Prime Logic: Math Algorithms in Computer Science

Number Theory Fundamentals and Algorithmic Applications

Mertkan Karaaslan (Zarabeth)

Abstract

This book serves as a comprehensive guide to number theory, explaining its fundamental principles and how they integrate with computer science. It bridges the gap between mathematical theory and its algorithmic applications in computing, focusing particularly on topics such as Number Systems, Prime Numbers, Modular Arithmetic, Diophantine equations, Euler's Totient Function, Discrete Logarithms, Chinese Remainder Theorem, and Continued Fractions. With each concept, the book provides detailed C++ code examples and problem-solving tasks to solidify understanding.

In addition to its focus on algorithmic applications, this book also serves as a comprehensive guide to the Olympiad mathematics world, offering insights and techniques that are invaluable for students participating in mathematical Olympiads. This resource is tailored for students, programmers, and anyone eager to learn how number theory underpins algorithms in modern computing and problem-solving competitions.

What Is Number Theory?

What are integers? Which integers are interesting? What can we do with them? How do we use them? How do they relate to mathematics? How do we use them to count more easily? What numbers can we make with them? How do we use them to write secret messages? How do we use them to read secret messages? How do we use them to run computers? How do we use them to run computers *more efficiently*? How do they help us with physics? What do they have to do with biology? How do we use them to talk with our spacecrafts?

These are questions about the subject of number theory. Number theory is the study of integers. As ancient as number theory is, humans are still uncovering mysteries behind integers and learning to use them in powerful new ways.

Integers:	$..., -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...$
Natural Numbers:	$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...$
Prime Numbers:	$2, 3, 5, 7, 11, 13, 17, 19, 23, ...$
Composite Numbers:	$4, 6, 8, 9, 10, 12, 14, 15, 16, ...$
Even Numbers:	$..., -6, -4, -2, 0, 2, 4, 6, 8, 10, 12, ...$
Odd Numbers:	$..., -7, -5, -3, -1, 1, 3, 5, 7, 9, 11, ...$
Perfect Squares:	$0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, ...$
Negative Cubes:	$-1, -8, -27, -64, -125, -216, -343, ...$
Powers of 2:	$1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...$
Abundant Numbers:	$12, 18, 20, 24, 30, 36, 40, 42, 48, 54, ...$
Palindromes:	$11, 313, 838, 3443, 7447, 57875, 10000001, ...$
Fibonacci Numbers:	$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$
Base Numbers:	$1_9, 44_5, 154_6, 10110_2, 2A41_{12}, ...$
Modulo 8 Residues:	$0, 1, 2, 3, 4, 5, 6, 7$

Each of these types of integers has stories—some that go back thousands of years. Since we use integers to describe so many things, knowing these stories helps us understand the world around us.

Divisibility/Primes/Co-prime/GCD/LCM

Divisibility

Divisibility is a fundamental concept in number theory, where one integer divides another without leaving a remainder. For example, if a divides b, it means there exists an integer k such that $b = a \times k$. In programming, we often check for divisibility using the modulus operator.

Example of Divisibility Code:

```cpp
#include <iostream>
using namespace std;

bool isDivisible(int a, int b) {
    return b % a == 0;
}

int main() {
    cout << isDivisible(3, 9) << endl;  // Output: 1 (true)
    cout << isDivisible(4, 9) << endl;  // Output: 0 (false)
    return 0;
}
```

Prime Numbers

Prime numbers are integers greater than 1 that are only divisible by 1 and themselves. To check whether a number is prime, we divide it by numbers up to its square root.

Example of Prime Check Code:

```cpp
#include <iostream>
#include <cmath>
using namespace std;

bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) return false;
    }
    return true;
}

int main() {
    cout << isPrime(7) << endl;  // Output: 1 (true)
    cout << isPrime(10) << endl; // Output: 0 (false)
    return 0;
}
```

Co-prime Numbers

Two numbers are co-prime if their greatest common divisor (GCD) is 1. This means they do not share any common factors other than 1.

Example of Co-prime Check Code:

```cpp
#include <iostream>
#include <algorithm> // for __gcd function
using namespace std;

bool areCoPrime(int a, int b) {
    return __gcd(a, b) == 1;
}

int main() {
    cout << areCoPrime(14, 15) << endl;  // Output: 1 (true)
    cout << areCoPrime(14, 28) << endl;  // Output: 0 (false)
    return 0;
}
```

Greatest Common Divisor (GCD)

The GCD of two numbers is the largest integer that divides both numbers without leaving a remainder. The Euclidean algorithm is an efficient way to compute the GCD.

Example of GCD Code:

```cpp
#include <iostream>
using namespace std;

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

int main() {
    cout << gcd(48, 18) << endl;  // Output: 6
    return 0;
}
```

Least Common Multiple (LCM)

The least common multiple of two integers is the smallest number that is divisible by both. It can be computed using the relationship between GCD and LCM:

$$\text{LCM}(a, b) = \frac{|a \times b|}{\text{GCD}(a, b)}$$

Example of LCM Code:

```cpp
#include <iostream>
using namespace std;

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

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

int main() {
    cout << lcm(12, 15) << endl;  // Output: 60
    return 0;
}
```

Continued Fractions

Continued fractions provide an alternative way to represent numbers as sums of integers and reciprocals. These are written as:

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \cdots + \frac{1}{a_n}}}$$

where $a_0, a_1, \ldots, a_n$ are integers. These representations can be finite or infinite, and continued fractions are especially useful for approximating irrational numbers.

Historically, continued fractions have been used to solve various mathematical problems. Christian Huygens, in 1687, applied continued fractions to calculate gear ratios for mechanical devices. Euler, Gauss, and other prominent mathematicians contributed to the development of this field.

One of the key properties of continued fractions is their ability to provide the best rational approximations to irrational numbers. For example, continued fractions are frequently used to approximate numbers such as π or $\sqrt{2}$.

Example Problem: Rational Approximation with Continued Fractions

Given a rational number, we can expand it into a continued fraction. For instance, the number 42/31 can be written as the continued fraction:

$$\frac{42}{31} = 1 + \frac{1}{2 + \frac{1}{1 + \frac{1}{4}}}$$

This is the finite continued fraction for 42/31.

