素性检测探究报告

2212000 宋奕纬

June 2024

1 问题定义

素性检测（Primality Testing）是指判定一个给定的整数是否为素数的过程。素数是仅能被 1 和自身整除的自然数，在数论和密码学中具有重要地位。有效的素性检测算法是现代密码学（如 RSA 加密算法）的基础。素数的稀有性和难以预测性使其在加密领域尤为重要。

2 应用领域

素性检测（Primality Testing）在数论和密码学中具有重要地位。它主要用于确定一个给定的整数是否为素数。素数的稀有性和难以预测性使其在加密领域尤为重要。以下将详细介绍素性检测在五个主要领域的应用。

2.1 RSA 加密算法

RSA 加密算法是最广泛使用的公钥加密算法之一，其安全性依赖于大素数的生成。在 RSA 密钥生成过程中，需要选取两个大素数 p 和 q，并通过素性检测算法（如 Miller-Rabin 测试）来验证这些数是否为素数。只有确保 p 和 q 为素数，才能生成安全的公钥和私钥对，用于加密和解密数据。通过有效的素性检测算法，可以防止选取的数为合数，从而提高 RSA 加密的安全性。

2.2 Diffie-Hellman 密钥交换协议

Diffie-Hellman 密钥交换协议是用于安全地交换加密密钥的公钥协议。该协议要求选择一个大素数 p 和一个基数 g，通过素性检测确保 p 为素数，

以防止中间人攻击和其他安全威胁。素性检测在这个过程中起到关键作用，确保所选的大数 p 确实是素数，从而保证密钥交换的安全性和可靠性。通过这种方式，通信双方可以在不直接传输密钥的情况下安全地共享密钥。

2.3 数字签名

数字签名用于验证数据的完整性和真实性，广泛应用于电子商务、在线通信等领域。许多数字签名算法（如 DSA）依赖于大素数来生成签名密钥。通过素性检测，可以确保选取的大数为素数，从而生成安全的签名密钥。这些密钥用于生成和验证数字签名，防止数据被篡改或伪造，保障通信的安全性和可靠性。

2.4 加密安全的随机数生成器

在加密领域，高质量的随机数是确保算法安全性的关键。加密安全的随机数生成器通过生成大素数来提升随机数的质量和安全性。素性检测用于验证生成的大数是否为素数，确保随机数的质量。高质量的随机数用于生成加密密钥、盐值等，在加密算法中发挥重要作用，提升系统的安全性和抗攻击能力。

2.5 公钥基础设施（PKI）

公钥基础设施（PKI）系统用于管理和分发公钥和私钥，确保数据传输的安全性。在 PKI 系统中，生成的密钥需要通过素性检测算法验证，以确保其安全性和有效性。素性检测确保所选的大数为素数，从而生成安全的密钥对，用于加密、解密和数字签名。通过素性检测，PKI 系统能够有效防止因选取不合适的密钥而导致的安全漏洞，保障数据的安全传输。

3 常用算法

接下来将介绍常用的素性检测算法，包括试除法、费马素性测试、Miller-Rabin 测试、AKS 素性测试和 Solovay-Strassen 测试。

3.1 试除法

试除法是最基本的素性检测方法。其基本思路是依次除以小于其平方根的所有整数来判断一个数是否为素数。如果一个数 n 不能被 2 到 $\sqrt{n}$ 之间的任何整数整除，则 n 为素数。该方法的时间复杂度为 $O(\sqrt{n})$，不适用于大数的素性检测。

C++ 实现

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

bool isPrime(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;
    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return false;
    }
    return true;
}

int main() {
    int n = 29; // 待检测的数
    if (isPrime(n)) {
        std::cout << n << " is prime." << std::endl;
    } else {
        std::cout << n << " is not prime." << std::endl;
    }
    return 0;
}
```

3.2 费马素性测试

费马素性测试基于费马小定理。费马小定理指出，若 p 是素数，则对于任意整数 a $(1 < a < p)$，有 $a^{(p-1)} \equiv 1 \pmod{p}$。费马测试通过选择多个随机数 a，验证上述等式是否成立。若等式不成立，则 n 为合数；若等式成立，则 n 可能为素数。该方法简单高效，但存在费马伪素数。

C++ 实现

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

```

```cpp
using namespace std;

// 快速幂模运算
long long power(long long base, long long exp, long long mod) {
    long long result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

// 费马素性测试
bool fermatTest(int n, int k) {
    if (n <= 1) return false;
    if (n <= 3) return true;

    srand(time(0));
    for (int i = 0; i < k; i++) {
        int a = 2 + rand() % (n - 3);
        if (power(a, n - 1, n) != 1) {
            return false;
        }
    }
    return true;
}

int main() {
    int n = 29; // 待检测的数
    int k = 5; // 测试次数

    if (fermatTest(n, k)) {
        cout << n << " is probably prime." << endl;
    } else {
        cout << n << " is composite." << endl;
    }
    return 0;
}
```

