GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: PRUEBAS DE PRIMALIDAD

1. Introducción

Las pruebas de primalidad son algoritmos utilizados en matemáticas y computación para determinar si un número dado es primo o no. Un número primo es aquel que solo es divisible por sí mismo y por 1, es decir, no tiene otros divisores más que esos dos.

2. Conocimientos previos

2.1. Número primo

Un número primo es un número natural mayor que 1 que solamente es divisible por sí mismo y por 1, es decir, no tiene más divisores que esos dos números. Algunos ejemplos de números primos son 2, 3, 5, 7, 11, 13, 17, etc. Los números primos son fundamentales en matemáticas y tienen muchas propiedades interesantes.

2.2. Pierre de Fermat

Pierre de Fermat fue un matemático francés del siglo XVII conocido por sus contribuciones a la teoría de números, geometría analítica y cálculo. Una de las contribuciones más famosas de Fermat es el *Teorema de Fermat*, que enunció en una nota al margen de su ejemplar de la obra de Diofanto. El teorema establece que no existen enteros positivos a, b, c y n mayores que 2 que satisfagan la ecuación $a^n + b^n = c^n$. Este teorema fue uno de los problemas abiertos más famosos en matemáticas y se conoció como el *Último Teorema de Fermat*. Fue finalmente demostrado por Andrew Wiles en 1994 utilizando técnicas avanzadas de matemáticas modernas.

2.3. Exponenciación binaria

La exponenciación binaria es un método eficiente para calcular potencias de un número. Consiste en descomponer el exponente en su representación binaria y realizar operaciones con las potencias de 2.

Este método es eficiente porque reduce el número de multiplicaciones requeridas para calcular la potencia, ya que solo se realizan multiplicaciones cuando el bit correspondiente en la representación binaria del exponente es 1.

2.4. Garry Miller

Gary Miller es un matemático y científico de la computación estadounidense conocido por su trabajo en teoría de números y algoritmos. Es especialmente reconocido por sus contribuciones al campo de la criptografía y la computación numérica.

2.5. Michael O. Rabin

Michael O. Rabin es un matemático e informático israelí-estadounidense conocido por sus contribuciones a la teoría de la computación, la criptografía y la complejidad computacional. Es uno

de los pioneros en el campo de la informática teórica y ha realizado importantes investigaciones en varios aspectos fundamentales de la ciencia de la computación.

3. Desarrollo

Existen diferentes métodos para probar la primalidad de un número, algunos de los cuales son más eficientes que otros dependiendo del tamaño del número a evaluar. A continuación vamos a explicar algunos de ellos.

3.1. Prueba por división

Por definición, un número primo no tiene más divisores que 1 y a sí mismo. Un número compuesto tiene al menos un divisor adicional, llamémoslo d. Naturalmente $\frac{n}{d}$ también es divisor de n. Es fácil ver que tampoco $d \leq \sqrt{n}$ o $\frac{n}{d} \leq \sqrt{n}$, por lo tanto uno de los divisores d y $\frac{n}{d}$ es $\leq \sqrt{n}$. Podemos utilizar esta información para comprobar la primalidad.

Intentamos encontrar un divisor no trivial, comprobando si alguno de los números entre 2 y $\sqrt{n}$ es un divisor de n. Si es divisor, entonces n Definitivamente no es primo, de lo contrario lo es.

Esta es la forma más simple de verificación primaria. Puede optimizar bastante esta función, por ejemplo, verificando solo todos los números impares en el ciclo, ya que el único número primo par es 2. En el artículo sobre factorización de enteros se describen múltiples optimizaciones de este tipo .

3.2. Prueba de primalidad de Fermat

Esta es una prueba probabilística. El pequeño teorema de Fermat establece que para un número primo p y un entero coprimo a se cumple la siguiente ecuación:

$$a^{p-1} \equiv 1 \text{ mód } p$$

En general, este teorema no se cumple para los números compuestos.

Esto se puede utilizar para crear una prueba de primalidad. Escogemos un número entero $2 \leq a \leq p-2$ y comprueba si la ecuación se cumple o no. Si no se sostiene, por ejemplo$a^{p-1} \not\equiv 1 \text{ mód } p$, lo sabemos p no puede ser un número primo. En este caso llamamos a la base a un testigo Fermat de la composición de p.