Code Example

The following C++ code calculates the continued fraction representation of a given rational number:

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

using namespace std;

// Function to compute continued fraction for a rational number
vector<int> continued_fraction(int numerator, int denominator) {
    vector<int> result;

    while (denominator != 0) {
        result.push_back(numerator / denominator);
        int remainder = numerator % denominator;
        numerator = denominator;
        denominator = remainder;
    }

    return result;
}

int main() {
    int numerator = 42;
    int denominator = 31;

    vector<int> cf = continued_fraction(numerator, denominator);

    cout << "Continued fraction representation of " << numerator << "/" << denominator << " is
    : [";
    for (size_t i = 0; i < cf.size(); ++i) {
        cout << cf[i];
        if (i != cf.size() - 1) cout << ", ";
    }
    cout << "]" << endl;

    return 0;
}
```

This program computes the continued fraction of a rational number by repeatedly dividing the numerator by the denominator and capturing the integer part of the result. The remainder then becomes the new denominator, and the process repeats until the denominator becomes zero. This is a basic implementation of the Euclidean algorithm, which lies at the heart of continued fraction calculations.

Output Example: For the input 42/31, the program outputs:

$$[1, 2, 1, 4]$$

This corresponds to the continued fraction:

$$1 + \frac{1}{2 + \frac{1}{1 + \frac{1}{4}}}$$

Diophantine Equations

Diophantine equations are polynomial equations that require integer solutions. A basic form is the linear Diophantine equation:

$$ax + by = c$$

where a, b, and c are given integers, and x, y are unknown integers to be found. The equation has a solution if and only if the greatest common divisor (GCD) of a and b divides c.

The most common method to solve this is through the Extended Euclidean Algorithm. If the GCD of a and b is g, then the equation $ax + by = g$ has integer solutions, which can be scaled to find a solution for $ax + by = c$.

Example Explanation

Let's solve the equation $15x + 25y = 5$. The GCD of 15 and 25 is 5, which divides the constant 5 on the right-hand side, meaning a solution exists.

Using the Extended Euclidean Algorithm, we can find one particular solution and then express the general solution. The general solution will involve adding multiples of $\frac{b}{g}$ to x and subtracting multiples of $\frac{a}{g}$ from y.

Code Implementation

The following C++ code solves a linear Diophantine equation of the form $ax + by = c$.

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

// Function to implement the extended Euclidean Algorithm
int extended_gcd(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int gcd = extended_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return gcd;
}

// Function to solve the Diophantine equation
bool diophantine(int a, int b, int c, int &x, int &y, int &g) {
    g = extended_gcd(a, b, x, y);
    if (c % g != 0) {
        return false; // No solution exists
    }
    x *= c / g;
    y *= c / g;
    return true; // Solution exists
}

int main() {
    int a = 15, b = 25, c = 5;
    int x, y, g;

    if (diophantine(a, b, c, x, y, g)) {
        cout << "Solution exists: " << endl;
        cout << "x = " << x << ", y = " << y << endl;
    } else {
        cout << "No solution exists." << endl;
    }
    return 0;
}
```

- The ‘`extended_gcd`‘ function computes the GCD of a and b while also finding x and y such that $ax+by = \text{GCD}(a,b)$.

- The ‘`diophantine`‘ function checks if a solution exists by verifying if c is divisible by the GCD of a and b. If a solution exists, it scales the values of x and y accordingly.

- The output gives one solution for x and y. Further solutions can be generated by adding multiples of $\frac{b}{\text{GCD}}$ to x and subtracting multiples of $\frac{a}{\text{GCD}}$ from y.

Functions: Counting Divisors/Divisor Sum

Divisor Function

The divisor function $d(n)$, also known as the sigma function $\sigma(n)$, counts the number of divisors of an integer n. For a given integer n, the divisor function is defined as:

$$d(n) = \prod_{i=1}^{k}(e_i + 1)$$

where $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ is the prime factorization of n, and $p_1, p_2, \ldots, p_k$ are the distinct primes dividing n, and $e_1, e_2, \ldots, e_k$ are their corresponding exponents.

For example, consider $n = 12$, whose prime factorization is:

$$12 = 2^2 \times 3^1$$

The divisor function can be computed as:

$$d(12) = (2+1)(1+1) = 3 \times 2 = 6$$

This result indicates that 12 has six divisors: $1, 2, 3, 4, 6, 12$.

The divisor function is fundamental in solving problems that involve counting divisors, such as analyzing factorization properties or solving equations involving divisibility.

Divisor Sum Function

The divisor sum function $\sigma(n)$ computes the sum of all divisors of n. It is defined as:

$$\sigma(n) = \sum_{d|n} d$$

where the sum is taken over all divisors d of n.

For example, the divisors of $n = 6$ are $1, 2, 3, 6$, and the divisor sum function yields:

$$\sigma(6) = 1 + 2 + 3 + 6 = 12$$

The divisor sum function plays an important role in number theory. One notable application is in determining perfect numbers, which are numbers that satisfy:

$$\sigma(n) = 2n$$

An example of a perfect number is 6, since $\sigma(6) = 12 = 2 \times 6$.

Efficient Computation

Efficient computation of the divisor sum function is possible using prime factorization. If n has the prime factorization:

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$$

then the divisor sum function can be computed as:

$$\sigma(n) = \prod_{i=1}^{k} \frac{p_i^{e_i+1} - 1}{p_i - 1}$$

This formula allows the divisor sum function to be computed efficiently for large numbers by using the prime factors of n.

Euler's Totient Function

Euler's Totient function, often denoted as $\phi(n)$, is a fundamental concept in number theory. The function counts the number of integers less than n that are relatively prime to n, i.e., numbers whose greatest common divisor with n is 1.

Definition and Properties

The formal definition of Euler's Totient function is:

$$\phi(n) = |\{x \in \mathbb{Z}^+ : 1 \leq x \leq n \text{ and } \gcd(x, n) = 1\}|$$

Example: Let us evaluate $\phi(6)$: - The integers less than 6 are {1, 2, 3, 4, 5}. - Out of these, only 1 and 5 are relatively prime to 6, since $\gcd(1, 6) = 1$ and $\gcd(5, 6) = 1$. Thus, $\phi(6) = 2$.

Formulas

Euler's Totient function has some specific properties, especially when dealing with prime numbers and powers of primes.

- If p is prime, then:

  $$\phi(p) = p - 1$$

  Since all numbers less than p are relatively prime to p.

- For two distinct primes p and q, the Totient function is multiplicative:

  $$\phi(pq) = \phi(p) \times \phi(q) = (p - 1)(q - 1)$$

  This holds true for any two numbers m and n such that $\gcd(m, n) = 1$:

  $$\phi(mn) = \phi(m) \times \phi(n)$$

- For a prime power p^k, the function is:

  $$\phi(p^k) = p^k - p^{k-1}$$

  This is because every p-th number up to p^k is divisible by p, and there are p^{k-1} such numbers.

Example Code: Calculation of Euler's Totient Function in C++ Here is a C++ implementation to calculate Euler's Totient function for a number n:

```cpp
#include <iostream>
using namespace std;

int phi(int n) {
    int result = n;
    for (int p = 2; p * p <= n; ++p) {
        if (n % p == 0) {
            while (n % p == 0) {
                n /= p;
            }
            result -= result / p;
        }
    }
    if (n > 1) {
        result -= result / n;
    }
    return result;
}

int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;
    cout << "Euler's Totient function value for " << n << " is: " << phi(n) << endl;
    return 0;
}
```

This function works by iterating through potential prime factors of n. For each prime p, it reduces n by removing all factors of p and updates the result accordingly.