3.3 Miller-Rabin 测试

Miller-Rabin 测试是一种基于费马小定理的改进随机化算法，通过多次随机选择测试值来提高准确性。

3.3.1 算法原理

Miller-Rabin 测试基于以下数学理论：

1. 将 $n-1$ 表示为 $2^r \times d$，其中 d 是奇数。

2. 对于随机选择的整数 a（$2 \leq a \leq n-2$），计算 $a^d \pmod{n}$。

3. 如果 $a^d \equiv 1 \pmod{n}$ 或者 $a^{(2^j \cdot d)} \equiv -1 \pmod{n}$（其中 $0 \leq j \leq r-1$），则 n 可能为素数，否则 n 为合数。

3.3.2 算法步骤

1. 选择待检测的奇数 n。

2. 将 $n-1$ 分解为 $2^r \times d$，其中 d 为奇数。

3. 选择多个随机数 a，每个 a 满足 $2 \leq a \leq n-2$。

4. 对每个 a：

   (a) 计算 $x = a^d \pmod{n}$。

   (b) 如果 $x = 1$ 或 $x = n-1$，则继续测试下一个 a。

   (c) 否则，重复以下步骤 $r-1$ 次：

      i. 计算 $x = x^2 \pmod{n}$。

      ii. 如果 $x = n-1$，则继续测试下一个 a。

      iii. 如果 $x = 1$，则 n 为合数。

   (d) 如果没有发现 $x = n-1$，则 n 为合数。

5. 如果所有 a 都通过测试，则 n 可能为素数。

3.3.3 C++ 实现

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
```

```cpp

using namespace std;

// 快速幂模运算
long long power(long long base, long long exp, long long mod) {
    long long result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

// Miller-Rabin测试
bool millerRabinTest(long long d, long long n) {
    long long a = 2 + rand() % (n - 4);
    long long x = power(a, d, n);

    if (x == 1 || x == n - 1) {
        return true;
    }

    while (d != n - 1) {
        x = (x * x) % n;
        d *= 2;

        if (x == 1) return false;
        if (x == n - 1) return true;
    }

    return false;
}

// 检测一个数是否为素数
bool isPrime(long long n, int k) {
    if (n <= 1 || n == 4) return false;
    if (n <= 3) return true;

    long long d = n - 1;
    while (d % 2 == 0) {
        d /= 2;
    }
```

```
50
51      for (int i = 0; i < k; i++) {
52          if (!millerRabinTest(d, n)) {
53              return false;
54          }
55      }
56
57      return true;
58  }
59
60  int main() {
61      srand(time(0));
62      long long n = 31; // 待检测的数
63      int k = 4; // 测试次数
64
65      if (isPrime(n, k)) {
66          cout << n << " is prime." << endl;
67      } else {
68          cout << n << " is not prime." << endl;
69      }
70
71      return 0;
72  }
```

3.4 AKS 素性测试

AKS 素性测试是一种确定性算法，可以在多项式时间内判断一个数是否为素数。该算法基于多项式同余理论。

3.4.1 算法原理

AKS 算法基于以下数学定理：

整数 $n(\geq 2)$ 是素数，当且仅当

$$(x+a)^n \equiv (x^n + a) \pmod{n} \tag{1}$$

这个同余多项式对所有与 n 互素的整数 a 均成立。这个定理是费马小定理的一般化，并且可以通过这个特征：

$$\binom{n}{k} \equiv 0 \pmod{n}, \quad \text{对任何} 0 < k < n \text{ 当且仅当} n \text{ 是素数}$$

来证明此定理。

虽然说关系式 (1) 基本上构成了整个素数测试，但是验证花费的时间却是指数时间。因此，为了减少计算复杂度，AKS 改为使用以下的同余多项式：

$$(x+a)^n \equiv (x^n+a) \pmod{x^r-1, n} \tag{2}$$

这个多项式与存在多项式 g, f 令：

$$(x+a)^n - (x^n+a) = (x^r-1)g + nf \tag{3}$$

意义是等同的。

这个同余式可以在多项式时间之内检查完毕。这里我们要注意所有的素数必定满足此条件 (令 $g=0$ 则 (3) 等于 (1)，因此符合 n 必定是素数)。然而，有一些合数也会满足这个条件式。有关系 AKS 正确性的证明包括了推导出存在一个足够小的 r 以及一个足够小的整数集合 A，令如果此同余式对所有 A 里面的整数都满足，则 n 必定为素数。

具体步骤如下：

1. 检查 n 是否是一个完全幂。
2. 找到最小的 r，使得 $o_r(n) > (\log_2 n)^2$。
3. 检查 $1 < \gcd(a, n) < n$ 的 a。
4. 对于 $a = 1$ 到 $\lfloor \sqrt{\phi(r)} \log_2 n \rfloor$，验证多项式同余关系。
5. 如果 n 通过上述所有测试，则为素数，否则为合数。

3.4.2 C++ 实现

由于 AKS 算法比较复杂，此处为简化后的：

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

using namespace std;

// 检查n是否是一个完全幂
bool isPerfectPower(int n) {
    for (int b = 2; b <= log2(n); ++b) {
        int a = pow(n, 1.0 / b);
        if (pow(a, b) == n || pow(a + 1, b) == n) {
```

```
            return true;
        }
    }
    return false;
}

// 快速幂模运算
long long power(long long base, long long exp, long long mod) {
    long long result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

// 计算最大公约数
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

// AKS素性测试
bool aksTest(int n) {
    if (n <= 1) return false;
    if (isPerfectPower(n)) return false;

    // Step 2: 找到最小的r, 使得 o_r(n) > (log2(n))^2
    int r = 1;
    int maxK = log2(n) * log2(n);
    while (true) {
        ++r;
        bool nextR = false;
        for (int k = 1; k <= maxK; ++k) {
            if (power(n, k, r) == 0 || power(n, k, r) == 1) {
                nextR = true;
                break;
            }
        }
        if (!nextR) break;
    }

