "0" << endl;

    return 0;
}
```

Maximo Flujo Ford-Fulkerson

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

const int INF = 1e9;

// Definición de una arista en la red
struct Edge {
    int to, capacity, flow;
};

// Función para agregar una arista en ambas direcciones
void addEdge(vector<vector<Edge>>& graph, int from, int to, int capacity) {
    Edge e1 = {to, capacity, 0};
    Edge e2 = {from, 0, 0};
    graph[from].push_back(e1);
    graph[to].push_back(e2);
}

// Función para encontrar el flujo máximo utilizando el algoritmo de Ford-
Fulkerson
int fordFulkerson(vector<vector<Edge>>& graph, int source, int sink) {
    int maxFlow = 0;
    while (true) {
        // Inicializar un vector para realizar un seguimiento de los nodos
visitados
        vector<int> parent(graph.size(), -1);

        queue<pair<int, int>> q;
        q.push({source, INF});
```

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

            for (const Edge& e : graph[cur]) {
                if (parent[e.to] == -1 && e.capacity - e.flow > 0) {
                    parent[e.to] = cur;
                    int newFlow = min(flow, e.capacity - e.flow);
                    if (e.to == sink) {
                        maxFlow += newFlow;
                        int v = e.to;
                        while (v != source) {
                            int u = parent[v];
                            for (Edge& edge : graph[u]) {
                                if (edge.to == v) {
                                    edge.flow += newFlow;
                                    break;
                                }
                            }
                            for (Edge& edge : graph[v]) {
                                if (edge.to == u) {
                                    edge.flow -= newFlow;
                                    break;
                                }
                            }
                            v = u;
                        }
                        goto found;
                    }
                    q.push({e.to, newFlow});
                }
            }
        }
        break;
        found: ;
    }

    return maxFlow;
}

int main() {
    int numNodes, numEdges;
    cin >> numNodes >> numEdges;

    vector<vector<Edge>> graph(numNodes);

    for (int i = 0; i < numEdges; i++) {
        int from, to, capacity;
        cin >> from >> to >> capacity;
        addEdge(graph, from, to, capacity);
    }
```

```
    int source, sink;
    cin >> source >> sink;

    int maxFlow = fordFulkerson(graph, source, sink);

    cout << "Flujo Máximo: " << maxFlow << endl;

    return 0;
}
```

Exponenciacion Binaria de Matrices

```
typedef vector<vector<int>> Matrix;

// Función para multiplicar dos matrices
Matrix matrixMultiply(const Matrix &a, const Matrix &b) {
    int n = a.size();
    Matrix result(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }

    return result;
}

// Función para elevar una matriz a una potencia n usando exponenciación binaria
Matrix matrixPower(const Matrix &matrix, int n) {
    int size = matrix.size();
    Matrix result(size, vector<int>(size, 0));

    // Inicializar la matriz de resultado como la matriz identidad
    for (int i = 0; i < size; i++) {
        result[i][i] = 1;
    }

    Matrix base = matrix;

    while (n > 0) {
        if (n % 2 == 1) {
            result = matrixMultiply(result, base);
        }
        base = matrixMultiply(base, base);
        n /= 2;
    }
```

```
    return result;
}
```

Matemáticas

Probabliidad

1. **Probabilidad**:

- La probabilidad de un evento (A), denotada como (P(A)), se define como el número de resultados favorables al evento dividido por el número total de resultados posibles.
    - Fórmula: (P(A) = \frac{{\text{{Número de resultados favorables}}}}{{\text{{Número total de resultados posibles}}}})

1. **Probabilidad Total**:

    - Para eventos (A_1, A_2, \ldots, A_n) que forman una partición del espacio muestral, la probabilidad de un evento (B) se puede calcular como: [P(B) = \sum_{i=1}^{n} P(A_i) \cdot P(B|A_i)]

2. **Teorema de Bayes**:

    - Permite calcular la probabilidad condicional inversa. Para dos eventos (A) y (B) con (P(B) > 0), se puede calcular la probabilidad de (A) dado (B) como: [P(A|B) = \frac{{P(B|A) \cdot P(A)}}{{P(B)}}]

3. **Regla del Producto**:

    - La probabilidad de la intersección de dos eventos (A) y (B) se puede calcular como el producto de la probabilidad de (A) dado (B) y la probabilidad de (B) como: [P(A \cap B) = P(A|B) \cdot P(B)]

4. **Probabilidad Conjunta**:

    - La probabilidad de que dos eventos (A) y (B) ocurran simultáneamente se puede denotar como (P(A \cap B)).

5. **Independencia de Eventos**:

    - Dos eventos (A) y (B) son independientes si y solo si la probabilidad de que ocurra (A) dado (B) es igual a la probabilidad de (A), y viceversa: [P(A|B) = P(A) \text{ y } P(B|A) = P(B)]

6. **Probabilidad Condicional**:

    - La probabilidad de que ocurra un evento (A) dado que ya ha ocurrido un evento (B) se denota como (P(A|B)).

7. **Regla de la Suma**:

- La probabilidad de la unión de dos eventos (A) y (B) se puede calcular como la suma de las probabilidades de cada evento menos la probabilidad de su intersección: [P(A \cup B) = P(A) + P(B) - P(A \cap B)]

Sucesiones

1. **Números de Fibonacci**: Esta sucesión comienza con 0 y 1, y cada término posterior es la suma de los dos términos anteriores.
   - Fórmula: (F(n) = F(n-1) + F(n-2)) con (F(0) = 0) y (F(1) = 1).
2. **Números de Catalan**: Estos números surgen en diversos problemas de combinatoria y se definen de la siguiente manera:
   - Fórmula: (C(n) = \frac{{(2n)!}}{{(n+1)!n!}}) con (C(0) = 1).
3. **Números Triangulares**: La secuencia de números triangulares se forma al sumar los números naturales consecutivos.
   - Fórmula: (T(n) = \frac{{n(n+1)}}{2}) con (T(1) = 1).
4. **Números Cuadrados**: Esta sucesión representa los cuadrados de los números naturales.
   - Fórmula: (S(n) = n^2) con (S(1) = 1).
5. **Números Primos**: La secuencia de números primos son aquellos que solo son divisibles por 1 y por sí mismos.
   - No hay una fórmula cerrada general para los números primos.
6. **Números Triangulares Centrales**: Estos números surgen en la geometría de los polígonos y se relacionan con los números triangulares.
   - Fórmula: (TC(n) = \frac{{n(n+1)(2n+1)}}{6}) con (TC(1) = 1).
7. **Números de Lucas**: Similar a la sucesión de Fibonacci pero con diferentes valores iniciales.
   - Fórmula: (L(n) = L(n-1) + L(n-2)) con (L(0) = 2) y (L(1) = 1).
8. **Números de Pell**: Otra secuencia similar a Fibonacci, pero con diferentes valores iniciales.
   - Fórmula: (P(n) = 2P(n-1) + P(n-2)) con (P(0) = 0) y (P(1) = 1).
9. **Números de Mersenne**: Estos son números enteros que son una unidad menos que una potencia de dos.
   - Fórmula: (M_n = 2^n - 1) donde (n) es un número primo.
10. **Números de Stirling de Segundo Tipo**: Estos números cuentan el número de formas distintas de dividir un conjunto de (n) elementos en (k) conjuntos no vacíos.
    - Fórmula: Los números de Stirling de Segundo Tipo tienen una fórmula recursiva.

Uso Angulos