Modular Arithmetic/Chinese Remainder Theorem

Modular Arithmetic

Modular arithmetic is a system of arithmetic for integers, where numbers "wrap aroundüpon reaching a certain value, called the modulus. It is used extensively in number theory, cryptography, and computer science.

In modular arithmetic, we deal with the congruence relation. We say that two integers a and b are congruent modulo n (written $a \equiv b \pmod{n}$) if their difference is divisible by n:

$$a \equiv b \pmod{n} \iff n \mid (a - b)$$

Properties of Modular Arithmetic Some of the important properties of modular arithmetic include:

- Addition: $(a + b) \mod n = ((a \mod n) + (b \mod n)) \mod n$
- Subtraction: $(a - b) \mod n = ((a \mod n) - (b \mod n)) \mod n$
- Multiplication: $(a \times b) \mod n = ((a \mod n) \times (b \mod n)) \mod n$
- Exponentiation: For $a^b \mod n$, there are efficient algorithms like Modular Exponentiation to compute large powers under a modulus.

Example Code: Modular Exponentiation in C++

To handle large powers efficiently, we can use modular exponentiation with time complexity $O(\log b)$:

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

long long mod_exp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {  // If exp is odd, multiply base with result
            result = (result * base) % mod;
        }
        base = (base * base) % mod;  // Square the base
        exp = exp / 2;
    }
    return result;
}

int main() {
    long long base, exp, mod;
    cout << "Enter base, exponent, and modulus: ";
    cin >> base >> exp >> mod;
    cout << "Result: " << mod_exp(base, exp, mod) << endl;
    return 0;
}
```

This algorithm works by reducing the exponentiation problem into smaller parts using the properties of exponents.

Chinese Remainder Theorem (CRT)

The Chinese Remainder Theorem is a powerful tool in number theory that allows for the solution of simultaneous congruences. It states that if $n_1, n_2, \ldots, n_k$ are pairwise coprime, then the system of congruences:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

has a unique solution modulo $N = n_1 \times n_2 \times \cdots \times n_k$. That is, there exists a unique $x \pmod{N}$ that satisfies all the congruences.

The key idea behind the CRT is to break down a problem into smaller problems using moduli that are easier to solve. Since the moduli are pairwise coprime, we can treat each congruence separately, and then "piece"together the solutions.

Step-by-step: 1. Solve each congruence $x \equiv a_i \pmod{n_i}$ individually. 2. Combine the solutions using the formula:

$$x = \sum_{i=1}^{k} a_i \times N_i \times y_i$$

Where: - $N_i = \frac{N}{n_i}$ - y_i is the modular inverse of N_i modulo n_i, i.e., $N_i \times y_i \equiv 1 \pmod{n_i}$.

Example Code: Chinese Remainder Theorem in C++

Here is an implementation of the Chinese Remainder Theorem in C++ for two congruences.

```cpp
#include <iostream>
using namespace std;

// Function to find gcd of two numbers
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

// Function to find modular inverse of a with respect to m
int mod_inverse(int a, int m) {
    a = a % m;
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1)
            return x;
    }
    return 1; // If no inverse exists
}

// Function to solve Chinese Remainder Theorem for two equations
int crt(int a1, int n1, int a2, int n2) {
    int m1_inv = mod_inverse(n1, n2);  // Modular inverse of n1 mod n2
    int m2_inv = mod_inverse(n2, n1);  // Modular inverse of n2 mod n1

    int x = (a1 * n2 * m2_inv + a2 * n1 * m1_inv) % (n1 * n2);
    return x;
}

int main() {
    int a1, n1, a2, n2;
    cout << "Enter a1, n1, a2, and n2: ";
    cin >> a1 >> n1 >> a2 >> n2;

    if (gcd(n1, n2) != 1) {
        cout << "Moduli are not coprime. CRT can't be applied." << endl;
    } else {
        cout << "Solution x is: " << crt(a1, n1, a2, n2) << endl;
    }
    return 0;
}
```

This code finds the solution for two congruences using the Chinese Remainder Theorem. If the moduli are not coprime, it alerts that CRT cannot be applied.

Discrete Logarithms

The discrete logarithm problem is the problem of finding the integer x in the equation $g^x \equiv h \pmod{p}$, where g is a base, p is a prime modulus, and h is the result. This problem is considered computationally hard, especially when p is large. It plays a fundamental role in modern cryptographic protocols such as the Diffie-Hellman key exchange and ElGamal encryption.

Given a prime number p, a generator g of the multiplicative group $\mathbb{Z}_p^*$, and an element h, the discrete logarithm problem asks to find x such that:

$$g^x \equiv h \pmod{p}$$

The integer x is the discrete logarithm of h to the base g, denoted by:

$$x = \log_g(h) \pmod{p}$$

The difficulty of the discrete logarithm problem depends on the size of the modulus p. For large values of p, it is infeasible to compute the discrete logarithm using brute-force methods, making this problem a cornerstone of cryptographic security.

Baby-Step Giant-Step Algorithm

The baby-step giant-step algorithm is a well-known algorithm to solve the discrete logarithm problem in time $O(\sqrt{p})$, significantly faster than brute force methods. The algorithm splits the search for x into two parts: baby steps and giant steps, reducing the complexity by searching over smaller intervals.

Steps of the Algorithm: 1. Baby Steps: Compute and store the values $g^j \mod p$ for $j = 0, 1, \ldots, n-1$, where $n = \lceil \sqrt{p} \rceil$. 2. Giant Steps: Compute the values $h \cdot (g^{-n})^i \mod p$ for $i = 0, 1, \ldots, n-1$ and check for a match with the precomputed baby steps. 3. Meet in the Middle: The solution x is found when a match is detected, combining the baby step and giant step indices.

Algorithm:

```cpp
#include <iostream>
#include <cmath>
#include <unordered_map>
using namespace std;

// Function to compute (base^exp) % mod using modular exponentiation
long long mod_exp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp = exp / 2;
    }
    return result;
}

// Baby-Step Giant-Step algorithm to solve g^x = h (mod p)
long long baby_step_giant_step(long long g, long long h, long long p) {
    long long n = (long long) sqrt(p) + 1;

    // Baby steps: compute and store g^j % p
    unordered_map<long long, long long> value;
    for (long long j = 0, cur = h; j < n; ++j) {
        value[cur] = j;
        cur = (cur * g) % p;
    }

    // Precompute g^n % p
    long long gn = mod_exp(g, n, p);