Sin embargo, también es posible que la ecuación sea válida para un número compuesto. Entonces, si la ecuación se cumple, no tenemos una prueba de primalidad. Solo podemos decir eso p probablemente sea primo. Si resulta que el número es realmente compuesto, llamamos a la base a un mentiroso de Fermat.

Ejecutando la prueba para todas las bases posibles a, podemos demostrar que un número es primo. Sin embargo, esto no se hace en la práctica, ya que supone mucho más esfuerzo que simplemente hacer una división de prueba. En lugar de ello, la prueba se repetirá varias veces con

opciones aleatorias para a. Si no encontramos ningún testigo de la composición, es muy probable que el número sea primo.

Sin embargo, hay una mala noticia: existen algunos números compuestos en los que $a^{n-1} \equiv 1 \mod n$ vale para todos a coprimo a n, por ejemplo para el número $561 = 3 \cdot 11 \cdot 17$. Estos números se denominan números de Carmichael. El test de primalidad de Fermat puede identificar estos números sólo si tenemos mucha suerte y elegimos una base a con $\gcd(a, n) \neq 1$.

La prueba de Fermat todavía se utiliza en la práctica, ya que es muy rápida y los números de Carmichael son muy raros. Por ejemplo, solo existen 646 números de este tipo a continuación 10^9.

3.3. Prueba de primalidad de Miller-Rabin

La prueba de Miller-Rabin amplía las ideas de la prueba de Fermat. Para un número impar n, $n-1$ es par y podemos factorizar todas las potencias de 2. Podemos escribir:

$$n - 1 = 2^s \cdot d, \text{ con } d \text{ impar.}$$

Esto nos permite factorizar la ecuación del pequeño teorema de Fermat:

$$\begin{aligned}
a^{n-1} \equiv 1 \mod n \quad &\iff a^{2^s d} - 1 \equiv 0 \mod n \\
&\iff (a^{2^{s-1} d} + 1)(a^{2^{s-1} d} - 1) \equiv 0 \mod n \\
&\iff (a^{2^{s-1} d} + 1)(a^{2^{s-2} d} + 1)(a^{2^{s-2} d} - 1) \equiv 0 \mod n \\
&\vdots \\
&\iff (a^{2^{s-1} d} + 1)(a^{2^{s-2} d} + 1) \cdots (a^{d} + 1)(a^{d} - 1) \equiv 0 \mod n
\end{aligned}$$

Si n es primo, entonces n tiene que dividir uno de estos factores. Y en la prueba de primalidad de Miller-Rabin verificamos exactamente esa afirmación, que es una versión más estricta de la afirmación de la prueba de Fermat para una base $2 \leq a \leq n-2$ comprobamos si alguno:

$$a^d \equiv 1 \mod n$$

sostiene o

$$a^{2^r d} \equiv -1 \mod n$$

se mantiene para algunos $0 \leq r \leq s-1$.

Si encontramos una base a que no satisface ninguna de las igualdades anteriores, entonces encontramos un testigo de la composición de n. En este caso hemos demostrado que n no es un número primo.

De manera similar a la prueba de Fermat, también es posible que el conjunto de ecuaciones se cumpla para un número compuesto. En ese caso la base a Se le llama mentiroso fuerte si una base a satisface las ecuaciones (una de ellas), n es solo un primo probable fuerte. Sin embargo, no existen números como los números de Carmichael, donde se encuentran todas las bases no triviales. De hecho, es posible demostrar que, como máximo, $\frac{1}{4}$ de las bases pueden ser fuertes mentirosos. Si n es compuesto, tenemos una probabilidad de $\geq 75\,\%$ que una base aleatoria nos dirá que es compuesta. Al realizar múltiples iteraciones y elegir diferentes bases aleatorias, podemos saber con muy alta probabilidad si el número es verdaderamente primo o si es compuesto.

Antes de realizar la prueba de Miller-Rabin, puedes comprobar además si uno de los primeros números primos es divisor. Esto puede acelerar mucho la prueba, ya que la mayoría de los números compuestos tienen divisores primos muy pequeños. P.ej $88\,\%$ de todos los números tienen un factor primo menor que 100.