```cpp
#include <iostream>
#include <cmath> // Incluir la cabecera cmath para funciones matemáticas
using namespace std; // Usar el espacio de nombres std

int main() {
    double angle = 45.0; // Ángulo en grados

    // Funciones trigonométricas en radianes
    double radians = angle * M_PI / 180.0;
    double cosineValue = cos(radians);
    double sineValue = sin(radians);

    // Funciones inversas
    double arccosValue = acos(cosineValue); // Devuelve el ángulo en radianes
    double arcsinValue = asin(sineValue); // Devuelve el ángulo en radianes
    double arctanValue = atan(sineValue / cosineValue); // Devuelve el ángulo en
radianes

    // Mostrar resultados
    cout << "Ángulo en grados: " << angle << endl;
    cout << "Coseno: " << cosineValue << endl;
    cout << "Seno: " << sineValue << endl;
    cout << "Arco coseno: " << arccosValue * 180.0 / M_PI << " grados" << endl; //
Convertir a grados
    cout << "Arco seno: " << arcsinValue * 180.0 / M_PI << " grados" << endl; //
Convertir a grados
    cout << "Arco tangente: " << arctanValue * 180.0 / M_PI << " grados" << endl;
// Convertir a grados

    return 0;
}
```

Teorema del residuo chino

```cpp
vector<long> euclidesExtendido(long a, long b) {
    if (b == 0) return {a, 1, 0};
    else {
        vector<long> ret = euclidesExtendido(b, a % b);
        long tmp = ret[1] - ret[2] * (a / b);
        ret[1] = ret[2];
        ret[2] = tmp;
        return ret;
    }
}

long maximoComunDivisor(long a, long b) {
    return b == 0 ? a : maximoComunDivisor(b, a % b);
}

vector<long> eliminarCoeficiente(long c, long a, long m) {
```

```
    vector<long> d = euclidesExtendido(c, m);

    if (a % d[0] != 0) return {0, 0}; // Devolviendo {0, 0} para representar nulo

    c /= d[0];
    a /= d[0];
    m /= d[0];

    vector<long> inv = euclidesExtendido(c, m);
    m = abs(m);
    a = (((a * inv[1]) % m) + m) % m;

    return {a, m};
}

vector<vector<long>> reducir(vector<long> a, vector<long> m) {
    vector<long> aNuevo;
    vector<long> mNuevo;

    for (int i = 0; i < a.size(); i++) {
        vector<long> factores; // Almacena los factores primos aquí
        // Dividir cada ecuación en factores primos
        // (Debes implementar la función de factorización en números primos)

        // Ordenar los factores
        sort(factores.begin(), factores.end());

        for (int j = 0; j < factores.size(); j++) {
            long val = factores[j];
            long total = val;
            while (j + 1 < factores.size() && factores[j + 1] == val) {
                total *= val;
                j++;
            }
            aNuevo.push_back(a[i] % total);
            mNuevo.push_back(total);
        }
    }

    for (int i = 0; i < aNuevo.size(); i++) {
        for (int j = i + 1; j < aNuevo.size(); j++) {
            if (mNuevo[i] % mNuevo[j] == 0 || mNuevo[j] % mNuevo[i] == 0) {
                if (mNuevo[i] > mNuevo[j]) {
                    if ((aNuevo[i] % mNuevo[j]) == aNuevo[j]) {
                        aNuevo.erase(aNuevo.begin() + j);
                        mNuevo.erase(mNuevo.begin() + j);
                        j--;
                        continue;
                    }
                    else return {{0, 0}}; // Devolviendo {0, 0} para representar
nulo
                }
                else {
                    if ((aNuevo[j] % mNuevo[i]) == aNuevo[i]) {
```

