Fibonacci and Pisano

Introduction

The Fibonacci sequence is the integer sequence that is given by the following:

$F_0 = 0, F_1 = 1,$ and $F_n = F_{n-1} + F_{n-2}$ for any $n > 1$.

Our code generates the fibonacci numbers by looping through n and storing the previous two values of the sequence.

```
def F(n):
    a, b = 0, 1
    for i in srange(n):
        a, b = b, a+b
    return a
```

```
for i in srange(12):
    print(fibonacci(i))
```

```
0
1
1
2
3
5
8
13
21
34
55
89
```

```
for i in srange(12):
    print(F(i))
```

```
0
1
1
2
3
5
8
13
21
34
55
89
```

In order to make our output more readable and efficient, we only need to keep track of the last 20 digits.

```
def F(n):
    a, b = 0, 1
    for i in srange(n):
        a, b = b % 10^20, a+b % 10^20
    return a
```

```
fibonacci(10000)
```

```
33644764876431783266621612005107543310302148460680063906564769974680\
081442166662368155595513633734025582065332680836159373734790483865268\
```

Binet's Formula

We have provided an implementation that defines each fibonacci number *implicitly*; that is, F_n is defined using F_{n-1} and F_{n-2}.

The fibonacci numbers can also be defined *explicity*, using Binet's formula:

$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$, for $n \geq 0$.

Looking at this formula, we should expect the fibonacci sequence to grow exponentially. Indeed, it does appear to do so. This means that as n grows, it will become harder and harder to compute the value of F_n.

Still, we can easily approach numbers on the magnitude of 10^7 using our basic implementation.

```
%time
F(10^7)
```

```
86998673686380546875
CPU time: 8.43 s,  Wall time: 8.43 s
```

Matrix Solution

In addition to the *implicit* and *explicit* solutions that were provided above, the Fibonacci numbers can also be defined using matrix multiplication.

Let $A_1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. In order to find the nth fibonacci number, we simply multiply A_1 by itself n times:

$$A_n = (A_1)^n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \cdots = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$, for $n > 0$

```
A = matrix(2,2,[1,1,1,0])
for i in srange(1,12):
    print i, A^i
```

```
1 [1 1]
[1 0]
2 [2 1]
[1 1]
3 [3 2]
[2 1]
4 [5 3]
[3 2]
5 [8 5]
[5 3]
6 [13  8]
[ 8  5]
7 [21 13]
[13  8]
8 [34 21]
[21 13]
9 [55 34]
[34 21]
10 [89 55]
[55 34]
11 [144  89]
[ 89  55]
```

The benefit of the matrix multiplication approach is that we can quickly calculate A_n using an iterative squaring technique:

$$(A_1)^2 = A_2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix},$$

$$(A_2)^2 = A_4 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix},$$

$$(A_4)^2 = A_8 = \begin{bmatrix} 34 & 21 \\ 21 & 13 \end{bmatrix},$$

$$(A_8)^2 = A_{16} = \begin{bmatrix} 1597 & 987 \\ 987 & 610 \end{bmatrix},$$

$\ldots,$ and so on.

Observe that $A_n = (A_1)^{b_1} \times (A_1)^{2b_2} \times (A_1)^{4b_3} \times (A_1)^{8b_4} \times \cdots = (A_1)^{b_1} \times (A_2)^{b_2} \times (A_4)^{b_3} \times (A_8)^{b_4} \times \ldots,$ where b_m represents the value of the mth digit starting from the right in the binary representation of n.

For example, if $n = 14,$ then $n_{01} = 1110,$ and thus

$$A_{14} = (A_1)^0 \times (A_2)^1 \times (A_4)^1 \times (A_8)^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \times \begin{bmatrix} 34 & 21 \\ 21 & 13 \end{bmatrix} = \begin{bmatrix} 610 & 377 \\ 377 & 233 \end{bmatrix}$$

```
A1 = matrix(2,2,[1,1,1,0])
print A1^0 * A1^2 * A1^4 * A1^8
```

```
[610 377]
[377 233]
```

```
A = matrix(2,2,[1,1,1,0])
print A^14
```

```
[610 377]
[377 233]
```

Using this approach, the runtime of our implementation can be reduced down to logarithmic complexity, since we'll only need to compute approximately $\log_2(n)$ different sets of operations.

```
def F(n):

    # fibonacci sequence begins at 0
    if n==0:
        return 0

    # we use iterated squaring of the [1 1\ 1 0] matrix to quickly
    # find the nth fibonacci number modulo 10^20
    F = matrix(2,2,[1,0,0,1])
    A = matrix(2,2,[1,1,1,0])
    for k in srange(1,n.nbits()+1): # for each '1' in binary, we
        if n & (1 << k-1):          # multiply that index's square
```

```
            F=F*A % 10^20           # to the final matrix F
        A=A^2 % 10^20
    return F[0,1]
```

```
%time
F(10^7)
```

```
    86998673686380546875
    CPU time: 0.00 s,  Wall time: 0.00 s
```

In fact, we can now take inputs that are much, much larger than 10^7.