3.4. Versión determinista

Miller demostró que es posible hacer que el algoritmo sea determinista comprobando únicamente todas las bases $\leq O((\ln n)^2)$. Bach luego dio un límite concreto, solo es necesario probar todas las bases $a \leq 2\ln(n)^2$.

Sigue siendo una cantidad bastante grande de bases. Por eso la gente ha invertido bastante poder de cálculo en encontrar límites inferiores. Resulta que para probar un entero de 32 bits sólo es necesario comprobar las primeras 4 bases primas: 2, 3, 5 y 7. El número compuesto más pequeño que no pasa esta prueba es $3,215,031,751 = 151 \cdot 751 \cdot 28351$. Y para probar números enteros de 64 bits basta con comprobar las primeras 12 bases primas: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 y 37.

También es posible hacer la verificación con solo 7 bases: 2, 325, 9375, 28178, 450775, 9780504 y 1795265022. Sin embargo, como estos números (excepto 2) no son primos, es necesario verificar adicionalmente si el número que estás verificando es igual a cualquier divisor primo de esas bases: 2, 3, 5, 13, 19, 73, 193, 407521, 299210837.

4. Implementación

4.1. C++

4.1.1. Prueba por división

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

```
}
```

4.2. Prueba de primalidad de Fermat

```
bool probablyPrimeFermat(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 (binpower(a, n - 1, n) != 1) return false;
   }
   return true;
}
```

Usamos la exponenciación binaria para calcular eficientemente la potencia a^{p-1}.

4.3. Prueba de primdealidad Miller-Rabin

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

u64 binpower(u64 base, u64 e, u64 mod) {
   u64 result = 1;
   base %= mod;
   while (e) {
      if (e & 1)
        result = (u128)result * base % mod;
      base = (u128)base * base % mod;
      e >>= 1;
   }
   return result;
}

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, int iter=5) {
//devuelve verdadero si n probablemente sea primo; en caso contrario, devuelve
    falso.
   if (n < 4) return n == 2 || n == 3;
   int s = 0;
```

```
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        s++;
    }
    for (int i = 0; i < iter; i++) {
        int a = 2 + rand() % (n - 3);
        if (check_composite(n, a, d, s))
            return false;
    }
    return true;
}
```

4.3.1. Versión determinista

```
bool MillerRabin(u64 n) {
//devuelve verdadero si n probablemente sea primo; en caso contrario, devuelve
     falso.
    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;
}
```

4.4. Java

4.4.1. Prueba por división

```
public static boolean isPrime(int x) {
    for (int d = 2; d * d <= x; d++) {
        if (x % d == 0) return false;
    }
    return x >= 2;
}
```

4.5. Prueba de primalidad de Fermat

```
import java.util.Random;

public long modPow(long a, long b, long c) {
```

```
    long res = 1;
    for (int i = 0; i < b; i++) res = (res * a) % c;
    return res % c;
}

public boolean isPrimeFermat(long n, int iteration) {
    if (n == 0 || n == 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) {
        Random rand = new Random();
        for (int i = 0; i < iteration; i++) {
            long r = Math.abs(rand.nextLong());
            long a = r % (n - 1) + 1;
            if (modPow(a, n - 1, n) != 1) return false;
        }
        return true;
    }
    return false;
}
```

4.6. Prueba de primalidad de Miller-Rabin

```
import java.util.Random;
import java.math.BigInteger;

public long modPow(long a, long b, long c) {
    long res = 1;
    for (int i = 0; i < b; i++) res *= a;
    return res % c;
}

public long mulMod(long a, long b, long mod) {
    return BigInteger.valueOf(a).multiply(BigInteger.valueOf(b)).mod(BigInteger
        .valueOf(mod)).longValue();
}

public boolean isPrimeMillerRabin(long n, int iteration) {
    if (n == 0 || n == 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) {
        long s = n - 1;
        while (s % 2 == 0) s /= 2;
        Random rand = new Random();
        for (int i = 0; i < iteration; i++) {
            long r = Math.abs(rand.nextLong());
            long a = r % (n - 1) + 1, temp = s;
            long mod = modPow(a, temp, n);
            while (temp != n - 1 && mod != 1 && mod != n - 1) {
```