```
                        aNuevo.erase(aNuevo.begin() + i);
                        mNuevo.erase(mNuevo.begin() + i);
                        i--;
                        break;
                    }
                    else return {{0, 0}}; // Devolviendo {0, 0} para representar
nulo
                }
            }
        }
    }

    vector<vector<long>> res(2, vector<long>(aNuevo.size()));
    for (int i = 0; i < aNuevo.size(); i++) {
        res[0][i] = aNuevo[i];
        res[1][i] = mNuevo[i];
    }

    return res;
}

vector<long> teoremaRestosChinos(vector<long> a, vector<long> m) {
    long M = 1;
    for (int i = 0; i < m.size(); i++) M *= m[i];

    vector<long> inv(a.size());
    for (int i = 0; i < inv.size(); i++) inv[i] = euclidesExtendido(M / m[i],
m[i])[1];

    long x = 0;
    for (int i = 0; i < m.size(); i++) {
        x += (M / m[i]) * a[i] * inv[i];
        x = ((x % M) + M) % M;
    }

    return {x, M};
}

vector<long> factorizacionPrimos(long n) {
    vector<long> factores;
    if (n <= 0) throw invalid_argument("n debe ser un entero positivo");
    else if (n == 1) return factores;

    // Implementa la lógica de factorización en números primos aquí

    return factores;
}

long pollardRho(long n) {
    if (n % 2 == 0) return 2;
    long x = 2 + rand() % 999999;
    long c = 2 + rand() % 999999;
    long y = x;
    long d = 1;
```

```
    while (d == 1) {
        x = (x * x + c) % n;
        y = (y * y + c) % n;
        y = (y * y + c) % n;
        d = maximoComunDivisor(abs(x - y), n);
        if (d == n) break;
    }
    return d;
}

bool esPrimo(long n) {
    if (n < 2) return false;
    if (n == 2 || n == 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    int limite = static_cast<int>(sqrt(n));

    for (int i = 5; i <= limite; i += 6)
        if (n % i == 0 || n % (i + 2) == 0) return false;

    return true;
}
```

Criba de Erastotenes Comprimida O(n log log n)

```
const double NUM_BITS = 128.0;
const int NUM_BITS_SHIFT = 7; // 2^7 = 128

void setBit(vector<long long>& arr, int n) {
    if ((n & 1) == 0) return; // n es par
    arr[n >> NUM_BITS_SHIFT] |= 1LL << ((n - 1) >> 1);
}

bool isNotSet(const vector<long long>& arr, int n) {
    if (n < 2) return false; // n no es primo
    if (n == 2) return true; // dos es primo
    if ((n & 1) == 0) return false; // n es par
    long long chunk = arr[n >> NUM_BITS_SHIFT];
    long long mask = 1LL << ((n - 1) >> 1);
    return (chunk & mask) != mask;
}

vector<int> primeSieve(int limit) {
    const int numChunks = static_cast<int>(ceil(static_cast<double>(limit) /
NUM_BITS));
    const int sqrtLimit = static_cast<int>(sqrt(static_cast<double>(limit)));
    vector<long long> chunks(numChunks, 0);
    chunks[0] = 1; // 1 no es primo

    vector<int> primes;
```

```
    for (int i = 3; i <= limit; i += 2)
        if (isNotSet(chunks, i))
            primes.push_back(i);

    return primes;
}

int main() {
    const int limit = 200;
    vector<int> primeNumbers = primeSieve(limit);

    for (int prime : primeNumbers) {
        cout << prime << " es primo!" << endl;
    }

    return 0;
}
```

Funcion Toteinica de Euler con Criba

```
vector<int> sieve(int limit) {
    if (limit <= 2) return vector<int>();
    const int numPrimes = (int)(1.25506 * limit / log((double)limit));
    vector<int> primes(numPrimes);
    int index = 0;
    vector<bool> isComposite(limit, false);
    const int sqrtLimit = (int)sqrt(limit);

    for (int i = 2; i <= sqrtLimit; i++) {
        if (!isComposite[i]) {
            primes[index++] = i;
            for (int j = i * i; j < limit; j += i) {
                isComposite[j] = true;
            }
        }
    }

    for (int i = sqrtLimit + 1; i < limit; i++) {
        if (!isComposite[i]) {
            primes[index++] = i;
        }
    }

    primes.resize(index);
    return primes;
}

int totient(int n, const vector<int>& primes) {
    const int MAX = primes.back();
    if (n >= MAX - 1) throw invalid_argument("MAX not large enough!");
    int ans = n;
```

```
    for (int i = 0, p = primes[0]; p * p <= n; i++) {
        if (n % p == 0) ans -= ans / p;
        while (n % p == 0) n /= p;
        p = primes[i];
    }

    if (n != 1) ans -= ans / n;
    return ans;
}

int main() {
    const int MAX = 1000000;
    vector<int> PRIMES = sieve(MAX);

    cout << "phi(15) = " << totient(15, PRIMES) << endl;
    cout << endl;

    for (int x = 1; x <= 11; x++) {
        cout << "phi(" << x << ") = " << totient(x, PRIMES) << endl;
    }

    return 0;
}
```

Maximo y Minimo Comun Divisor (GCD y LCD)

```
long gcd(long a, long b) {
    return b == 0 ? (a < 0 ? -a : a) : gcd(b, a % b);
}

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

Modulo

pow(x,n) MOD m

```
const long long MAX = static_cast<long long>(sqrt(numeric_limits<long
long>::max()));
const long long MIN = -MAX;

long long gcd(long long a, long long b) {
    return b == 0 ? (a < 0 ? -a : a) : gcd(b, a % b);
}

tuple<long long, long long, long long> egcd(long long a, long long b) {
    if (b == 0) return {a < 0 ? -a : a, 1LL, 0LL};
```

```
    auto v = egcd(b, a % b);
    long long tmp = get<1>(v) - get<2>(v) * (a / b);
    get<1>(v) = get<2>(v);
    get<2>(v) = tmp;
    return v;
}

long long modInv(long long a, long long m) {
    a = ((a % m) + m) % m;
    auto v = egcd(a, m);
    long long x = get<1>(v);
    return ((x % m) + m) % m;
}

long long modPow(long long a, long long n, long long mod) {
    if (n < 0) {
        return modPow(modInv(a, mod), -n, mod);
    }

    if (n == 0LL) return 1LL;
    long long p = a, r = 1LL;

    for (long long i = 0; n != 0; i++) {
        long long mask = 1LL << i;
        if ((n & mask) == mask) {
            r = (((r * p) % mod) + mod) % mod;
            n -= mask;
        }
        p = ((p * p) % mod + mod) % mod;
    }

    return ((r % mod) + mod) % mod;
}

int main() {
    long long r2 = modPow(3, 4, 1000000); // 81
    cout << r2 << endl;
    return 0;
}
```