Because our runtime is logarithmic, we might expect to be able to compute output for values ranging up to $n = 2^{10^7}$.

The inefficiencies of our implementation, however, limit us. Even still, taking inputs for values between $n = 2^{10^5}$ and $n = 2^{10^6}$ is doable.

```
log(7^245738,2).n(digits=8)
```

```
    689873.78
```

```
%time
F(7^245738)
```

```
    54442120219743276449
    CPU time: 39.10 s,  Wall time: 39.11 s
```

We want to investigate even more ways to optimize our code, so that computing values up to $n = 2^{10^7}$ (and even larger!) becomes manageable.

Body

The Pisano period of some number m, written $\pi(m)$, is the period of the Fibonacci sequence when taken under modulo m, and it tells us after how many numbers the sequence will start to repeat itself.

For example, $\pi(4) = 6$, since the sequence $F_n \pmod 4$ repeats itself every 6 numbers.

```
for i in srange(14):
    print(fibonacci(i) % 3)
```

```
    0
    1
    1
    2
    0
    2
    2
    1
    0
    1
    1
    2
```

```
0
2
```

The Pisano periods can be computed using the following algorithm:

1. Begin at the values $a = 0$ and $b = 1$. These are the values that begin every sequence of $F_n \pmod{m}$. (Remember that $F_0 = 0$ and $F_1 = 1$; therefore $F_0 \pmod{m} = 0$ and $F_1 \pmod{m} = 1$ for all values of m.)
2. Iterate up to m^2, checking whether the next two numbers in the $F_n \pmod{m}$ sequence are 0 and 1, respectively. (The reason we iterate up to m^2 is because this is the maximum value for any Pisano period. Consider, for any sequence X_n, the function $\omega(n, m) = X_n \pmod{m}$. We know that the output of ω must range between 0 and m (by definition). Therefore, after m unique cycles of output, (each ranging through a different ordering of the values from 0 to m), ω is garaunteed to generate at least one of the cycles that came before it, as each of the collections of residues for m would have had to have been used. Because F_n is recursive, we know that at this point, the cycles will continue to all repeat, making ω periodic.)

Turning our algorithm into code, we now have a function that finds the Pisano period of any given m.

```
def pisano(m):
    if m==1: return 1
    a, b = 0, 1
    for i in srange(0, m ^ 2):
        a, b = b, (a + b) % m
        if a == 0 and b == 1:
            return i + 1
```

```
for i in srange(1,25):
    print(pisano(i))
```

```
1
3
8
6
20
24
16
12
24
60
10
24
28
48
40
24
36
24
18
60
16
30
```

```
48
24
```

