確率的素数判定法

1　素数の判定

素数を判定するのに**試し割り法**や**エラトステネスの篩い**などを用いれば，確実に素数であると判定することができる。しかし，これらの方法で大きな素数を判定するには膨大な時間がかかってしまう。そこで，合成数（1 とその数自身以外の約数を持つ自然数）ではないことを確率的に判断する方法を用いる。このような判定方法を**確率的素数判定法**という。OpenSSL で利用されている素数判定法は，確率的素数判定法の Miller-Rabin 素数判定法が使われている。Miller-Rabin 素数判定法は，フェルマーテストの条件をより精密にした判定法である。また，他の素数判定法に比べ Miller-Rabin 法による素数判定が最も速いことから利用されている。

ちなみに，確実に素数であることを判定する決定的アルゴリズムに AKS 素数判定法 (AKS primality test, Agrawal-Kayal-Saxena primality test) がある。2002 年に発表された世界初の多項式時間で解ける素数判定法である。

2　フェルマーテスト

2.1　フェルマーの小定理

フェルマーの小定理は，p を素数としたとき，p と互いに素な整数 a に対して次の式が成り立つ。

$$a^{p-1} \equiv 1 \pmod{p} \tag{1}$$

この式を Python の条件式で表すには，まず「2 つの整数 a, b が法 n に関して合同であることを $a \equiv b \pmod{n}$ で表す」を考えてみる。$a \equiv b \pmod{n}$ は，a を n で割った剰余 $(a \bmod n)$ が b を n で割った剰余 $(b \bmod n)$ と等しいことから，Python の条件式で `a % n == b % n` と表すことができる。これを踏まえて (1) 式を Python の条件式で記述すると，`a**(n - 1) % n == 1` となる。しかし左辺にはべき乗が含まれているため，そのまま計算すると a^{n-1} がとても大きい値になると効率が悪くなる。そこで**べき剰余**を求めるのに**バイナリ法**を利用する。

2.2　バイナリ法

コンピュータ上で累乗演算を効率的に行う方法を**バイナリ法**という。累乗を単純に計算するとオーダーは $O(n)$ になってしまうが，バイナリ法を使うことで $O(\log n)$ になる。バイナリ法は，下位ビット (Right-to-left) から計算する方法と上位ビット (Left-to-right) から計算する方法がある。下位ビット (Right-to-left) から計算する方法については，前期の授業にて解説した。

2.2.1 下位ビット (Right-to-left) から計算する方法

下位ビット (Right-to-left) から計算する方法は，b^n の n に対して

$$n = a_0 2^0 + a_1 2^1 + a_2 2^2 + \cdots + a_{i-1} 2^{i-1} + a_i 2^i \quad (a_0, a_1, \cdots, a_i \text{は } 0 \text{ または } 1)$$

のように分解する。例えば $n = 37$ は，$37 = 2^0 + 2^2 + 2^5 = 1 + 4 + 32$ である。このことから b^{37} は $b^{1+4+32} = b^1 b^4 b^{32}$ として計算することができる。また分解した n は 2 進法の各桁に該当するので，これを下位ビットからみると，b^n は $b^1, b^2, b^4, \cdots, b^{2^{(i-1)}}, b^{2^i}$ の何れかの組み合わせの掛け算になり，プログラムでは `b = b * b` を繰り返すことで組み合わせの値が得られる。この組み合わせは，n の 2 進数に対して下位ビットからビットが 0 か 1 なのかを調べればよい。

$$n = 37_{(10)} = 100101_{(2)}, \quad \text{下位ビットから } 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 0 \times 2^4 + 1 \times 2^5$$

下位ビットから得られる 1 か 0 を，組み合わせの要・不要フラグとして見る

$b^1 \rightarrow$ 必要, $b^2 \rightarrow$ 不要, $b^4 \rightarrow$ 必要, $b^8 \rightarrow$ 不要, $b^{16} \rightarrow$ 不要, $b^{32} \rightarrow$ 必要