    // Giant steps: search for match
    for (long long i = 1, cur = gn; i <= n; ++i) {
        cur = (cur * gn) % p;
        if (value.count(cur)) {
            return i * n - value[cur];
        }
    }
    return -1; // No solution found
}

int main() {
    long long g, h, p;
    cout << "Enter base (g), result (h), and modulus (p): ";
    cin >> g >> h >> p;

    long long x = baby_step_giant_step(g, h, p);
    if (x == -1) {
        cout << "No solution found." << endl;
    } else {
        cout << "Discrete logarithm x is: " << x << endl;
    }
    return 0;
}
```

Let's consider an example where we need to solve $2^x \equiv 15 \pmod{29}$ using the Baby-Step Giant-Step algorithm.

1. Baby Steps: We compute $2^j \mod 29$ for $j = 0, 1, \ldots, 4$:

$$2^0 \equiv 1, \quad 2^1 \equiv 2, \quad 2^2 \equiv 4, \quad 2^3 \equiv 8, \quad 2^4 \equiv 16 \pmod{29}$$

We store these values in a table.

2. Giant Steps: We compute $15 \cdot (2^{-5})^i \mod 29$ for $i = 0, 1$, and find a match. We know that $x = 13$. Thus, $2^{13} \equiv 15 \pmod{29}$, and the discrete logarithm $x = 13$.

Properties of the Discrete Logarithm

- The existence of the discrete logarithm depends on whether the number h belongs to the cyclic group generated by g.
- The difficulty of computing discrete logarithms increases significantly as the size of the modulus p grows, which is why it forms the basis for secure cryptographic systems.

Cryptographic Applications
The discrete logarithm problem is at the heart of various cryptographic algorithms:

- Diffie-Hellman Key Exchange: Allows two parties to securely exchange cryptographic keys over an insecure channel.
- ElGamal Encryption: A public-key encryption scheme based on the difficulty of the discrete logarithm problem.
- Digital Signature Algorithm (DSA): A widely used standard for digital signatures that relies on the discrete logarithm problem.

The security of these systems relies on the assumption that computing the discrete logarithm is computationally infeasible for large values of p, ensuring that adversaries cannot efficiently solve for x.

Number Systems

A number system is a mathematical way of representing numbers using a set of digits or symbols in a consistent manner. Number systems play a crucial role in both mathematics and computer science, particularly when dealing with different bases and their respective arithmetic operations.

There are four commonly used number systems in computer science:

1. Binary Number System (Base 2): The binary system uses only two digits, 0 and 1, to represent numbers. Each binary digit is known as a bit. Binary is the fundamental system for computers because it represents the off (0) and on (1) states of digital circuits.

For example, the binary number 1101_2 is equivalent to $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$ in decimal.

```
// Function to convert decimal to binary
string decimalToBinary(int n) {
    string binary = "";
    while (n > 0) {
        binary = to_string(n % 2) + binary;
        n = n / 2;
    }
    return binary;
}

int main() {
    int decimal = 13;
    cout << "Binary of 13: " << decimalToBinary(decimal) << endl;
    return 0;
}
```

2. Octal Number System (Base 8): The octal system uses digits from 0 to 7. Each digit in this system represents a power of 8. The octal system is often used in computing as a shorthand for representing binary numbers, where each octal digit corresponds to three binary digits.

For example, the octal number 157_8 is equivalent to $1 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 = 111$ in decimal.

```
// Function to convert decimal to octal
string decimalToOctal(int n) {
    string octal = "";
    while (n > 0) {
        octal = to_string(n % 8) + octal;
        n = n / 8;
    }
    return octal;
}

int main() {
    int decimal = 111;
    cout << "Octal of 111: " << decimalToOctal(decimal) << endl;
    return 0;
}
```

3. Decimal Number System (Base 10): The decimal system is the most familiar number system and uses digits from 0 to 9. It is the base-10 system that humans typically use for everyday arithmetic. Each digit's position represents a power of 10.

For example, the decimal number 375_{10} can be expanded as $3 \times 10^2 + 7 \times 10^1 + 5 \times 10^0 = 375$.

4. Hexadecimal Number System (Base 16): The hexadecimal system uses 16 symbols to represent numbers: digits 0-9 and letters A-F, where A represents 10, B represents 11, and so on. This system is particularly useful in computer science because one hexadecimal digit corresponds to four binary digits, making it easier to represent large binary numbers.

For example, the hexadecimal number $2A3_{16}$ is equivalent to $2 \times 16^2 + A \times 16^1 + 3 \times 16^0 = 675$ in decimal.

```
// Function to convert decimal to hexadecimal
string decimalToHex(int n) {
    string hex = "";
    char hexDigits[] = "0123456789ABCDEF";
    while (n > 0) {
        hex = hexDigits[n % 16] + hex;
        n = n / 16;
    }
    return hex;
}

int main() {
    int decimal = 675;
    cout << "Hexadecimal of 675: " << decimalToHex(decimal) << endl;
    return 0;
}
```

Primitive Root

A primitive root of a prime p is an integer g such that the powers of g modulo p generate all integers from 1 to $p-1$. In other words, for every integer a that is co-prime with p, there exists an integer k such that:

$$g^k \equiv a \pmod{p}$$

The integer g is called a primitive root modulo p, and g is said to have an order of $p-1$. This means that g is a generator of the multiplicative group of integers modulo p. The concept of primitive roots is central in number theory and plays a crucial role in cryptography.

Example: Primitive Roots for $p = 7$

For $p = 7$, the elements of $\mathbb{Z}_7^*$ are $1, 2, 3, 4, 5, 6$. The primitive roots modulo 7 are 3 and 5 because their powers generate all elements of $\mathbb{Z}_7^*$.

For example, consider the powers of 3 modulo 7:

$$3^1 \equiv 3, \quad 3^2 \equiv 2, \quad 3^3 \equiv 6, \quad 3^4 \equiv 4, \quad 3^5 \equiv 5, \quad 3^6 \equiv 1 \pmod{7}$$

Since all elements of $\mathbb{Z}_7^*$ appear in the list of powers, 3 is a primitive root modulo 7.

Existence of Primitive Roots

It is a well-established result that a prime number p has at least one primitive root. In fact, the number of primitive roots modulo a prime p is given by Euler's totient function $\phi(p-1)$, which counts how many integers less than $p-1$ are co-prime with $p-1$. For non-prime numbers, not every number has a primitive root.

Algorithm for Finding Primitive Roots

To find a primitive root of a prime number p, we can use the following method: 1. Check whether a candidate g generates all numbers in $\mathbb{Z}_p^*$ by computing $g^k \mod p$ for all k. 2. If all distinct numbers from 1 to $p-1$ are generated, g is a primitive root.

C++ Code for Finding Primitive Roots

The following C++ code demonstrates how to find a primitive root modulo p.

