10.4 散策おしまい

さて、家についた。手作業を自動化して encrypt() で暗号化、decrypt() で復号化ができるようにしよう。これまでに書いたスクリプトの寄せ集めなので当然、効率は悪い。encrypt() には sqrt.math() を使っているので import math は忘れずに。はじめに encrypt() の説明をしよう。

[py script]

```
>>> def primechk(n):
...   isP = True
...   for i in range(3, n//2, 2):
...     if n % i == 0:
...       isP = False
...   return isP
...
>>> def phifacto(n):
...   i = 3
...   while 1:
...     if (n % i != 0) and (primechk(i) == True):
...       return i
...     else:
...       i += 2
...
>>> def genprime(n):                                    #[（注）手抜きあり]
...   if n % 2 == 0:
...     n += 1
...   while 1:
...     if primechk(n) == True:
...       return n
...     else:
...       n += 2
...
>>> def encrypt(s):
...   code = ''
...   for c in s:
...     code += str(ord(c))
...   code = int(code)
...   p = genprime( int(math.sqrt(code)) )
...   q = genprime( p+2 )
...   r = phifacto( (p-1)*(q-1) )
...   print( 'n(p,q)=%d(%d,%d), r=%d' % (p*q, p, q, r) )
...   print( 'cript: %d' % (code**r % (p*q)) )
...
>>> encrypt('TeX') # 実行前に import math
n(p,q)=8444827(2903,2909), r=3
cript: 6010984
```

encrypt() は引数に文字列をとる。この文字列を暗号化するには ascii コードで数値化しなくて

はいけない。前々節でやったとおりだが、数値 code は計算に用いるので int で整数値に直している。暗号化に必要なのはふたつの素数 p, q の積 n で、しかも n = p*q > code であることが重要だ。そこで、確実にそうなるようにお手軽で “手抜き” の方法で p, q を生成している。

方法は p, q を $\sqrt{\texttt{code}}$ よりちょっとだけ大きな値にすることである。その役目は genprime() が担っている。p を $\sqrt{\texttt{code}}$ 以上の奇数にしてから、素数になるまで 2 ずつ大きな値を調べるだけだ。最初に見つけた素数が p になる。q は p より 2 大きいところから同じようにして調べる。これで積 p*q は code より少し大きな値になるのだ。

n = p*q を決めたら、次に必要な数は (p-1)*(q-1) と互いに素となる数 r だ。これは phifacto() が受け持つ。要するに、(p-1)*(q-1) を 3 から順に奇数で割って、割り切れない素数を見つけたらそれを r としている。primechk 関数の真偽は’t’, ’f’ ではなく、真偽値 True, False を使った。

ここまでで必要な数がそろったので、code**r % n を計算すれば、その数字列—実際は数値—が暗号である。出力例を見ると暗号以外の数値も表示しているね。なぜなら、復号するためには公開鍵である n、r の値が必要だからだ。本当に必要なのは p, q だけどね。

次は decrypt() についてだ。

[py script]

```
>>> def solvemod(r, m):
...   t = 1
...   while (r*t % m) != 1:
...     t += 1
...   return t
...
>>> def ord2chr(s):
...   str = ’’; i = 0
...   while s[i:] != ’’:
...     if s[i] == ’1’:
...       str += chr(int(s[i:i+3]))
...       i += 3
...     else:
...       str += chr(int(s[i:i+2]))
...       i += 2
...   return str
...
>>> def decrypt(s, p, q, r):
...   t = solvemod(r, (p-1)*(q-1))
...   print( ’t=%d’ % t )
...   print( ’plain: %s (%d)’ % (ord2chr(str(s**t % (p*q))), s**t % (p*q)) )
...
>>> decrypt(6010984, 2903, 2909, 3)
t=5626011
plain: TeX (8410188)
```

decrypt() は 4 個の引数をとる。本来なら復号には、暗号化した数字列と鍵 n、r だけでよいのであるが、単に楽をしたかったのだ。と言っても、n を受け取って素因数分解するようにコードを付け足すのは簡単である。以前作った intfacto 関数を少し書き換えるだけでよいからだ。そうすればスクリプトは decrypt(s, n, r) でよい。これは難しいことではないので、君たち自身で試す価値はあるだろう。

さて、復号に必要なのはもちろん暗号の数字列と n、r である。r は $rt \equiv 1 \pmod{\varphi(n)}$ の解 t を求める際に使う。で、$\varphi(n) = (p-1)(q-1)$ だったから、すぐにでも p, q の値が欲しかったのさ。$rt \equiv 1 \pmod{\varphi(n)}$ を解く関数 solvemod() は、ユークリッドの互除法なんて使ってない。愚直に t = 1 から順に試しているだけである。でも、現実的にこれで十分なのだ。

実際に decrypt() を実行した結果を見てほしい。ここでは余計と思える t の値も表示している。その理由は、計算に時間がかかるからだ。復号のための計算は (暗号)^t を n で割った余りなのだから、この例の t=5626011 でもかなりの時間を要する。コンピュータの性能にもよるが、t が 8 桁を超えるようなら処理を止める方がよいかもしれない。その判断のために t の値を出力したのだ。だから、これらのスクリプトで "暗号 $\leftrightarrow$ 復号 ごっこ" をするなら、暗号にする文字列はせいぜい 3 文字までがいいところだ。

そして "手抜き" で p, q を生成したため、処理時間がかかるのとは別の、重大な問題が存在している。次の例を見てもらいたい。

[py script]

```
>>> encrypt('A')
n(p,q)=143(11,13), r=7
cript: 65
>>> encrypt('B')
n(p,q)=143(11,13), r=7
cript: 66
>>> encrypt('C')
n(p,q)=143(11,13), r=7
cript: 89
```

あれ？　'A' と 'B' は暗号化されてないの？　だって、ascii コードそのままじゃない。そう、実は暗号化に失敗している。こうなってしまった原因は 'A', 'B' の ascii コードが 65, 66 で、それらより少し大きなふたつの素数を生成するスクリプトのせいで、(p,q) が (11,13) になったからである。

直接の原因は、'A' と 'B' の ascii コードにも 11 や 13 が素因数に含まれていることだ。前節の復号の説明にあたってオイラーの定理を使っているが、$x^{\varphi(n)} \equiv 1 \pmod{n}$ には「x と n は互い

に素」という条件がついていたはずだ。だから 65 や 66 については、素数 p, q に 13 や 11 を使ってはいけないのだ。したがって、もしこういうことが起こらないようにしたければ、ふたつの素数の生成を工夫するしかない。それには、genprime 関数が code も引数にとり、code % n != 0 のチェックを加えればよい。つまり、

[py script]

```
>>> def genprime(n, code):
          ⋮
...     if (primechk(n) == True) and (code % n != 0):
          ⋮
```

ということだ。

チェック抜きでも平文が 1 文字でなければ、生成される p, q が平文の素因数とかぶる可能性は低い。でも、いまは散歩から帰ったところだよね。十分な時間はあるのだから、君たち自身でこれ以外にも手を加えてみるとよいだろう。それに、割られる数と割る数が共通の素因数を持つ割り算が、限られた余りしか出さない理由を調べるのは楽しいと思うよ。以前書いた、余りが循環する様子を見るスクリプトは、よい手助けになるはずだ。ゆったりとスクリプトを書いてみようじゃないか。