プログラムでは，求める解の変数を `p` としたとき，最初に `p = 1` とする。`b = b * b` を繰り返して，下位ビットから 1 となる必要な組み合わせのときだけ `p = p * b` を計算すれば $p = b^n$ が求まる。

2.2.2 上位ビット (Left-to-right) から計算する方法

上位ビット (Left-to-right) から計算する方法は，累乗の次のような性質を利用する。

$$b^{2i} = (b^i)^2 \tag{2}$$
$$b^{2i+1} = (b^i)^2 b \tag{3}$$

この 2 つの性質を利用して，指数を分割していく。例えば b^{37} は，(3) 式により $b^{37} = (b^{18})^2 b$ が得られ，次に (2) 式により $(b^{18})^2 b = ((b^9)^2)^2 b$ が得られる。このように最後まで続けると b^{37} は次のようになる。

$$b^{37} = (b^{18})^2 b = ((b^9)^2)^2 b = (((b^4)^2 b)^2)^2 b = ((((b^2)^2)^2 b)^2)^2 b = (((((b)^2)^2)^2 b)^2)^2 b$$

$(((((b)^2)^2)^2 b)^2)^2 b$ について b の位置に注目すると，$n = 37_{(10)} = 100101_{(2)}$ の上位ビットから見たとき，1 が入っている同じ位置に「b」（最上位ビット）または「$)^2 b$」が現れることがわかる。このことから，上位ビットから計算するときは，初期値を 1 とした求める値を繰り返し二乗して，上位ビットから 1 が現れたときに b をかければよい。

プログラムでは，求める解の変数を `p` としたとき，最初に `p = 1` とする。`p = p * p` を繰り返して，上位ビットから 1 が現れるときだけ `p = p * b` を計算すれば $p = b^n$ が求まる。

2.2.3 下位ビットと上位ビットから計算したときの違い

下位ビットから計算すると，`b = b * b` を繰り返すため `b` の値が大きくなる。しかし上位ビットから計算すると，`b` の値は固定のままで済む。このため，大きな自然数を扱う場合は上位ビットから計算する方式が良い。ただし，べき剰余を計算するのであれば優位性は失われる。

2.2.4 バイナリ法と情報漏洩

バイナリ法は，ビットの 1 の個数による演算時間や演算負荷の変動を電力消費の変動として計測することが可能であると，実験で実証されている。このため，特に秘密鍵には情報漏洩の危険性から，バイナリ法を利用

していない。この問題を解決する方法として，モンゴメリラダー (The Montgomery Powering Ladder) 法が 2002 年に考案されている。

2.3 べき剰余とバイナリ法

累乗はとても大きな値となるため，$a^e \bmod n$ のようなべき剰余を求めるには a^e が大きくなるほど効率が悪くなってしまう。そこでべき剰余は次の式を利用する。

$$ab \bmod n = ((a \bmod n)(b \bmod n)) \bmod n$$

バイナリ法で乗算を行うたびに n の剰余をとっても，べき剰余の計算結果は変わらない。また Python には，べき剰余のアルゴリズムが最初から実装されている。それが組み込み関数 `pow(x, y[, z])` である。べき剰余 $a^e \bmod n$ は，`pow(a, e, n)` のように呼び出す。

2.4 フェルマーテストの実装

フェルマーテストは，フェルマーの小定理を用いて素数判定を行う。(1) 式は，p が素数のときに成り立つが，素数ではない合成数も成り立つ場合が出てくる。そこで，自然数 a をランダムに選んで繰り返し判定を行うことで，確率的素数を判定する。

フェルマーテストのアルゴリズム

1. 素数を判定したい整数 n を入力し，試行回数 i を与える。
2. ランダムに選んだ 2 以上 n 未満の整数 a を 1 つ与える。
3. a と n が互いに素でなければ 2. に移る。
4. $a^{n-1} \equiv 1 \pmod{n}$ を満たさない場合は合成数と判定し終了する。
5. 繰り返し数に 1 を足して i 未満なら 2. に戻り，繰り返し数が i ならば素数と判定し終了する。

