9.3 リピュニット

もしかしたら 11... の並木道に見えるかも知れない横道に、少し入ってみよう。それはリピュニット（repunit = repeated unit）の道だ。リピュニットとは $111\cdots1$ のように、1 だけからなる数をいう。小学生や数字マニアに好まれそうな数である。

話題になるのは、リピュニットのうち素数は何か、である。n 桁のリピュニットを R_n と書くことにすると

$$R_2 = 11, \quad R_3 = 3 \cdot 37, \quad R_4 = 11 \cdot 101, \quad R_5 = 41 \cdot 217, \quad \ldots$$

のように、R_3 以降はなかなか素数が現れそうにない雰囲気だ。こんなときこそ **Python3** の出番である。

といっても、以前書いたスクリプトにリピュニットを与えて、それが素数がどうか調べるだけでは、いまこの道に入った意味がない。明らかに素数にならないものは除いて、可能性のあるリピュニットを調べるようにしてみよう。

まず、R_{2k} のタイプは素数ではない。つまり、偶数個の 1 からなるリピュニットは、各桁の 1 を交互に足したり引いたりすれば必ず 0 になるので 11 の倍数である。このことから、11 以外の R_{2k} は除外できる。また、R_{3k} のタイプも素数ではない。各桁の 1 を全部足せば 3 の倍数になる。よって、これも除外できる。うむ、何かの倍数であることを判定する方法を知ったので、9... の散歩道ついでに立ち寄った甲斐があるというものだ。

他の可能性についてはどうだろう。R_{4k} や R_{9k} のたぐいは R_{2k} と R_{3k} に含まれるので、除外できないのは R_n の n が素数であったり、$n \neq 3k$ のような奇数ということになる。多少なりとも計算量を抑えるために、とりあえず $n = 2k$ と $n = 3k$ だけは除外して考えておこう。

ところで、n 桁のリピュニットを式で表すとどうなるのだろう。

$$111\cdots1 = 10^n + 10^{n-1} + 10^{n-2} + \cdots + 1$$

である。しかし、これをそのままスクリプトにするのは気が引ける。こういうときは、数列の公式に頼るのがよい。リピュニットの式は、**初項**が 1 で**公比**が 10 の**等比数列**の和である。その和 S_n を公式より求めると

$$S_n = \frac{10^n - 1}{10 - 1} = \frac{10^n - 1}{9}$$

である。実際、$n = 10$ で考えると、$10^{10} - 1 = 9999999999$ であるから（やっぱり 9... の散歩道！）、間違いなく $S_{10} = 1111111111$ となって、見事に 10 個の 1 が並ぶ。

2

とりあえず、この程度の理解で **Python3** に任せよう。

[py script]

```
>>> def pchk(n):
...   i = 3
...   while i <= math.sqrt(n):
...     if n % i == 0:
...       return 0
...     i += 2
...   return 1
...
>>> def repu(f):
...   if f == 2:
...     print('素リピュニット')
...   elif (f % 2 == 0) or (f % 3 == 0):
...     print('合成リピュニット')
...   else:
...     n = (10**f - 1) / 9                    #[（注）場合によっては書き換え]
...     if pchk(n) == 1:
...       print('素リピュニット')
...     else:
...       print('合成リピュニット')
...
>>> repu(2) # 実行前に import math
素リピュニット
>>> repu(5)
合成リピュニット
```

ここでは、素数になるリピュニットを “素リピュニット”、そうでないリピュニットを “合成リピュニット” と呼ぶことにした。素数の判定は以前の pchk 関数そのままである。だから repu 関数は、引数が調べる価値があるかどうかを判定するだけのものである。

ところで repu 関数で、f 桁の 11...1 を作るときに

[py script]

```
...     n = (10**f - 1) / 9                    #[（注）場合によっては書き換え]
```

としているのだが、**Python** のバージョンのせいか何かで n が整数値にならないことがある。その場合、同じ 11...1 を作るなら

[py script]

```
...     n = 1                                  #[（注）場合によっては書き換え]
...     while (math.log(n) / math.log(10) + 1) < f:
...       n = 10 * n + 1
```

という方法に書き換えるとよいだろう。バージョンや仕様の細かな違いには、臨機応変に対処しよう。

さて、引数を調べる価値の有無を判断するために使った or は、論理演算子である。A or B は A か B のどちらかが真のとき、真となる。また、and 演算子と同様に、A が真なら B をいちいち評価しない。

リピュニットは完全数などより、おそらくはるかに少ない。そもそも、リピュニットは 10^n ごとに 1 個ずつしかないにもかかわらず、10^{2k} と 10^{3k} のリピュニットは最初から除外されてしまう。これだけでも、かなりの損失？である。$n = 2$ から順に試してみると分かるように、いつでも表示されるのは合成リピュニットである。そして、不安をかき立てるのは、引数 n がある程度の大きさになったとき、突然、計算に時間がかかるようになることだ。

君たちのコンピュータの性能にもよるが、もしかしたらスクリプトを実行して素リピュニットを目にすることができないかもしれない。理由は、$R_2 = 11$ の次の素リピュニットは R_{19} だからだ。もちろん 19 桁の数である。さすがにこの大きさになると、非力なコンピュータでは簡単に計算結果を出してくれない。そもそも、この散歩で書いているスクリプト自体が非力なのだけれどね。

すると、どうしたらいいんだろう。とる道はふたつある。ひとつの道は、もっと効率の良いスクリプトを書くことだ。そのためには、だいぶ経験を積まなくてはならない。もうひとつの道は、このスクリプトのまま気長に待つことだ。しかし、この道に進むのは勇気がいるだろう。R_n の計算が我慢できる範囲であっても、R_{n+1} は単純に考えても 10 倍かかると思わなくてはいけない。どっちの道をとるかは君たち次第だ。ちなみに、R_{19} の次は R_{23} である。

さあ、だいぶあちこち散歩したね。そろそろ家に帰る頃だろう。repu 関数と pchk 関数の改良は家に帰ってから、ぜひ君たちが取り組んでほしい。とくに、pchk 関数がこのままでは R_{19} が素リピュニットであることを確認するのに、多少は忍耐を要するからだ。R_{23} ではなおさらだ。もし、R_{19} が手強いなら、逆に合成リピュニットを素因数分解してみるのもよい。R_n を素因数分解するのはなかなかの快感だと思う。