Inversa Modular

```
vector<long> egcd(long a, long b) {
    if (b == 0) return {a, 1L, 0L};
    vector<long> v = egcd(b, a % b);
    long tmp = v[1] - v[2] * (a / b);
    v[1] = v[2];
    v[2] = tmp;
    return v;
}
```

```
long modInv(long a, long m) {

    a = ((a % m) + m) % m;

    vector<long> v = egcd(a, m);
    long gcd = v[0];
    long x = v[1];

    if (gcd != 1) return 0;
    return ((x + m) % m + m) % m;
}

int main() {
    // Prints 3 since 2*3 mod 5 = 1
    cout << modInv(2, 5) << endl;

    // Prints 0 because there is no
    // modular inverse such that 4*x mod 18 = 1
    cout << modInv(4, 18) << endl;

    return 0;
}
```

Factorizacion de Primos

```
vector<long long> primeFactorization(long long n);
long long pollardRho(long long n);
long long gcd(long long a, long long b);
bool isPrime(const long long n);

vector<long long> primeFactorization(long long n) {
    vector<long long> factors;
    if (n <= 0) throw invalid_argument("Invalid input");
    else if (n == 1) return factors;
    priority_queue<long long> divisorQueue;
    divisorQueue.push(n);
    while (!divisorQueue.empty()) {
        long long divisor = divisorQueue.top();
        divisorQueue.pop();
        if (isPrime(divisor)) {
            factors.push_back(divisor);
            continue;
        }
        long long next_divisor = pollardRho(divisor);
        if (next_divisor == divisor) {
            divisorQueue.push(divisor);
        } else {
            divisorQueue.push(next_divisor);
            divisorQueue.push(divisor / next_divisor);
        }
    }
```

```
    return factors;
}

long long pollardRho(long long n) {
    if (n % 2 == 0) return 2;
    long long x = 2 + (rand() % 999999);
    long long c = 2 + (rand() % 999999);
    long long y = x;
    long long d = 1;
    while (d == 1) {
        x = (x * x + c) % n;
        y = (y * y + c) % n;
        y = (y * y + c) % n;
        d = gcd(abs(x - y), n);
        if (d == n) break;
    }
    return d;
}

long long gcd(long long a, long long b) {
    return b == 0 ? a : gcd(b, a % b);
}

bool isPrime(const long long n) {
    if (n < 2) return false;
    if (n == 2 || n == 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;
    long long limit = static_cast<long long>(sqrt(n));
    for (long long i = 5; i <= limit; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return false;
    }
    return true;
}

int main() {
    vector<long long> result = primeFactorization(872342345);
    cout << "[";
    for (int i = 0; i < result.size(); i++) {
        cout << result[i];
        if (i < result.size() - 1) {
            cout << ",";
        }
    }
    cout << "]" << endl;

    return 0;
}
```

Saber si son Coprimos

```
long gcf(long a, long b) {
    return b == 0 ? a : gcf(b, a % b);
}

bool areCoprime(long a, long b) {
    return gcf(a, b) == 1;
}
```

Test Rabin Miller

```
import random
# Rabin_Miller primality check. Tests whether or not a number
# is prime with a failure rate of: (1/2)^certainty
def isPrime(n, certainty = 12 ):
    if(n < 2): return False
    if(n != 2 and (n & 1) == 0): return False
    s = n-1
    while((s & 1) == 0): s >>= 1
    for _ in range(certainty):
        r = random.randrange(n-1) + 1
        tmp = s
        mod = pow(r,tmp,n)
        while(tmp != n-1 and mod != 1 and mod != n-1):
            mod = (mod*mod) % n
            tmp <<= 1
        if (mod != n-1 and  (tmp & 1) == 0): return False
    return True

print(isPrime(5))
print(isPrime(1433))
print(isPrime(567887653))
print(isPrime(75611592179197710043))
print(isPrime(2055615302359620959301385122560474243849168107861717371811 63))
```