フェルマーテストで繰り返し数を増やすほど確率的素数は素数である可能性が高くなるが，素数ではないにも関わらず確率的素数と判定されてしまった数を擬素数という。その中でも**カーマイケル数**は，いかなる整数 a を用いてもフェルマーテストで必ず確率的素数と判定されてしまう合成数である。カーマイケル数は，$561, 1105, 1729, 2465, \cdots$ など無数に存在する。

3 Miller-Rabin 素数判定法

3.1 Miller-Rabin 法

Miller-Rabin 法で判定する素数は，2 を既知の素数として扱うことで 2 よりも大きな素数が判定の対象となるので，奇素数である。そこで，フェルマーの小定理から，奇素数（2 以外の素数）n と底 a を $1 \leq a < n$ としたとき次の式が成り立つ。

$$a^{n-1} \equiv 1 \pmod{n} \tag{4}$$

しかしフェルマーテストは，素数ではないにも関わらず必ず確率的素数としてしまうカーマイケル数が存在する。そこで判定の条件をより細かく検討してみる。

n は奇素数なので，$n-1$ は偶数であり，$(n-1)/2$ は整数である。そこで (4) 式を $(a^{(n-1)/2})^2 \equiv 1 \pmod{n}$

と変形して，整数 $a^{(n-1)/2}$ を x としたとき、次の式で表せる。

$$x^2 \equiv 1 \pmod{n}$$

さらに両辺から 1 を引くと，合同式は次のように表せる。

$$\begin{aligned} x^2 - 1 &\equiv 0 \pmod{n} \\ (x+1)(x-1) &\equiv 0 \pmod{n} \end{aligned}$$

このことから，x はどちらかの合同式に当てはまる。

$$x \equiv 1 \pmod{n} \quad \text{または} \quad x \equiv -1 \pmod{n} \tag{5}$$

これを手がかりとする。n は奇数なので, $n-1$ は偶数であり, 偶数は必ず 2 で割り切れることから $n-1 = 2^s t$ と表すことができる。ここで s は正の整数，t は奇数である。例えば $n = 13$ なら，$13 - 1 = 12 = 2^2 \times 3$ と表せる。$n - 1 = 2^s t$ としたとき，$a^{n-1} = a^{2^s t}$ について，a^t を次々と 2 乗して得られる次の列を考える。

$$a^t, a^{2t}, a^{2^2 t}, \cdots, a^{2^{s-1} t}, a^{2^s t} = a^{n-1} \tag{6}$$

それぞれの列の値を法 n として考えたとき，(4) 式から最後の項 $a^{2^s t} = a^{n-1}$ は 1 と合同である。その 1 つ前の項 $a^{2^{s-1} t}$ は (5) 式から，1 と合同か -1 と合同かのどちらかである。もし 1 と合同であれば，もう 1 つ前の項 $a^{2^{s-2} t}$ にさかのぼり，それも 1 と合同か -1 と合同かのどちらかである。このように 1 つ前の項が 1 と合同のとき，次々とさかのぼると最後の項 (a^t) まで 1 と合同であるか，あるいは途中で -1 と合同になるか，のどちらかでしかあり得ない。さかのぼる過程において，1 つ前の項に 1 でも -1 でもない剰余類が現れた場合は，その数は素数ではないと判定できる。

以上をまとめると，n が奇素数で $n - 1 = 2^s t$ とし，a が $1 \leq a < n$ であるような整数のとき，次のいずれかの条件が成り立つ。

条件 1. $a^t \equiv 1 \pmod{n}$

条件 2. 整数 $r(0 \leq r < s)$ の何れかに対して $a^{2^r t} \equiv -1 \pmod{n}$

この条件は，n が奇素数の場合に成り立つ性質を取り出したものなので，フェルマーテストより精密な判定条件になることが期待できる。

3.2 Miller-Rabin 素数判定法の実装

判定条件をプログラムにするには，自然数 a をランダムに選び，(6) から a^t の項を最初に求める。$a^t \bmod n$ が 1 のときは，条件 1 を満たすので素数候補とする。それ以外のときは，条件 2 を満たすことを確認するため a^t から $a^{2^{s-1} t}$ までを順に法 n の -1 を探し，見つかったときは素数候補とする。もしも条件 2 の探索中に -1 よりも先に 1 が得られたときは，条件 1 と矛盾するため素数ではないと判定する。また法 n の -1 が見つからないときも条件 2 を満たさないので，素数ではないと判定する。