```cpp
// Function to compute (base^exp) % mod
long long mod_exp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

// Function to check if ’g’ is a primitive root modulo ’p’
bool is_primitive_root(long long g, long long p) {
    long long phi = p - 1;
    vector<long long> factors;

    // Find all prime factors of phi (p-1)
    for (long long i = 2; i * i <= phi; i++) {
        if (phi % i == 0) {
            factors.push_back(i);
            while (phi % i == 0)
                phi /= i;
        }
    }
    if (phi > 1) {
        factors.push_back(phi);
    }

    // Check if g^((p-1)/factor) mod p != 1 for all factors
    for (auto factor : factors) {
        if (mod_exp(g, (p - 1) / factor, p) == 1) {
            return false;
        }
    }
    return true;
}

int main() {
    long long p;
    cout << "Enter prime number p: ";
    cin >> p;

    for (long long g = 2; g < p; ++g) {
        if (is_primitive_root(g, p)) {
            cout << g << " is a primitive root modulo " << p << endl;
            break;
        }
    }
    return 0;
}
```

Properties of Primitive Roots

- If g is a primitive root modulo p, then the powers $g^1, g^2, \ldots, g^{p-1}$ are distinct and form a complete set of residues modulo p.
- If g is a primitive root modulo p, then $g^k \equiv 1 \pmod{p}$ only when $k = p$

Tasks For Practice

1 Divisibility/Primes/Co-prime/GCD/LCM

Zeus and The Birthday Paradox [6]

Zeus has recently found an interesting concept called the Birthday Paradox. It states that given a random set of 23 people, there is around 50% chance that some two of them share the same birthday. Zeus finds this very interesting, and decides to test this with the inhabitants of Tedland.

In Tedland, there are 2^n days in a year. Zeus wants to interview k people from Tedland, each of them has birthday in one of 2^n days (each day with equal probability). He is interested in the probability of at least two of them have the birthday at the same day.

Zeus knows that the answer can be written as an irreducible fraction $\frac{A}{B}$. He wants to find the values of A and B (he does not like to deal with floating point numbers). Can you help him?

Input

The first and only line of the input contains two integers n and k ($1 \leq n \leq 10^{18}, 2 \leq k \leq 10^{18}$), meaning that there are 2^n days in a year and that Zeus wants to interview exactly k people.

Output

If the probability of at least two k people having the same birthday in $2n$ days long year equals $\frac{A}{B}$($A \geq 0, B \geq 1$, $\gcd(A, B) = 1$), print the A and B in a single line.

Since these numbers may be too large, print them modulo $10^6 + 3$. Note that A and B must be coprime **before** their remainders modulo $10^6 + 3$ are taken.

Examples

input:
3 2
output:
1 8

input:
1 3
output:
1 1

input:
4 3
output:
23 128

Note

In the first sample case, there are $2^3 = 8$ days in Tedland. The probability that 2 people have the same birthday among 2 people is clearly $\frac{1}{8}$, so $A = 1, B = 8$.

In the second sample case, there are only $2^1 = 2$ days in Tedland, but there are 3 people, so it is guaranteed that two of them have the same birthday. Thus, the probability is 1 and $A = B = 1$.

Solution

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

typedef long long ll;
typedef vector<int> vi;

const int MOD = 1e6 + 3;

ll power(ll base, ll exp)
{
  ll ans = 1;
    while(exp)
    {
    if(exp&1) ans = (ans*base)%MOD;
    base = (base*base)%MOD;
    exp>>=1;
  }
    return ans;
}

int main()
{
  ios_base::sync_with_stdio(false); cin.tie(0);
  ll n, k;
  cin >> n >> k;
  if(n <= 63 && k > (1LL<<n))
  {
    cout << 1 << " " << 1;
    return 0;
  }
  ll v2 = 0;
  int digits = __builtin_popcountll(k - 1);
  v2 = k - 1 - digits;
  ll ntmp = n % (MOD - 1);
  if(ntmp < 0) ntmp += (MOD - 1);
  ll ktmp = k % (MOD - 1);
  if(ktmp < 0) ktmp += (MOD - 1);
  ll v2tmp = v2 % (MOD - 1);
  if(v2tmp < 0) v2tmp += (MOD - 1);
  ll exponent = ntmp*(ktmp - 1) - v2tmp;
  exponent %= (MOD - 1);
  if(exponent < 0) exponent += MOD - 1;
  ll denom = power(2, exponent);
  ll numpart = 0;
  if(k - 1 >= MOD)
  {
    numpart = 0;
  }
  else
  {
    ll prod = 1;
    ll ntmp2 = power(2, ntmp);
    prod = power(2, v2tmp);
    prod = power(prod, MOD - 2);
    if(prod < 0) prod += MOD;
    for(ll y = 1; y <= k - 1; y++)
    {
      prod = (prod * (ntmp2 - y))%MOD;
    }
    numpart = prod;
  }
  ll num = (denom - numpart)%MOD;
  num %= MOD; denom %= MOD;
  if(num < 0) num += MOD;
  if(denom < 0) denom += MOD;
  cout << num << " " << denom;
  return 0;
}
```

Source Code 1: Zeus and The Birthday Paradox

2 Continued Fraction

Continued Fractions [7]

A continued fraction of height n is a fraction of form $a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \cdots + \frac{1}{a_n}}}$. You are given two rational numbers, one is represented as $\frac{p}{q}$ and the other one is represented as a finite fraction of height n. Check if they are equal.

Input

The first line contains two space-separated integers $p, q (1 \leq q \leq p \leq 10^{18})$ — the numerator and the denominator of the first fraction.

The second line contains integer $n (1 \leq n \leq 90)$ — the height of the second fraction. The third line contains n space-separated integers $a_1, a_2, ..., a_n (1 \leq a_i \leq 10^{18)}$ — the continued fraction.

Please, do not use the %lld specifier to read or write 64-bit integers in C++. It is preferred to use the cin, cout streams or the % I64d specifier.

Output

Print *"YES"* if these fractions are equal and *"NO"* otherwise.

Examples

input:
9 4
2
2 4

output:
YES

input:
9 4
3
2 3 1

output:
YES

input:
9 4
3
1 2 4

output:
NO

Note

In the first sample $2 + \frac{1}{4} = \frac{9}{4}$.

In the second sample $2 + \frac{1}{3+\frac{1}{1}} = 2 + \frac{1}{4} = \frac{9}{4}$.

In the third sample $1 + \frac{1}{2+\frac{1}{4}} = \frac{13}{9}$.