Geometria

Retornar el angulo mas pequeño entre dos vectores

```
const double PI = 3.14159265358979323846;

double radianesAGrados(double radianes) {
    return radianes * 180.0 / PI;
}

double gradosARadianes(double grados) {
    return grados * PI / 180.0;
```

```
}

//2D
double angleBetweenVectors(double v1x, double v1y, double v2x, double v2y) {
    // Para determinar el ángulo entre dos vectores v1 y v2, podemos usar la
siguiente fórmula:
    // dot(v1, v2) = len(v1) * len(v2) * cos(θ) y resolver para θ, donde dot(a, b)
es el producto escalar
    // y len(c) es la longitud de c.
    double dotproduct = (v1x * v2x) + (v1y * v2y);
    double v1Length = sqrt(v1x * v1x + v1y * v1y);
    double v2Length = sqrt(v2x * v2x + v2y * v2y);

    double value = dotproduct / (v1Length * v2Length);

    // La precisión de redondeo de un valor double puede hacer que el valor que
estamos a punto de pasar a la función acos esté ligeramente fuera de su dominio,
así que hacemos una comprobación de seguridad.
    if (value <= -1.0) return PI;
    if (value >= +1.0) return 0;
    return acos(value);
}
//3D
double angleBetweenVectors(
    double v1x, double v1y, double v1z,
    double v2x, double v2y, double v2z) {

    // Para determinar el ángulo entre dos vectores v1 y v2, podemos usar
    // la siguiente fórmula: dot(v1,v2) = len(v1)*len(v2)*cosθ y resolver
    // para θ donde dot(a,b) es el producto escalar y len(c) es la longitud de c.
    double dotproduct = (v1x * v2x) + (v1y * v2y) + (v1z * v2z);
    double v1Length = sqrt(v1x * v1x + v1y * v1y + v1z * v1z);
    double v2Length = sqrt(v2x * v2x + v2y * v2y + v2z * v2z);

    double value = dotproduct / (v1Length * v2Length);

    // La precisión de doble precisión puede llevar a un dominio ligeramente
inválido
    // para la función arccos, así que asegúrate de comprobar las condiciones de
límite.
    if (value <= -1.0) return PI;
    if (value >= +1.0) return 0;
    return acos(value);
}

int main() {

    double v1x = 1.0, v1y = 0.0, v1z = 0.0; // Ejemplo de vector 1
    double v2x = 0.0, v2y = 1.0, v2z = 0.0; // Ejemplo de vector 2
    double angle = angleBetweenVectors(1.0, 0.0, 0.0, 1.0); // Ejemplo de uso de
la función
    cout << "Ángulo entre los vectores: " << angle << " radianes" << endl;
    angle = angleBetweenVectors(v1x, v1y, v1z, v2x, v2y, v2z); // Calcular el
ángulo
```

```
    cout << "Ángulo entre los vectores: " << angle << " radianes" << endl;

    return 0;
}
```

Area de la interseccion de dos circulos

```
const double EPS = 1e-6;

struct Point {
    double x, y;
    Point() : x(0), y(0) {}
    Point(double x, double y) : x(x), y(y) {}
};

Point rotatePoint(Point fp, Point pt, double angle) {
    double fpx = fp.x;
    double fpy = fp.y;
    double ptx = pt.x;
    double pty = pt.y;
    double x = ptx - fpx;
    double y = pty - fpy;
    double xRotated = x * cos(angle) + y * sin(angle);
    double yRotated = y * cos(angle) - x * sin(angle);
    return Point(fpx + xRotated, fpy + yRotated);
}

double arccosSafe(double x) {
    if (x >= 1.0) return 0;
    if (x <= -1.0) return M_PI;
    return acos(x);
}

vector<Point> circleCircleIntersection(Point c1, double r1, Point c2, double r2) {
    double r, R;
    Point c, C;

    if (r1 < r2) r = r1,R = r2,c = c1,C = c2;
    else r = r2,R = r1,c = c2,C = c1;


    double dist = hypot(c1.x - c2.x, c1.y - c2.y);

    if (dist < EPS && abs(r - R) < EPS)
        return vector<Point>(); // Infinite solutions

    if (r + dist < R)
        return vector<Point>(); // No intersection (small circle contained within
big circle)

    if (r + R < dist)
```

```
        return vector<Point>(); // No intersection (circles are disjoint)

    double cx = c.x;
    double Cx = C.x;
    double cy = c.y;
    double Cy = C.y;

    double vx = cx - Cx;
    double vy = cy - Cy;
    double x = (vx / dist) * R + Cx;
    double y = (vy / dist) * R + Cy;
    Point point(x, y);

    if (abs(r + R - dist) < EPS || abs(R - (r + dist)) < EPS)
        return vector<Point>{point};

    double angle = arccosSafe((r * r - dist * dist - R * R) / (-2.0 * dist * R));
    Point pt1 = rotatePoint(C, point, angle);
    Point pt2 = rotatePoint(C, point, -angle);

    return vector<Point>{pt1, pt2};
}

double circleCircleIntersectionArea(Point c1, double r1, Point c2, double r2) {
    double r = r1, R = r2;
    Point c = c1, C = c2;

    if (r1 > r2) r = r2,R = r1,c = c2,C = c1;


    double dist = hypot(c1.x - c2.x, c1.y - c2.y);
    vector<Point> intersections = circleCircleIntersection(c1, r1, c2, r2);

    if (intersections.empty()) {
        return M_PI * r * r; // Smaller circle is contained within larger circle
    } else if (intersections.size() == 1) {
        if (dist < R) {
            return M_PI * r * r; // Check if the smaller circle is inside the
larger circle
        } else {
            return 0;
        }
    } else {
        double d = dist;
        double part1 = r * r * arccosSafe((d * d + r * r - R * R) / (2 * d * r));
        double part2 = R * R * arccosSafe((d * d + R * R - r * r) / (2 * d * R));
        double part3 = 0.5 * sqrt((-d + r + R) * (d + r - R) * (d - r + R) * (d +
r + R));
        return part1 + part2 - part3;
    }
}

int main() {
    Point c1(0, 0);
```

```
    double r1 = 3.0;
    Point c2(4, 0);
    double r2 = 2.0;

    double area = circleCircleIntersectionArea(c1, r1, c2, r2);

    cout << "Area of intersection: " << area << endl;

    return 0;
}
```