ここまでの判定を繰り返し行い，判定がすべて素数候補のときは確率的素数とする。

Miller-Rabin 素数判定法のアルゴリズム

1. 素数を判定したい正の整数 n を入力し，試行回数 k を与える。
2. n が 1 のときは素数ではないと判定して終了する。n が 2 のときは素数と判定して終了する。n が 2 以外の偶数のときは合成数と判定して終了する。
3. $n-1=2^s t$ となる t と s を求める。$n-1$ を 2 で繰り返し割って，割り切れなくなったときの値が t である。また s は，その繰り返した回数である。（$n-1$ を 2^s で割り切れる最大の整数が s である。）
4. 繰り返し数が k 未満なら，繰り返し数に 1 を足して 5. 以降を実行する。繰り返し数が k ならば素数と判定して終了する。
5. ランダムに選んだ 2 以上 n 未満の整数 a を 1 つ与え，$f=a^t \bmod n$ を計算する。
6. f が 1 または $n-1$(法 n の -1) のときは 4. に戻る。
7. $r=0$ とする。
8. r を 1 繰り上げ，r が s ならば合成数と判定して終了する。
9. $f=a^{2^r t} \bmod n$ を計算する。
10. f が $n-1$(法 n の -1) のときは 4. に戻る。
11. f が 1 のときは合成数と判定して終了する。
12. 8. に戻る。

3.3 Miller-Rabin 素数判定法の試行回数

Miller-Rabin 素数判定法による素数候補の試行回数（繰り返し数）は，どのぐらい行えば良いのであろうか。そこで次の論文を紹介する。

> AMERICAN MATHEMATICAL SOCIETY（AMS : アメリカ数学会）
> MATHEMATICS OF COMPUTATION, VOLUME 61 (1993), PAGES 177-194
> 「Average case error estimates for the strong probable prime test」
> Ivan Damgård, Peter Landrock and Carl Pomerance.
> http://www.ams.org/journals/mcom/1993-61-203/home.html

この論文によると、k ビットの数に対して 1 回の Miller-Rabin 素数判定法が誤判定される確率は $\leq k^2 4^{2-\sqrt{k}}$ とある。現在の認証局 (CA, Certificate Authority, Certification Authority) は，2010 年頃までに 1024 ビットから 2048 ビット証明書に移行した。1024 ビットの証明書に必要な素数は 512 ビット，2048 ビットの証明書に必要な素数は 1024 ビットなので，それぞれを確率を計算すると次の通り。

- $k=512$ ビットに対して $\leq$ 約 2^{-23}
- $k=1024$ ビットに対して $\leq 2^{-40}$

OpenSSL 1.0.0（2010 年 3 月リリース）では，Miller-Rabin 素数判定法に対して C 言語のソースコード `openssl/crypto/bn/bn.h`（リスト 1）にて反復回数をマクロで定義している。マクロは前述の論文を元に，素数を誤判定する確率が 2^{-80} より小さくなるように指定されている。$k=512$ ビットであれば試行回数は 6 回，$k=1024$ ビットであれば試行回数は 3 回行う。