Solution

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

#define pb push_back
#define mp make_pair
#define sz(a) (int)(a).size()
#define all(a) (a).begin(), (a).end()
#define rall(a) (a).rbegin(), (a).rend()

#define forn(i,n) for (int i=0; i<int(n); ++i)
#define fornd(i,n) for (int i=int(n)-1; i>=0; --i)
#define forab(i,a,b) for (int i=int(a); i<int(b); ++i)

typedef long long ll;
typedef long double ld;
typedef unsigned long long ull;

const int INF = (int) 1e9;
const long long INF64 = (long long) 1e18;
const long double eps = 1e-9;
const long double pi = 3.14159265358979323846;

ll p,q,n;
vector <ll> a;
vector <ll> m;

bool read(){
    if (!(cin >> p >> q >> n)) return false;
    a.assign(n,0);
    forn(i,n)
        cin >> a[i];
    return true;
}

void calc(ll a, ll b){
    if (a == 0 || b == 0) return;
    ll q = a/b;
    m.pb(q);
    a -= b*q;
    calc(b,a);
}

void solve(){
    if (n > 1 && a[n-1] == 1){
        a[n-2]++;
        a.pop_back();
        n--;
    }
    m.clear();
    calc(p,q);
    bool ans = true;
    if (sz(a)!=sz(m)) ans = false;
    forn(i,min(sz(a),sz(m)))
        ans = ans && (a[i]==m[i]);
    if (ans)
        puts("YES");
    else
        puts("NO");
}

int main(){
#ifdef dudkamaster
    freopen("input.txt","rt",stdin);
    freopen("output.txt","wt",stdout);
#endif
    while (read())
        solve();
    return 0;
}
```

Source Code 2: Continued Fractions

3 Diophantine Equations

How Many Points? [8]

Given two points A and B on the $X - Y$ plane, output the number of the lattice points on the segment AB.

Note that A and B are also lattice points. Those who are confused with the definition of lattice points, lattice points are those points which have both x and y integer co-ordinates.

For example, for $A(3,3)$ and $B(-1,-1)$ the output is 5. The points are: $(-1,-1),(0,0),(1,1),(2,2)$ **and** $(3,3)$.

Input

Input starts with an integer $T(\leq 125)$, denoting the number of test cases.

Each case contains four integers, A_x, A_y, B_x and B_y. Each of them will fit into a 32 bit signed integers.

Output

For each test case, print the case number and the number of lattice points between AB.

Examples

input:
2
3 3 -1 -1
0 0 5 2

output:
Case 1: 5
Case 2: 2

Solution

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

int main() {

    // For fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int t;
    cin >> t;

    for(int ts = 1; ts <= t; ++ts) {
        pair <long long, long long> A, B;
        cin >> A.first >> A.second >> B.first >> B.second;

        cout << "Case " << ts << ": " << __gcd(abs(A.first - B.first), abs(A.second - B.second)) + 1 << '\n';
    }

    return 0;
}
```

Source Code 3: How Many Points?

4 Functions: Counting Divisors/Divisor Sum

Efficient Pseudo Code [9]

Sometimes it's quite useful to write pseudo codes for problems. Actually you can write the necessary steps to solve a particular problem. In this problem you are given a pseudo code to solve a problem and you have to implement the pseudo code efficiently. Simple! Isn't it? :)

```
pseudo code
{

    take two integers n and m

    let p = n ^ m (n to the power m)

    let sum = summation of all the divisors of p

    let result = sum MODULO 1000,000,007
}
```

Now, given n and m you have to find the desired result from the pseudo code. For example if $n = 12$ and $m = 2$. Then if we follow the pseudo code, we get:

```
pseudo code

{

    take two integers n and m

    so, n = 12 and m = 2

    let p = n ^ m (n to the power m)

    so, p = 144

    let sum = summation of all the divisors of p

    so, sum = 403, since the divisors of p are:
        {1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144}

    let result = sum MODULO 1000,000,007

    so, result = 403
```

Input

Input starts with an integer $T (\leq 5000)$, denoting the number of test cases.

Each test case will contain two integers, $n (1 \leq n)$ and $m (0 \leq m)$. Each of n and m will be fit into a 32 bit signed integer.

Output

For each case of input you have to print the case number and the result according to the pseudo code.

Examples

input:
3
12 2
12 1
36 2

output:
Case 1: 403

Case 2: 28
Case 3: 3751

Solution

```
#include<bits/stdc++.h>
using namespace std;
typedef long long LL;

LL mod = 1000000007;

vector<LL> primes;
map<LL, LL> times;

LL bigmod (LL b, LL p, LL m)
{
    if (p==0) return 1;
    if (p%2==0) {
        LL x=bigmod(b, p/2, m)%m;
        return (x*x)%m;
    }
    else return (b%m * bigmod(b, p-1, m))%m;
}

LL inv_mod(LL a, LL m) {
    return bigmod(a, m-2, m);
}

void factorization(long long n) {
    if (n % 2 == 0) {
        primes.push_back(2);
        while (n % 2 == 0) {
            times[2]++;
            n /= 2;
        }
    }
    for (long long d = 3; d * d <= n; d += 2) {
        if (n % d == 0) {
        primes.push_back(d);
            while (n % d == 0) {
                times[d]++;
                n /= d;
            }
        }
    }
    if (n > 1) {
        if (!times[n])
            primes.push_back(n);
        times[n]++;
    }
}

int main()
{
    int t, ca=1;
    cin>>t;
    while (t--) {
        primes.clear();
        times.clear();
        LL n, m;
        cin>>n>>m;

        factorization(n);
        for (auto i: primes) {
            times[i] *= m;
        }

        LL ans = 1;

        for (auto i: primes) {
            LL now = bigmod(i, times[i] + 1, mod);
            now -= 1;
            if (now < 0)  // The program may give a negative result for modulo of negative numbers
                now += mod;  // So we convert it to a positive value by adding mod
            LL low = inv_mod(i - 1, mod);
            now *= low;
            ans = ((ans % mod) * (now % mod)) % mod;
        }
        cout<<"Case "<<ca++<<": "<<ans<<endl;
    }

    return 0;
}
```

Source Code 4: Efficient Pseudo Code

5 Euler's Totient Function

Mathematically Hard [10]

Mathematically some problems look hard. But with the help of the computer, some problems can be easily solvable.

In this problem, you will be given two integers a and b. You have to find the summation of the scores of the numbers from a to b (inclusive). The score of a number is defined as the following function:

$$\text{score}(x) = n^2, \quad n < x \quad \text{and} \quad \gcd(n, x) = 1$$

To illustrate, n is the number of relatively prime numbers with x, which are smaller than x.

For example, For 6, the relatively prime numbers with 6 are $\{1, 5\}$. So, score $(6) = 2^2 = 4$. For 16, the relatively prime numbers with 16 are $\{1, 3, 5, 7, 9, 11, 13, 15\}$. So, score $(16) = 8^2 = 64$.

Now, you have to solve this task.

Input

Input starts with an integer $T(\leq 10^5)$, denoting the number of test cases.

Each case will contain two integers a and b $(2 \leq a \leq b \leq 5 \cdot 10^6)$.