Punto de interesccion entre dos circulos

```
const double EPS = 0.0000001;

// Define a Point structure
struct Point {
    double x;
    double y;

    Point(double x, double y) : x(x), y(y) {}
};

// Define a Circle structure
struct Circle {
    double x;
    double y;
    double r;

    Circle(double x, double y, double r) : x(x), y(y), r(r) {}
};

// Due to double rounding precision, the value passed into the acos function
// may be outside its domain of [-1, +1], which would return NaN. We want to avoid
that.
double acossafe(double x) {
    if (x >= 1.0) return 0;
    if (x <= -1.0) return M_PI;
    return acos(x);
}

// Rotates a point about a fixed point at some angle 'a'
Point rotatePoint(Point fp, Point pt, double a) {
    double x = pt.x - fp.x;
    double y = pt.y - fp.y;
    double xRot = x * cos(a) + y * sin(a);
    double yRot = y * cos(a) - x * sin(a);
    return Point(fp.x + xRot, fp.y + yRot);
}

// Given two circles, this method finds the intersection point(s) of the two
```

```
circles (if any exists)
vector<Point> circleCircleIntersectionPoints(Circle c1, Circle c2) {
    double r, R, d, dx, dy, cx, cy, Cx, Cy;

    if (c1.r < c2.r) {
        r = c1.r,R = c2.r,cx = c1.x,cy = c1.y,Cx = c2.x,Cy = c2.y;
    } else {
        r = c2.r,R = c1.r,Cx = c1.x,Cy = c1.y,cx = c2.x,cy = c2.y;
    }

    // Compute the vector <dx, dy>
    dx = cx - Cx;
    dy = cy - Cy;

    // Find the distance between two points.
    d = sqrt(dx * dx + dy * dy);

    // There are an infinite number of solutions.
    // Seems appropriate to also return an empty vector.
    if (d < EPS && abs(R - r) < EPS) return vector<Point>();

    // No intersection (circles centered at the same place with different sizes)
    else if (d < EPS) return vector<Point>();

    double x = (dx / d) * R + Cx;
    double y = (dy / d) * R + Cy;
    Point P(x, y);

    // Single intersection (kissing circles)
    if (abs((R + r) - d) < EPS || abs(R - (r + d)) < EPS) return vector<Point>{P};

    // No intersection. Either the small circle is contained within the big circle
    // or the circles are simply disjoint.
    if ((d + r) < R || (R + r < d)) return vector<Point>();

    Point C(Cx, Cy);
    double angle = acossafe((r * r - d * d - R * R) / (-2.0 * d * R));
    Point pt1 = rotatePoint(C, P, +angle);
    Point pt2 = rotatePoint(C, P, -angle);
    return vector<Point>{pt1, pt2};
}

int main() {
    Circle c1(0, 0, 1);
    Circle c2(2, 0, 1);

    vector<Point> intersectionPoints = circleCircleIntersectionPoints(c1, c2);

    if (intersectionPoints.empty()) {
        cout << "No intersection." << endl;
    } else {
        cout << "Intersection Points:" << endl;
        for (const Point& p : intersectionPoints) {
            cout << "x: " << p.x << ", y: " << p.y << endl;
```

```
        }
    }

    return 0;
}
```

Puntos mas cercanos

```
using namespace std;

const double EPS = 1e-9;

struct Point {
    double x, y;

    Point(double xx, double yy) : x(xx), y(yy) {}

    double dist(const Point& pt) const {
        double dx = x - pt.x, dy = y - pt.y;
        return sqrt(dx * dx + dy * dy);
    }
};

struct XSort {
    bool operator()(const Point& pt1, const Point& pt2) const {
        if (abs(pt1.x - pt2.x) < EPS) return false;
        return (pt1.x < pt2.x);
    }
};

struct YXSort {
    bool operator()(const Point& pt1, const Point& pt2) const {
        if (abs(pt1.y - pt2.y) < EPS) {
            if (abs(pt1.x - pt2.x) < EPS) return false;
            return (pt1.x < pt2.x);
        }
        return (pt1.y < pt2.y);
    }
};

vector<Point> closestPair(vector<Point>& points) {
    if (points.empty() || points.size() < 2) return vector<Point>();

    int n = points.size();
    int xQueueFront = 0, xQueueBack = 0;

    sort(points.begin(), points.end(), XSort());
    set<Point, YXSort> yWorkingSet;

    Point pt1(0, 0), pt2(0, 0);
    double d = numeric_limits<double>::max();
```

```
    for (int i = 0; i < n; i++) {
        Point nextPoint = points[i];

        while (xQueueFront != xQueueBack && nextPoint.x - points[xQueueFront].x >
d) {
            Point pt = points[xQueueFront++];
            yWorkingSet.erase(pt);
        }

        double upperBound = nextPoint.y + d;
        auto next = yWorkingSet.upper_bound(nextPoint);
        while (next != yWorkingSet.end() && next->y <= upperBound) {
            double dist = nextPoint.dist(*next);
            if (dist < d) {
                pt1 = nextPoint;
                pt2 = *next;
                d = dist;
            }
            next++;
        }

        double lowerBound = nextPoint.y - d;
        next = yWorkingSet.lower_bound(Point(0, lowerBound));
        while (next != yWorkingSet.end() && next->y > lowerBound) {
            double dist = nextPoint.dist(*next);
            if (dist < d) {
                pt1 = nextPoint;
                pt2 = *next;
                d = dist;
            }
            next++;
        }

        if (yWorkingSet.count(nextPoint)) {
            pt1 = pt2 = nextPoint;
            d = 0;
            break;
        }

        yWorkingSet.insert(nextPoint);
        xQueueBack++;
    }

    return {pt1, pt2};
}

int main() {
    vector<Point> points = {Point(0, 0), Point(1, 1), Point(2, 2), Point(3, 3)};
    vector<Point> closest = closestPair(points);

    if (closest.size() == 2) {
        cout << closest[0].x << " " << closest[0].y << " " << closest[1].x << " "
<< closest[1].y << endl;
```

```
    } else {
        cout << "No closest pair found." << endl;
    }

    return 0;
}
```

