밀러-라빈 소수 판별법

2015004002 김경한

목차

1. 모듈러 연산

나머지를 구하는 연산

모듈러 연산의 정의

나머지 정리

임의의 정수 A와 양의 정수 B가 주어질 때,
다음을 만족하는 유일한 정수 Q와 R이
존재합니다.

$$A = B \times Q + R, 0 \leq R < B$$

모듈러 연산

임의의 정수 A와 양의 정수 B가 주어질 때,

$$A = B \times Q + R, 0 \leq R < B$$

를 만족하는 정수 $R = A \ mod \ B$ 입니다.

모듈러 연산의 성질

덧셈, 뺄셈

$(A + B) \ mod \ C =$

$(A \ mod \ C + B \ mod \ C) \ mod \ C$

$(A - B) \ mod \ C =$

$(A \ mod \ C \ - B \ mod \ C) \ mod \ C$

곱셈

$(A \times B) \ mod \ C =$

$(A \ mod \ C \times B \ mod \ C) \ mod \ C$

나눗셈…?

$(A \div B) \ mod \ C =$

$(A \times B^{-1}) \ mod \ C =$

$(A \ mod \ C \times B^{-1} \ mod \ C) \ mod \ C$

모듈러 역수

- 곱셈에서 역수란, 어떤 수를 자신의 역수로 곱했을 때 1이 나오게 하는 수입니다.
- A의 역수에 수를 곱하는 것은 그 수로 A를 나누는 것과 같습니다.
- 그래서 모듈러 연산에서는 나누기 대신 모듈러 역수를 씁니다.
- $(A \times X)\ mod\ C = 1$를 만족하는 $X = A^{-1}$, $A\ (mod\ C)$의 모듈러 역수입니다.

모듈러 거듭제곱법

A의 제곱

$A^2 \, mod \, C = (A \times A) \, mod \, C =$

$(A \, mod \, C \times A \, mod \, C) \, mod \, C =$

$(A \, mod \, C)^2 \, mod \, C$

$A^B (B = 2^n)$식의 거듭제곱

$A^{2^n} \, mod \, C =$

$\left(A^{2^{n-1}} \, mod \, C\right)^2 mod \, C =$

$\left(A^{2^{n-2}} \, mod \, C\right)^{2^2} mod \, C = \ldots$

$(A \, mod \, C)^{2^n} \, mod \, C$

임의의 거듭제곱 (Ex. 21제곱)

$B = 21 = 10101_{(2)}$ 이므로

$A^{21} \, mod \, C =$

$(A \times A^4 \times A^{16}) \, mod \, C =$

$\left(A^{2^0} \, mod \, C \times A^{2^2} mod \, C \times A^{2^4} mod \, C\right) mod \, C$

소스 코드

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

int power(int A, int B, int C) {
    int ret = 1;
    while (true) {
        if (B & 1) ret = (ret * A) % C;
        if (B /= 2) A = (A * A) % C;
        else break;
    }
    return ret;
}