リスト 1　OpenSSL 1.0.0 openssl/crypto/bn/bn.h より

```
/* number of Miller-Rabin iterations for an error rate  of less than 2^-80
 * for random 'b'-bit input, b >= 100 (taken from table 4.4 in the Handbook
 * of Applied Cryptography [Menezes, van Oorschot, Vanstone; CRC Press 1996];
 * original paper: Damgaard, Landrock, Pomerance: Average case error estimates
 * for the strong probable prime test. -- Math. Comp. 61 (1993) 177-194) */
#define BN_prime_checks_for_size(b) ((b) >= 1300 ?  2 : \
                                (b) >=  850 ?  3 : \
                                (b) >=  650 ?  4 : \
                                (b) >=  550 ?  5 : \
                                (b) >=  450 ?  6 : \
                                (b) >=  400 ?  7 : \
                                (b) >=  350 ?  8 : \
                                (b) >=  300 ?  9 : \
                                (b) >=  250 ? 12 : \
                                (b) >=  200 ? 15 : \
                                (b) >=  150 ? 18 : \
                                /* b >= 100 */ 27)
```

また OpenSSL 1.1.0i（2018 年 8 月リリース）から試行回数が変更された。試行回数の元となった内容は，アメリカ国立標準技術研究所 (National Institute of Standards and Technology, NIST, 以下 NIST) によって提唱された Digital Signature Standard (DSS) である。この内容が，米国政府標準の電子文書認証方式となっている。2013 年 7 月に発行された FIPS PUB 186-4 には，「Appendix F: Calculating the Required Number of Rounds of Testing Using the Miller-Rabin Probabilistic Primality Test」というの内容があり，OpenSSL 1.1.0i はこれに添った試行回数に変更された（リスト 2）。$k = 512$ ビットであれば試行回数は 5 回（OpenSSL 1.1.0h までは 6 回），$k = 1024$ ビットであれば試行回数は 5 回（OpenSSL 1.1.0h までは 3 回）行う。

リスト 2　OpenSSL 1.1.1g include/openssl/bn.h より

```
/*
 * BN_prime_checks_for_size() returns the number of Miller-Rabin iterations
 * that will be done for checking that a random number is probably prime. The
 * error rate for accepting a composite number as prime depends on the size of
 * the prime |b|. The error rates used are for calculating an RSA key with 2 primes,
 * and so the level is what you would expect for a key of double the size of the
 * prime.
 *
 * This table is generated using the algorithm of FIPS PUB 186-4
 * Digital Signature Standard (DSS), section F.1, page 117.
 * (https://dx.doi.org/10.6028/NIST.FIPS.186-4)
 *
(中略)
 *
 * prime length | RSA key size | # MR tests | security level
 * -------------+--------------|------------+---------------
 *  (b) >= 6394 |     >= 12788 |          3 |        256 bit
 *  (b) >= 3747 |     >=  7494 |          3 |        192 bit
 *  (b) >= 1345 |     >=  2690 |          4 |        128 bit
 *  (b) >= 1080 |     >=  2160 |          5 |        128 bit
 *  (b) >=  852 |     >=  1704 |          5 |        112 bit
```

```
 *  (b) >=  476 |     >=   952 |          5 |         80 bit
 *  (b) >=  400 |     >=   800 |          6 |         80 bit
 *  (b) >=  347 |     >=   694 |          7 |         80 bit
 *  (b) >=  308 |     >=   616 |          8 |         80 bit
 *  (b) >=   55 |     >=   110 |         27 |         64 bit
 *  (b) >=    6 |     >=    12 |         34 |         64 bit
 */

# define BN_prime_checks_for_size(b) ((b) >= 3747 ?  3 : \
                                (b) >=  1345 ?  4 : \
                                (b) >=  476 ?  5 : \
                                (b) >=  400 ?  6 : \
                                (b) >=  347 ?  7 : \
                                (b) >=  308 ?  8 : \
                                (b) >=  55  ? 27 : \
                                /* b >= 6 */ 34)
```

さらに OpenSSL 3.0（2021 年 9 月リリース）からは，処理の流れと試行回数が修正された。Miller-Rabin 素数判定法の処理の流れは，NIST の FIPS 186-4「C.3 .1 Miller Rabin Probabilistic Primality Test.」で提示されているプロセスに変更された。試行回数は，エラー確率が 2^{-128} が保証されるように，$k = 2048$ ビット以下であれば 64 回，$k = 2048$ ビットより大きければ 128 回行う（リスト 3）。