The Pisano period of prime numbers

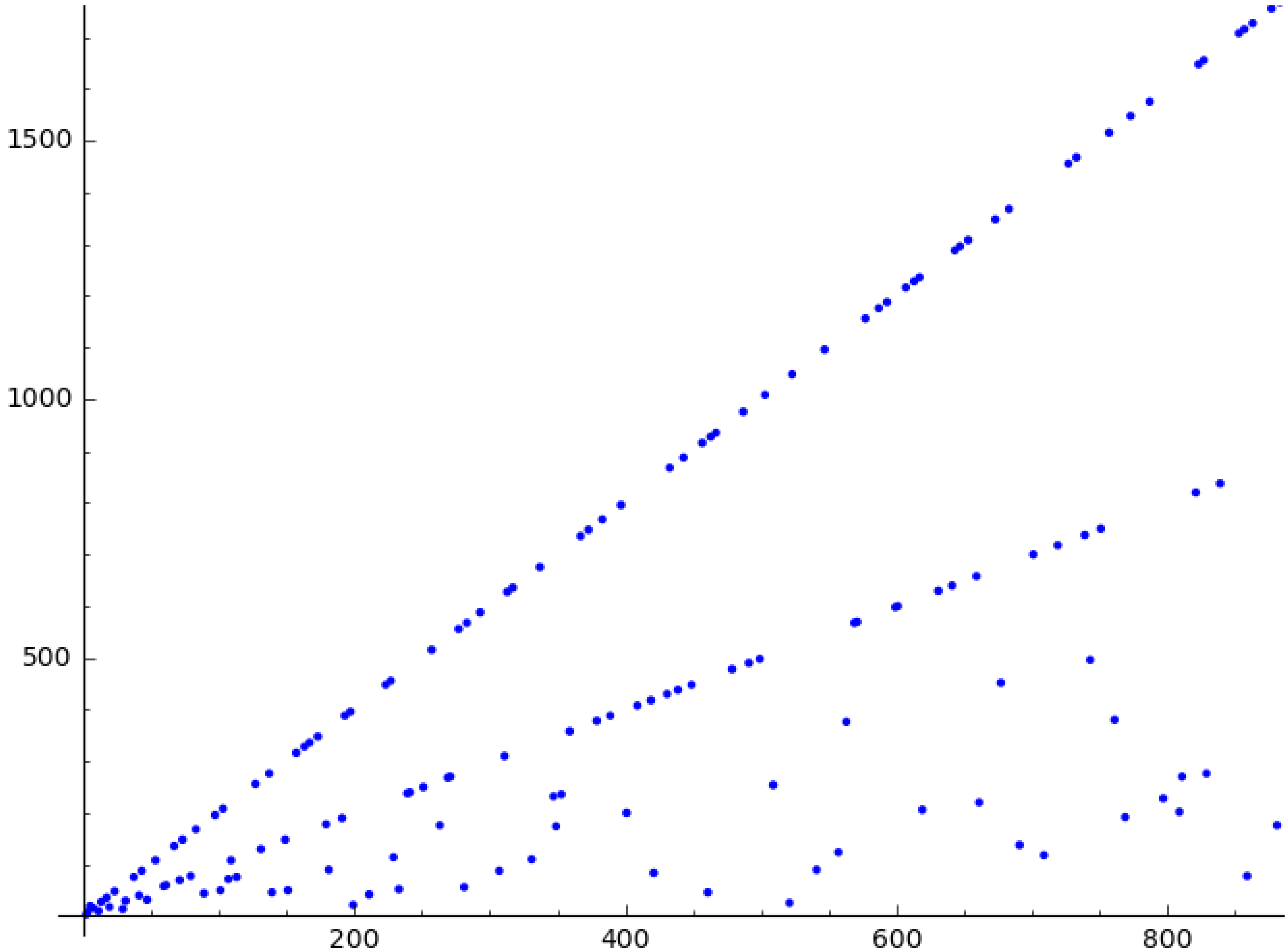
The Pisano periods can reflect certain properties and behaviors of the Fibonacci sequence when taken modulo m.

For example, let's see how F_n behaves when taken under the modulo of a prime number.

```
for i in srange(100):
    if is_prime(i): print i, '\t', pisano(i)
```

```
2       3
3       8
5       20
7       16
11      10
13      28
17      36
19      18
23      48
29      14
31      30
37      76
41      40
43      88
47      32
53      108
59      58
61      60
67      136
71      70
73      148
79      78
83      168
89      44
97      196
```

```
pisano_list=[]
for i in srange(2,1000):
    if is_prime(i): pisano_list.append([i,pisano(i)])
list_plot(pisano_list)
```



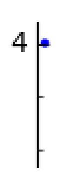
It appears that the periods of primes fall into at least two distinct categories. Looking at the ratios of these periods to their prime will help us determine what behaviors our data reflects.