Determinar si 3 puntos son coliniales

Estan en el misma linea

```
const double EPS = 1e-9;

int collinear(const pair<double, double>& a, const pair<double, double>& b, const
pair<double, double>& c) {
    if (hypot(b.first - a.first, b.second - a.second) < EPS) {
        throw invalid_argument("a cannot equal b");
    }

    double v1_x = b.first - a.first;
    double v1_y = b.second - a.second;
    double v2_x = c.first - a.first;
    double v2_y = c.second - a.second;

    double determinant = (v1_x * v2_y) - (v2_x * v1_y);

    if (abs(determinant) < EPS) {
        return 0;
    }

    return (determinant < 0 ? -1 : 1);
}

int main() {
    pair<double, double> a = make_pair(1, 1);
    pair<double, double> b = make_pair(3, 3);
    pair<double, double> c = make_pair(7, 7);

    // Returns 0 means that yes these points are collinear
    cout << collinear(a, b, c) << endl;

    // Returns +1 meaning that c is to the left of the line
    c = make_pair(2, 7);
    cout << collinear(a, b, c) << endl;

    // Returns -1 meaning that c is to the right of the line
    c = make_pair(2, -7);
    cout << collinear(a, b, c) << endl;

    return 0;
}
```

Determinar si puntos 3D son coplanos

En el mismo plano

```
const double EPS = 1e-7;

struct Vector {
    double x, y, z;
    Vector(double xx, double yy, double zz) : x(xx), y(yy), z(zz) {}
};

Vector cross(const Vector& v1, const Vector& v2) {
    double v3x = v1.y * v2.z - v1.z * v2.y;
    double v3y = v1.z * v2.x - v1.x * v2.z;
    double v3z = v1.x * v2.y - v1.y * v2.x;
    return Vector(v3x, v3y, v3z);
}

double dot(const Vector& v1, const Vector& v2) {
    return (v1.x * v2.x) + (v1.y * v2.y) + (v1.z * v2.z);
}

bool areCoplanar(
    double ax, double ay, double az,
    double bx, double by, double bz,
    double cx, double cy, double cz,
    double dx, double dy, double dz) {

    Vector v1(bx - ax, by - ay, bz - az);
    Vector v2(cx - ax, cy - ay, cz - az);
    Vector v3(dx - ax, dy - ay, dz - az);

    Vector v4 = cross(v1, v2);

    return abs(dot(v3, v4)) < EPS;
}

int main() {
    cout << "The points (0,0,0), (1,0,1), (0,1,1), (1,1,2) are coplanar: " <<
        areCoplanar(0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 2) << endl;

    cout << "The points (0,0,0), (3,3,3), (3,0,0), (0,4,0) are coplanar: " <<
        areCoplanar(0, 0, 0, 3, 3, 3, 3, 0, 0, 0, 4, 0) << endl;

    cout << "The points (0,0,0), (1,1,1), (2,2,2), (3,3,3) are coplanar: " <<
        areCoplanar(0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3) << endl;

    return 0;
}
```

Intereseccion de una linea con un circulo

```
const double EPS = 1e-9;

vector<pair<double, double>> lineCircleIntersection(
    double a, double b, double c, double x, double y, double r);

void display(const vector<pair<double, double>>& pts);

int main() {
    // Vertical Line passing through (1,0)
    auto result1 = lineCircleIntersection(1, 0, 1, 0, 0, 1);
    display(result1);

    // Horizontal line passing through (0,1)
    auto result2 = lineCircleIntersection(0, 1, 1, 0, 0, 1);
    display(result2);

    // Vertical line passing through (-1,0)
    auto result3 = lineCircleIntersection(1, 0, -1, 0, 0, 1);
    display(result3);

    return 0;
}

void display(const vector<pair<double, double>>& pts) {
    if (pts.empty()) {
        cout << "No intersection points" << endl;
    } else if (pts.size() == 1) {
        cout << "Tangent point: (" << pts[0].first << ", " << pts[0].second << ")"
<< endl;
    } else {
        cout << "Intersection points:" << endl;
        for (const auto& p : pts) {
            cout << "(" << p.first << ", " << p.second << ")" << endl;
        }
    }
}

vector<pair<double, double>> lineCircleIntersection(
    double a, double b, double c, double x, double y, double r) {
    double A = a * a + b * b;
    double B = 2 * a * b * y - 2 * a * c - 2 * b * b * x;
    double C = b * b * x * x + b * b * y * y - 2 * b * c * y + c * c - b * b * r *
r;
    double D = B * B - 4 * A * C;
    double x1, y1, x2, y2;
    // Vertical line case :0
    if (abs(b) < EPS) {
        // ax + by = c, but b = 0, so x = c/a
        double vx = c / a;
        // No intersection
```

```
        if (abs(x - vx) > r) return {};
        // Vertical line is tangent to circle
        if (abs((vx - r) - x) < EPS || abs((vx + r) - x) < EPS)
            return {{vx, y}};
        double dx = abs(vx - x);
        double dy = sqrt(r * r - dx * dx);
        return {{vx, y + dy}, {vx, y - dy}};
        // Line is tangent to circle
    }
    else if (abs(D) < EPS) {
        x1 = -B / (2 * A);
        y1 = (c - a * x1) / b;
        return {{x1, y1}};
        // No intersection point
    }
    else if (D < 0) {
        return {};
    }
    else {
        D = sqrt(D);
        x1 = (-B + D) / (2 * A);
        y1 = (c - a * x1) / b;
        x2 = (-B - D) / (2 * A);
        y2 = (c - a * x2) / b;
        return {{x1, y1}, {x2, y2}};
    }
}
```

Convertir puntos a forma general

ax + b + c = 0

```
void segmentToGeneralForm(double x1, double y1, double x2, double y2, double &a,
double &b, double &c) {
    a = y1 - y2;
    b = x2 - x1;
    c = x1 * y2 - y1 * x2;
}

int main() {
    double a, b, c;
    segmentToGeneralForm(1, 1, 3, -4, a, b, c);
    cout << fixed;
    cout.precision(2);
    cout << a << "x + " << b << "y + " << c << " = 0" << endl;

    return 0;
}
```