int main() {
    int a, b, c;
    cin >> a >> b >> c;
    cout << power(a, b, c) << endl;
    return 0;
}
```

시간 복잡도

while문 : $O(\log B)$

거듭제곱 : $O(\log^2 C)$

총합 : $O(\log^3 C) \quad (if, B \leq C)$

2. 페르마의 소정리

어떤 수가 소수일 필요 조건

페르마의 소정리

정의

임의의 소수 p와 정수 a가 주어졌을 때,
다음을 항상 만족합니다.

$$a^p \equiv a \ (mod\ p)$$

적용

정수 a가 p의 배수가 아니면 양변을 a로 나눠

$$a^{p-1} \equiv 1 \ (mod\ p)$$

와 같이 나타낼 수 있습니다.
단, 이는 모든 소수가 만족시키는 필요조건인
반면 충분조건이 아닙니다. (소수 판별 불가능)

페르마의 소정리

보조정리

임의의 소수 p에 대해, $x^2 \equiv 1 \ (mod\ p)$이면 다음을 항상 만족합니다.

$$x \equiv 1 \ (mod\ p) \ or \ x \equiv -1 \ (mod\ p)$$

증명

모듈러 연산의 정의에 의해

$x^2 - 1 = (x+1)(x-1)$은 p의 배수이고,

따라서 둘 중 하나는 p의 배수여야 합니다.

3. 밀러-라빈 소수판별법

소수를 판별하는 확률적 알고리즘

이론적 배경

소수의 성질

n이 홀수인 소수라고 할 때,
$n-1=2^s d\ (d=2k+1,\ s,k \in N)$라
할 수 있습니다. 그러면 n의 배수가 아닌
어떤 a에 대해 다음 식 중 하나가
성립합니다.

$$\begin{cases} a^d \equiv 1\ (mod\ n) \\ a^{2^r \cdot d} \equiv -1\ (mod\ n), 0 \le r \le s-1 \end{cases}$$

증명

페르마의 소정리에 의해 $a^{n-1} \equiv 1\ (mod\ n)$,
즉 $a^{n-1}-1=a^{2^r \cdot d}-1$
$=\left(a^{2^{r-1} \cdot d}+1\right)\left(a^{2^{r-1} \cdot d}-1\right)=\ \cdots$
$=\left(a^{2^{r-1} \cdot d}+1\right)\left(a^{2^{r-2} \cdot d}+1\right)\cdots\left(a^d-1\right)$
이 중에서 하나는 n의 배수여야 하므로
성질이 성립함을 알 수 있습니다.

확률적 소수 판별

강한 증거

앞에서의 관계를 바탕으로

$$\begin{cases} a^d \not\equiv 1 \ (mod\ n) \\ a^{2^r \cdot d} \not\equiv -1 \ (mod\ n), 0 \le r \le s-1 \end{cases}$$

이 조건을 만족하는 a는 n이 합성수라는 것에 대한 **강한 증거**가 됩니다.

= 이 조건을 만족하는 a가 있으면 n은 항상 합성수입니다.

강한 거짓증거

반대로 조건을 만족하지 못하는 a는 n이 소수"일 것 같다"는 것에 대한 **강한 거짓증거**가 됩니다. 이는 페르마의 소정리가 소수에 대한 필요조건이기 때문입니다.

= 그래서 a를 여러 번 바꿔가며 잘못 판단했을 확률을 낮춰주게 됩니다.

알고리즘 및 시간 복잡도

- 입력 : N = 소수인지 검사할 숫자, K : 테스트를 시행할 횟수
- 출력 : True / False (합성수이다 / 아마 소수일 것이다)
- $N-1=2^s \cdot d$꼴로 바꾼 뒤, $[1, N)$에서 임의의 a를 골라 테스트를 시행합니다.
- 한 번이라도 합성수라고 판단하면 True를 반환, K번 모두 소수라 판단하면 False를 반환

알고리즘 및 시간 복잡도

- 제곱을 반복하는 방법에 따라 시간 복잡도가 결정됩니다.
- 산술연산 : $O(K \log N)$ but overflow...
- 모듈로 거듭제곱 : $O(K \log^3 N)$
- FFT (빠른 푸리에 변환) : $O(K \log^2 N)$

작은 수에 대한 판별

- N이 (unsigned) int 범위일 경우 : a = 2, 7, 61 (K = 3)
- N이 (unsigned) long long 범위 일 경우 : a = 2, 325, 9375, 28178, 450775, 9780504, 1795265022 (K = 7)
- N의 범위가 커서 모듈러 거듭제곱을 이용해야 하는 경우에는, N이 충분히 작다면, 1부터 $\sqrt{N}$까지 전부 나눠보는 방식이 더 빠릅니다. ($N \le 10^{10}$)

소스 코드(N : int 범위)

```cpp
#include <iostream>
using namespace std;
using ll = long long;

const int candidate[3] = {2, 7, 61};

ll power(ll A, int B, int C) {
    ll ret = 1;
    while (true) {
        if (B & 1) ret = (ret * A) % C;
        if (B /= 2) A = (A * A) % C;
        else break;
    }
    return ret;
}

bool miller_rabin(int n, int a) {
    int d = n - 1;
    while (d % 2 == 0) {
        if ((int)power((ll)a, d, n) == n - 1)
            return true;
        d /= 2;
    }
    int tmp = (int)power((ll)a, d, n);
    return tmp == (n - 1) || tmp == 1;
}
```

```cpp
bool is_prime(int n) {
    if (n <= 1) return false;
    if (n <= 1000) {
        for (int i = 2; i * i <= n; i++)
            if (n % i == 0) return false;
        return true;
    }
    for (int a : candidate)
        if (!miller_rabin(n, a))
            return false;
    return true;
}

int main() {
    int n;
    cin >> n;
    cout << (is_prime(n) ? 'O' : 'X') << '\n';
    return 0;
}
```