```
            mod = mulMod(mod, mod, n);
            temp *= 2;
         }
         if (mod != n - 1 && temp % 2 == 0) return false;
      }
      return true;
   }
   return false;
}
```

4.6.1. Versión deterministas

```
import java.util.Random;

private static boolean checkComposite(long n, long a, long d, int s) {
   long x = powerModP(a, d, n);
   if (x == 1 || x == n - 1) return false;
   for (int r = 1; r < s; r++) {
      x = powerModP(x, 2, n);
      if (x == n - 1) return false;
   }
   return true;
}

private static long powerModP(long x, long y, long p) {
   long res = 1;
   x = x % p;
   if (x == 0) return 0;
   while (y > 0) {
      if ((y & 1) == 1) res = multiplyModP(res, x, p);
      y = y >> 1;
      x = multiplyModP(x, x, p);
   }
   return res;
}

private static long multiplyModP(long a, long b, long p) {
   long aHi = a >> 24;
   long aLo = a & ((1 << 24) - 1);
   long bHi = b >> 24;
   long bLo = b & ((1 << 24) - 1);
   long result = ((((aHi * bHi << 16) % p) << 16) % p) << 16;
   result += ((aLo * bHi + aHi * bLo) << 24) + aLo * bLo;
   return result % p;
}

public static boolean isPrime(long n) {
   if (n < 2) return false;
```

```
    int r = 0;
    long d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1; r++;
    }
    for (int a : new int[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a) return true;
        if (checkComposite(n, a, d, r)) return false;
    }
    return true;
}
```

5. Aplicaciones

Las pruebas de primalidad tienen diversas aplicaciones en diferentes campos, entre las cuales se pueden mencionar:

1. **Criptografía:** En criptografía, los números primos juegan un papel fundamental en la generación de claves seguras. Los algoritmos de encriptación y firma digital utilizan números primos en sus operaciones, por lo que es importante poder verificar si un número dado es primo. Las pruebas de primalidad se utilizan para garantizar la seguridad de los sistemas criptográficos.
2. **Generación de números aleatorios:** En la generación de números aleatorios, es común utilizar números primos para asegurar la aleatoriedad y la calidad de los números generados. Las pruebas de primalidad permiten verificar si un número es primo antes de ser utilizado en algoritmos de generación de números aleatorios.
3. **Algoritmos de factorización:** La factorización de números enteros en sus factores primos es un problema importante en matemáticas y computación. Las pruebas de primalidad se utilizan en algoritmos de factorización para determinar si un número es primo y así facilitar el proceso de descomposición en factores primos.
4. **Teoría de números computacional:** En el campo de la teoría de números computacional, las pruebas de primalidad son fundamentales para investigar propiedades de los números primos, estudiar distribuciones y patrones, y resolver problemas matemáticos complejos relacionados con los números primos.
5. **Optimización y búsqueda de números primos grandes:** En la búsqueda de números primos grandes, como los utilizados en registros de primos o en concursos matemáticos, las pruebas de primalidad son esenciales para verificar si un número dado es primo y contribuir al descubrimiento de nuevos números primos.

Estas son solo algunas de las aplicaciones más comunes de las pruebas de primalidad. La teoría de números y las pruebas de primalidad tienen una amplia gama de aplicaciones en diversos campos, desde la criptografía hasta la investigación matemática.

6. Complejidad

La complejidad de la prueba basada en la división es O($\sqrt{N}$). La complejidad temporal de la prueba de primalidad de Fermat es (O($k \times (\log n)^{2+\epsilon}$)), donde k es el número de veces que se ejecuta el test y determina la fiabilidad del test, y n es el número natural mayor que 1 que se está probando por primalidad. La complejidad temporal de la prueba de primalidad de Miller-Rabin es (O($k \ln^3 n$)), donde n es el número que se está probando por primalidad y k es el número de rondas realizadas. Este algoritmo es eficiente en términos de tiempo polinomial en relación con la cantidad de dígitos de n. La versión determinista de la prueba de Primalidad de Miller-Rabin tiene una complejidad temporal de O($(\log n)^2$)).

7. Ejercicios

A continuación un grupo de ejercicios que se pueden resolver aplicando los contenidos explicados:

- SPOJ - Prime or Not
- Project euler - Investigating a Prime Pattern