Distancia entre coordenadas geograficas

```
// Compute the distance between geographic coordinates in units
  // of its radii (multiply by 6371km for Earth)
double dist(double lat1, double lon1, double lat2, double lon2) {
    double dLat = radians(lat2 - lat1);
    double dLon = radians(lon2 - lon1);
    double a =
        sin(dLat / 2.0) * sin(dLat / 2.0)
        + cos(radians(lat1)) * cos(radians(lat2)) * sin(dLon / 2.0) * sin(dLon /
2.0);
    return 2.0 * atan2(sqrt(a), sqrt(1 - a));
}
```

Area de Triangulo

```
double triangleArea(double side1, double side2, double side3) {
    double s = (side1 + side2 + side3) / 2.0;
    return sqrt(s * (s - side1) * (s - side2) * (s - side3));
}

double triangleArea(double ax, double ay, double bx, double by, double cx, double
cy) {
    double v1_x = bx - ax;
    double v1_y = by - ay;
    double v2_x = cx - ax;
    double v2_y = cy - ay;

    double determinant = v1_x * v2_y - v2_x * v1_y;

    return abs(determinant) / 2.0;
}

int main() {
    double ax = 0, ay = 1;
    double bx = 2, by = 0;
    double cx = 3, cy = 4;

    cout << triangleArea(ax, ay, bx, by, cx, cy) << endl;

    double side1 = hypot(bx - ax, by - ay);
    double side2 = hypot(cx - ax, cy - ay);
    double side3 = hypot(bx - cx, by - cy);

    cout << triangleArea(side1, side2, side3) << endl;

    return 0;
}
```

Interseccion punto con Circulo

Given a circle and a point around or inside the circle we wish to find place(s) of intersection of the lines from the point which are tangent to the circle.

```
struct Point {
    double x, y;
    Point(double x = 0, double y = 0) : x(x), y(y) {}
};

const double EPS = 0.0000001;

double arcsinSafe(double x) {
    if (x <= -1.0) return -M_PI / 2.0;
    if (x >= +1.0) return +M_PI / 2.0;
    return asin(x);
}

vector<Point> pointCircleTangentPoints(Point center, double radius, Point pt) {
    double px = pt.x, py = pt.y;
    double cx = center.x, cy = center.y;

    double dx = cx - px;
    double dy = cy - py;
    double dist = sqrt(dx * dx + dy * dy);

    if (dist < radius) return vector<Point>();

    double angle1, angle2;

    angle1 = arcsinSafe(radius / dist);
    angle2 = atan2(dy, dx);

    double angle = angle2 - angle1;
    Point p1(cx + radius * sin(angle), cy + radius * -cos(angle));

    angle = angle1 + angle2;
    Point p2(cx + radius * -sin(angle), cy + radius * cos(angle));

    if (hypot(p1.x - p2.x, p1.y - p2.y) < EPS) return vector<Point>{pt};

    return vector<Point>{p1, p2};
}

int main() {
    double radius = 2.0;
    Point circleCenter(5, 0);
    Point origin(0, 0);

    vector<Point> points = pointCircleTangentPoints(circleCenter, radius, origin);

    if (points.size() == 2) {
        Point pt1 = points[0];
        Point pt2 = points[1];
```

```
        cout << "Points found: (" << pt1.x << ", " << pt1.y << ") (" << pt2.x <<
", " << pt2.y << ")\n";
    }

    return 0;
}
```

Rotar Puntos

```
#include <iostream>
#include <cmath>

struct Point2D {
  double x, y;

  Point2D(double x, double y) : x(x), y(y) {}
};

// Rotate point 'pt' a certain number of radians clockwise
// relative to some fixed point 'fp'. Note that the angle
// should be specified in radians, not degrees.
Point2D rotatePoint(Point2D fp, Point2D pt, double angle) {
  double fpx = fp.x;
  double fpy = fp.y;
  double ptx = pt.x;
  double pty = pt.y;
  double x = ptx - fpx;
  double y = pty - fpy;
  double xRotated = x * cos(angle) + y * sin(angle);
  double yRotated = y * cos(angle) - x * sin(angle);

  return Point2D(fpx + xRotated, fpy + yRotated);
}

int main() {

  // Suppose we want to rotate the point (4,5) about the point
  // (3, 5) 45 degrees (pi/4 radians) clockwise 8 times until
  // it cycles back to its original position:

  double angle = M_PI / 4.0;
  Point2D fixedPoint(3, 5);
  Point2D point(4, 5);

  // Prints all 8 rotations
  Point2D rotatedPoint = point;
  for (int i = 0; i < 8; i++) {
    cout << "Rotated point is now at: (" << rotatedPoint.x << ", " <<
rotatedPoint.y << ")\n";
    rotatedPoint = rotatePoint(fixedPoint, rotatedPoint, angle);
  }
```

```
  cout << "Rotated point is now at: (" << rotatedPoint.x << ", " << rotatedPoint.y
<< ")\n";

  return 0;
}
```

Convex Hull

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}

bool collinear(pt a, pt b, pt c) {
    return orientation(a, b, c) == 0;
}

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
```

```
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i],
include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    a = st;
}

int main() {
    vector<pt> points = {{0, 0}, {1, 1}, {2, 2}, {0, 2}, {2, 0}, {3, 3}, {4, 4}};


    convex_hull(points);

    cout << "Convex Hull" << endl;
    for (const auto& point : points) {
        cout << "(" << point.x << ", " << point.y << ")" << endl;
    }

    return 0;
}
```