Output

For each case, print the case number and the summation of all the scores from a to b.

Examples

input:
3
6 6
8 8
2 20

output:
Case 1: 4
Case 2: 16
Case 3: 1237

Note

Two integers are said to be relatively prime, if the greatest common divisor for them is 1.

Euler's totient function $\phi(n)$ applied to a positive integer n is defined to be the number of positive integers less than or equal to n that are relatively prime to n. $\phi(n)$ is read **"phi of n"**.

Given the general prime factorization of $n = p_1^{e1} p_2^{e2} \dots p_m^{em}$, one can compute $\phi(n)$ using the formula:

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\dots\left(1 - \frac{1}{p_m}\right)$$

Solution

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

#define M 5000000

int phi[M+2];
unsigned long long phiSum[M+2];

void calculatePhi(){
    for(int i=2; i<=M; i++)
        phi[i] = i;
    for(int i =2; i<=M; i++)
        if(phi[i]==i)
            for(int j=i; j<=M; j+=i)
                phi[j]-=phi[j]/i;
}

int main(){

    calculatePhi();
    phiSum[1] = 0;

    for(int i=2; i<=M; i++)
        phiSum[i]= ((unsigned long long)phi[i]* (unsigned long long)phi[i])+phiSum[i-1];

    // for(int i=1; i<10; ++i)
    //  cout<<phi[i]<<" "<<phiSum[i]<<endl;
    // deb(phi[2]);
    // deb(phiSum[20]);

    int tc,t(1);
    for(scanf("%d",&tc); tc; ++t,--tc){
        int a,b;
        scanf("%d%d",&a,&b);
        unsigned long long x = phiSum[b]-phiSum[a-1];
        printf("Case %d: %llu\n",t,x);
    }
    return 0;
}
```

Source Code 5: Mathematically Hard

6 Modular Arithmetic/CRT

Remainders Game [11]

Today Pari and Arya are playing a game called Remainders.

Pari chooses two positive integer x and k, and tells Arya k but not x. Arya have to find the value x **mod** k. There are n ancient numbers $c_1, c_2, ..., c_n$ and Pari has to tell Arya x **mod** c_i if Arya wants. Given k and the ancient values, tell us if Arya has a winning strategy independent of value of x or not. Formally, is it true that Arya can understand the value x **mod** k for any positive integer x?

Note, that x **mod** y means the remainder of x after dividing it by y.

Input

The first line of the input contains two integers n and k ($1 \leq n, k \leq 1000000$) - the number of ancient integers and value k that is chosen by Pari.

The second line contains n integers $c_1, c_2, ..., c_n$ ($1 \leq c_i \leq 1000000$).

Output

Print "Yes"(without quotes) if Arya has a winning strategy independent of value of x, or "No"(without quotes) otherwise.

Examples

input:
4 5
2 3 5 12
output:
Yes

input:
2 7
2 3
output:
No

Note

In the first sample, Arya can understand x **mod 5** because 5 is one of the ancient numbers.

In the second sample, Arya can't be sure what x **mod 7** is. For example 1 and 7 have the same remainders after dividing by 2 and 3, but they differ in remainders after dividing by 7.

Solution

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

inline int in() { int x; scanf("%d", &x); return x; }
const long long N = 1200021;

int cntP[N], isP[N];

int main()
{
  for(int i = 2; i < N; i++)
    if(!isP[i])
      for(int j = i; j < N; j += i)
        isP[j] = i;
  int n = in(), k = in();
  for(int i = 0; i < n; i++)
  {
    int x = in();
    while(x > 1)
    {
      int p = isP[x];
      int cnt = 0;
      while(x % p == 0)
      {
        cnt++;
        x /= p;
      }
      cntP[p] = max(cntP[p], cnt);
    }
  }
  bool ok = 1;
  while(k > 1)
  {
    ok &= (cntP[isP[k]] > 0);
    cntP[isP[k]]--;
    k /= isP[k];
  }
  cout << (ok ? "Yes\n" : "No\n");
}
```

Source Code 6: Remainders Game

7 Discrete Logarithms

Distributing Chocolates [12]

I have bought n chocolates for my young cousins. Every chocolate is different. So, in the contest I added the problem that how many ways I can distribute the chocolates to my K cousins. I can give more chocolates to some cousins, and may give no chocolate to some. For example, I have three cousins and I bought 2 chocolates a and b. Then I can distribute them in the following 9 ways:

#	Cousin 1	Cousin 2	Cousin 3
1	a, b		
2		a, b	
3			a, b
4	a	b	
5	a		b
6		a	b
7	b	a	
8	b		a
9		b	a

Tabelle 1: Possible ways to distribute chocolates to 3 cousins

As the result can be large, I asked for the result modulo **100 000 007** (a prime). But after that I found that this problem is easier than I thought. So, I changed the problem a little bit. I will give you the number of my cousins K and the result modulo 100000007($10^8 + 7$). Your task is to find the number of chocolates I have bought. If there are several solutions, you have to find the minimum one.

Input

Input starts with an integer $T(\leq 1000)$, denoting the number of test cases.

Each case starts with a line containing two integers $K(2 \leq K \leq 10^7)$ and result $(0 \leq result < 100000007)$. You can assume that the input data is valid, that means a solution exists.

Output

For each case, print the case number and the minimum possible number of chocolates I have bought.

Examples

input:
2
3 9
2 100

output:
Case 1: 2
Case 2: 23502611

Solution

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef int64_t ll;
typedef uint64_t ull;

// Modulo value (given as 10^8 + 7)
const ll M = 100000007;

// Custom hash function to prevent hash collisions in unordered_map
struct custom_hash {
    static ull splitmix64(ull x) {
        // SplitMix64 hash function
        x += 0x9e3779b97f4a7c15ull;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ull;
        x = (x ^ (x >> 27)) * 0x94d049bb133111ebull;
        return x ^ (x >> 31);
    }
    size_t operator()(ull x) const {
        static const ull FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};

// Modular multiplication to handle large numbers without overflow
ll mod_mul(ll a, ll b, ll mod) {
    return (__int128)a * b % mod;
}

// Modular exponentiation (fast power)
ll mod_pow(ll a, ll b, ll mod) {
    ll res = 1;
    a %= mod;
    while (b > 0) {
        if (b & 1)
            res = mod_mul(res, a, mod);
        a = mod_mul(a, a, mod);
        b >>= 1;
    }
    return res;
}

// Modular inverse using Fermat's Little Theorem (since mod is prime)
ll mod_inv(ll a, ll mod) {
    return mod_pow(a, mod - 2, mod);
}