リスト 3　OpenSSL 3.0.10 openssl/crypto/bn/bn_prime.c より

```
/*
 * Use a minimum of 64 rounds of Miller-Rabin, which should give a false
 * positive rate of 2^-128. If the size of the prime is larger than 2048
 * the user probably wants a higher security level than 128, so switch
 * to 128 rounds giving a false positive rate of 2^-256.
 * Returns the number of rounds.
 */
static int bn_mr_min_checks(int bits)
{
    if (bits > 2048)
        return 128;
    return 64;
}

(中略)

/*
 * Tests that |w| is probably prime
 * See FIPS 186-4 C.3.1 Miller Rabin Probabilistic Primality Test.
 *
 * Returns 0 when composite, 1 when probable prime, -1 on error.
 */
static int bn_is_prime_int(const BIGNUM *w, int checks, BN_CTX *ctx,
                           int do_trial_division, BN_GENCB *cb)
{
(以下略)
```

ちなみに OpenSSL 3.1（2023 年 3 月リリース）の処理の流れと試行回数は，OpenSSL 3.0 から変更されていない。

4　素数の生成

RSA 暗号では，1024 ビットの大きさの素数を生成する必要がある。このような大きな素数をどのように生成しているのであろうか。OpenSSL では，素数を生成するのに次のような手順を行っている。

1. 指定ビット数の乱数を生成する。
2. Miller-Rabin 法で素数か判定し，素数ではないときは 1. に戻る。

この手順に従えば，まず大きな乱数を生成する必要がある。次に Miller-Rabin 法による素数の判定は，第 3 章の通りだ。では，どのように大きな乱数を生成すれば良いだろうか。

4.1　大きな乱数を生成する

Python では簡単に大きな乱数を作ることも可能だが，一般的には大きな乱数を生成するには難しい。まず擬似乱数をどのように生成するか，ということが問題である。ここでは擬似乱数の問題まで詳しく取り上げないが，例えば 2006 年に株式会社バンダイナムコゲームス発売の Xbox360 向けゲームソフト「カルドセプトサーガ」に「次のダイス目が偶数か奇数か推測できる」という致命的バグが見つかり，店頭からソフトを回収することになった。これは，擬似乱数が奇数と偶数を交互に繰り返す問題があったと推測されている。また乱数で大きな数値を作るのに，ある程度繰り返すと元の数値に戻るような周期性があっては，生成した数値を見破られる危険性も伴う。例えば 2000 年の頃の Excel VBA では，$2^{24} = 16777216$ 以下の周期を持つといわれている。このような問題が発生しない乱数が生成できるという前提で話を進める。ちなみに Python の乱数については，モデル化やシミュレーション向きで主にメルセンヌ・ツイスタ法 (Mersenne Twister) で生成する random モジュールと，セキュリティや暗号向きの secrets モジュールで生成する方法が提供されている。

表 1　8bit の乱数

8bit	7bit	6bit	5bit	4bit	3bit	2bit	1bit
1	0 or 1	0 or 1	0 or 1	0 or 1	0 or 1	0 or 1	1

1024 ビットの大きな乱数を生成するのに，2^{1023} 以上 $2^{1024} - 1$ 以下の数値を選ばなければならない。しかも奇素数なので，最初から奇数の乱数が欲しい。そこで，2 進法を利用して乱数を生成する。例えば 8 ビットの乱数を生成してみる。まず 8 ビットの先頭ビットには，必ず 1 が入る。もしも先頭ビットに 0 が入った時点で 7 ビットの数値になってしまうからである。また奇数を生成するので，最後尾ビットには必ず 1 が入る。$2^0 = 1$ であり，2^1 以上のビットは全て偶数なので，「(偶数 + 1) = 奇数」となるからである。それ以外のビットは，表 1 のように 0 か 1 の乱数で埋めていけばよい。

同じような方法でビット数を 1024 ビットに拡張すれば，1024 ビットの乱数が生成できる。また OpenSSL も同様な方法で乱数を生成している。