```

```
    // Step 3: 检查1 < gcd(a, n) < n的a
    for (int a = 2; a < min(r, n); ++a) {
        if (gcd(a, n) > 1 && gcd(a, n) < n) return false;
    }

    // Step 4: 验证多项式同余关系
    for (int a = 1; a <= sqrt(phi(r)) * log2(n); ++a) {
        if (power(a, n, n) != power(a, 1, n)) return false;
    }

    return true;
}

int main() {
    int n = 29; // 待检测的数

    if (aksTest(n)) {
        cout << n << " is prime." << endl;
    } else {
        cout << n << " is composite." << endl;
    }
    return 0;
}
```

3.5 Solovay-Strassen 测试

Solovay-Strassen 测试是一种基于雅可比符号的随机化算法。

3.5.1 算法原理

Solovay-Strassen 测试基于以下数学定理：如果 p 是素数，对于任何整数 a，满足 $a^{(p-1)/2} \equiv (\frac{a}{p}) \pmod{p}$。该算法通过验证多个随机数 a 的雅可比符号和指数幂之间的关系来判断素性。

先输入奇整数 $n \geq 3$ 和安全参数 t （计算次数）

1. 随机选取整数 b，b 的范围是 $[2, n-2]$。
2. 计算 $r = b^{(n-1)/2} \pmod{n}$。
3. 如果 $r \neq 1$ 且 $r \neq n-1$，则 n 是合数。
4. 计算 Jacobi 符号 $s = \left(\frac{a}{n}\right)$。

5. 如果 $r \neq s$，则 n 是合数。

6. 上述过程重复 t 次。

3.5.2 C++ 实现

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

// 计算雅可比符号
int jacobi(int a, int n) {
    if (a == 0) return 0;
    if (a == 1) return 1;
    int ans;
    if (a % 2 == 0) {
        ans = jacobi(a / 2, n);
        if (n % 8 == 3 || n % 8 == 5) ans = -ans;
    } else {
        ans = jacobi(n % a, a);
        if (a % 4 == 3 && n % 4 == 3) ans = -ans;
    }
    return ans;
}

// 快速幂模运算
long long power(long long base, long long exp, long long mod) {
    long long result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

// Solovay-Strassen测试
bool solovayStrassenTest(int n, int k) {
    if (n <= 1) return false;
    if (n % 2 == 0) return false;

```

```
    srand(time(0));
    for (int i = 0; i < k; i++) {
        int a = 2 + rand() % (n - 2);
        int jacobian = (n + jacobi(a, n)) % n;
        int mod = power(a, (n - 1) / 2, n);
        if (jacobian == 0 || mod != jacobian) return false;
    }
    return true;
}

int main() {
    int n = 29; // 待检测的数
    int k = 5; // 测试次数

    if (solovayStrassenTest(n, k)) {
        cout << n << " is probably prime." << endl;
    } else {
        cout << n << " is composite." << endl;
    }
    return 0;
}
```

4 总结

素性检测是数论和密码学中的一个重要问题。有效的素性检测算法能够快速判定一个大数是否为素数，这在加密算法如 RSA 和 Diffie-Hellman 中尤为关键。常用的素性检测算法包括试除法、费马素性测试、Miller-Rabin 测试、AKS 素性测试和 Solovay-Strassen 测试。素性检测在密码学中的应用尤为重要，确保了加密密钥的安全性和数据传输的保密性。未来的研究可以继续优化现有算法，提高大数素性检测的效率，开发新的算法以应对更大规模数的素性检测需求。