// Baby-Step Giant-Step algorithm to solve for minimal n in K^n     result (mod M)
ll bsgs(ll K, ll result) {
    if (result == 1)
        return 0; // The minimal n is 0 if result is 1
    ll m = sqrt(M) + 1;
    unordered_map<ll, ll, custom_hash> table; // Hash table for baby steps
    ll e = 1;
    // Baby steps: K^i mod M
    for (ll i = 0; i < m; ++i) {
        if (!table.count(e))
            table[e] = i;
        e = mod_mul(e, K, M);
    }
    // Compute K^{-m} mod M
    ll factor = mod_inv(mod_pow(K, m, M), M);
    e = result % M;
    // Giant steps
    for (ll j = 0; j <= m; ++j) {
        if (table.count(e)) {
            ll n = j * m + table[e];
            return n; // Minimal n found
        }
        e = mod_mul(e, factor, M);
    }
    return -1; // No solution found (should not happen as per problem statement)
}

```

```
int main() {
    int T;
    scanf("%d", &T); // Read number of test cases
    for (int case_num = 1; case_num <= T; ++case_num) {
        ll K, result;
        scanf("%lld %lld", &K, &result); // Read K and result for each test case
        ll n = bsgs(K % M, result % M); // Compute minimal n using BSGS algorithm
        printf("Case %d: %lld\n", case_num, n); // Output the result
    }
    return 0;
}
```

Source Code 7: Distributing Chocolates

8 Number Systems

Trailing Zeroes (I) [13]

We know what a base of a number is and what the properties are. For example, we use decimal number system, where the base is **10** and we use the symbols - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**. But in different bases we use different symbols. For example in binary number system we use only **0** and **1**.

Now in this problem, you are given an integer. You can convert it to any base you would want to. But the condition is that if you convert it to any base then the number in that base should have at least one trailing zero, that means a zero at the end.

For example, in decimal number system **2** doesn't have any trailing zero. But if we convert it to binary then 2 becomes $(10)_2$ and it contains a trailing zero. Given this task, you have to find the number of bases where the given number contains at least one trailing zero. You can use any base from two to infinity.

Input

Input starts with an integer $T(\leq 10000)$, denoting the number of test cases.

Each case contains an integer $N(1 \leq N \leq 10^{12})$.

Output

For each case, print the case number and the number of possible bases where N contains at least one trailing zero.

Examples

input:
3
9
5
2

output:
Case 1: 2
Case 2: 1
Case 3: 1

Note

For **9**, the possible bases are: **3** and **9**. Since in base **3**; **9** is represented as **100**, and in base **9**; **9** is represented as **10**. In both bases, **9** contains a trailing zero.

Solution

```cpp
#include <bits/stdc++.h>
using namespace std;
#define m 1000010

long long int primes[m], cnt, siv[m];

void sieve()
{
    long long int i, j;
    for (i=3; i<m; i+=2)
        if(!siv[i])
            for (j=i*i; j<m; j+=i+i)
                siv[j]=1;
    primes[cnt++]=2;
    for (i=3; i<m; i+=2)
        if(!siv[i]) primes[cnt++]=i;
    return;
}



int main()
{
    sieve();
    long long int t,c=0;
    scanf("%lld",&t);
    while(t--)
    {
        long long int n,sum=1,s,k,i;
        scanf("%lld",&n);

        // finding the number of divisors of N
        for(i=0;i<m && primes[i]*primes[i]<=n;i++)
        {
            if(n%primes[i]==0)
            {
                k=0;
                while(n%primes[i]==0)
                {
                    n/=primes[i];
                    k++;
                    if(n==0 || n==1)
                        break;
                }
                sum*=k+1;
            }

        }

        /* If the number N is divided by a prime number than the
        sum has to be multiped by (1+1) where first 1 is the count of
        that prime number, N is divisible by and second 1 is the plus one
        of the formula. */

        if(n!=1)
            sum*=2;
        printf("Case %lld: %lld\n",++c,sum-1);
    }
    return 0;
}
```

Source Code 8: Trailing Zeroes (I)

9 Primitive Root

Product of coprimes [14]

You are given a positive integer **m**. Calculate the product of all positive integers less then or equal to **m** and coprime with **m**, and give the answer modulo **m**.

Input

The only line of input file contains a positive integer $m \leq 10^{18}$.

Output

In the output file you should write the answer to the task.

Examples

input:
1
output:
0

Solution

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef uint64_t ull;
typedef __uint128_t u128;

// Function to perform modular multiplication safely
ull mod_mul(ull a, ull b, ull mod) {
    return ((u128)a * b) % mod;
}

// Function to perform modular exponentiation
ull mod_pow(ull base, ull exponent, ull mod) {
    ull result = 1;
    base %= mod;
    while (exponent > 0) {
        if (exponent & 1)
            result = mod_mul(result, base, mod);
        base = mod_mul(base, base, mod);
        exponent >>= 1;
    }
    return result;
}

// Miller-Rabin primality test
bool is_prime(ull n) {
    if (n < 2)
        return false;
    if (n <= 3)
        return true;
    if (n % 2 == 0)
        return false;

    // Write n-1 as d * 2^s by factoring powers of 2 from n-1
    ull s = 0;
    ull d = n - 1;
    while (d % 2 == 0) {
        d >>= 1;
        s++;
    }

    // Witnesses for deterministic test up to 1e18
    ull witnesses[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41};

    for (ull a : witnesses) {
        if (a >= n)
            continue;
        ull x = mod_pow(a, d, n);
        if (x == 1 || x == n - 1)
            continue;
        bool composite = true;
        for (ull r = 1; r < s; ++r) {
            x = mod_mul(x, x, n);
            if (x == n - 1) {
                composite = false;
                break;
            }
        }
        if (composite)
            return false;
    }
    return true;
}

int main() {
    ull m;
    cin >> m;

    if (m == 1) {
        cout << 0 << endl;
    } else if (is_prime(m)) {
        cout << m - 1 << endl;
    } else {
        cout << 1 << endl;
    }

    return 0;
}
```

Source Code 9: Product of coprimes

References

1. https://math.mit.edu/research/highschool/primes/circle/documents/2023/Roonak$_{a}nd_{C}athal.pdf$
2. https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html
3. https://davidaltizio.web.illinois.edu/ModularArithmetic.pdf
4. https://mit6875.github.io/HANDOUTS/numbertheory-lecnotes.pdf
5. https://www.math.brown.edu/johsilve/frint.html
6. Task 1: Zeus and The Birthday Paradox, Codeforces Problem 711E
7. Task 2: Continued Fractions, Codeforces Contest 305B
8. Task 3: How Many Points?, LightOJ Problem
9. Task 4: Efficient Pseudo Code, LightOJ Problem
10. Task 5: Mathematically Hard, LightOJ Problem
11. Task 6: Remainders Game, Codeforces Problem 687B
12. Task 7: Distributing Chocolates, LightOJ Problem
13. Task 8: Trailing Zeroes (I), LightOJ Problem
14. Task 9: Product of Coprimes, E-Olymp Problem 647