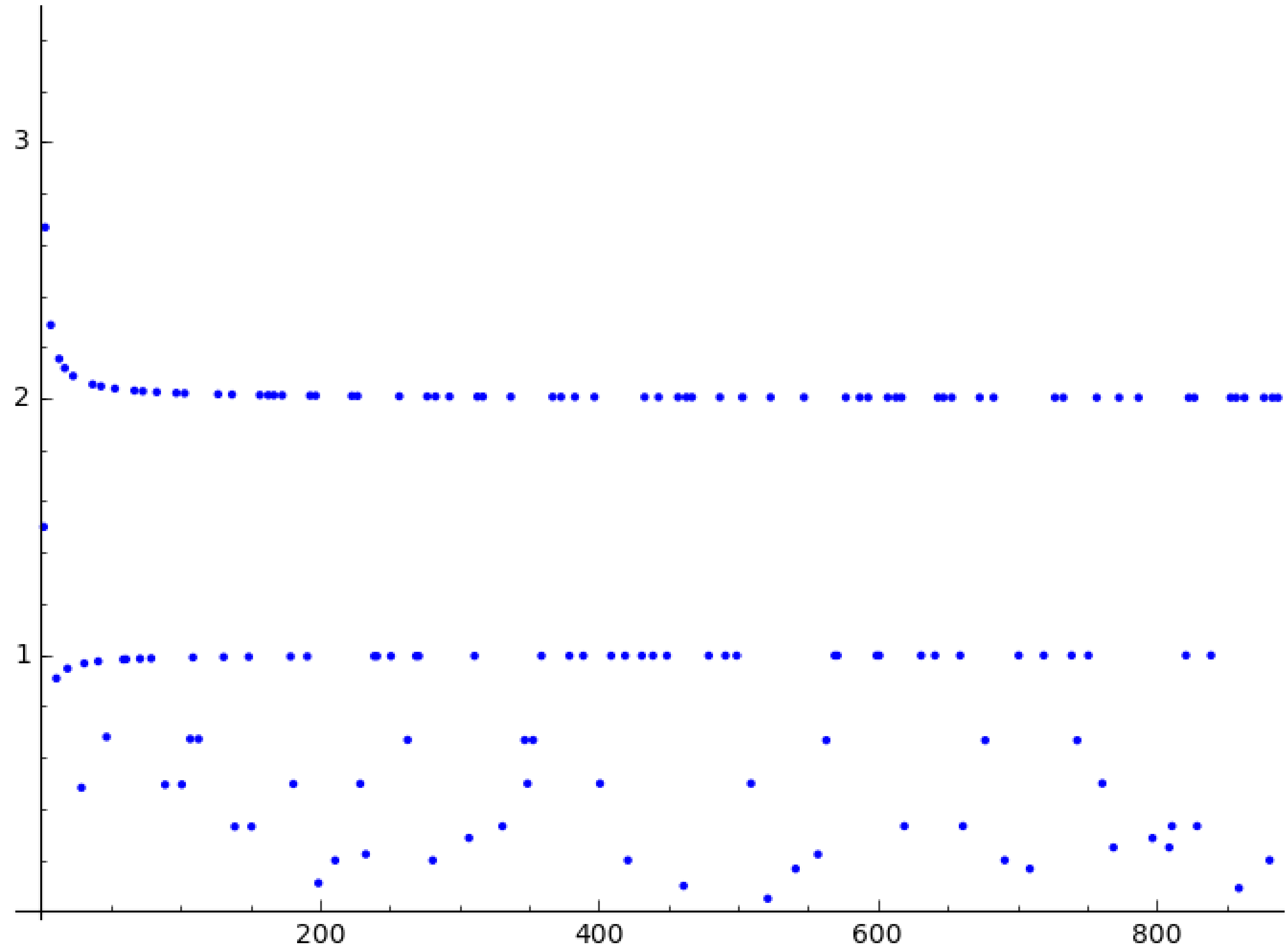
```
for i in srange(200):
    if is_prime(i): print i, '\t', float(pisano(i)) / i
```

```
2       1.5
3       2.66666666667
5       4.0
7       2.28571428571
11      0.909090909091
13      2.15384615385
17      2.11764705882
19      0.947368421053
23      2.08695652174
29      0.48275862069
31      0.967741935484
37      2.05405405405
41      0.975609756098
43      2.04651162791
47      0.68085106383
53      2.03773584906
59      0.983050847458
```

61	0.983606557377
67	2.02985074627
71	0.985915492958
73	2.02739726027
79	0.987341772152
83	2.02409638554
89	0.494382022472
97	2.0206185567
101	0.49504950495
103	2.01941747573
107	0.672897196262
109	0.990825688073
113	0.672566371681
127	2.0157480315
131	0.992366412214
137	2.01459854015
139	0.330935251799
149	0.993288590604
151	0.331125827815
157	2.0127388535
163	2.01226993865
167	2.0119760479
173	2.01156069364
179	0.994413407821
181	0.497237569061
191	0.994764397906
193	2.0103626943
197	2.01015228426
199	0.110552763819

```
pisano_ratio_list=[]
for i in srange(2,1000):
    if is_prime(i): pisano_ratio_list.append([i,pisano(i)/i])
list_plot(pisano_ratio_list)
```





Some of the primes have ratios that approach 2, some that approach 1, and others that all seem to be scattered beneath.

Furthermore, each the primes with a ratio close to 2 appear to end in either 3 or 7; the primes close to 1 appear to end in either 1 or 9; and the primes that are neither seem to end in any of the four values.

```
for i in srange(200):
    if is_prime(i) and (mod(i,10)==3 or mod(i,10)==7): print i, '\t',
pisano(i)
```

```
3       8
7       16
13      28
17      36
23      48
37      76
43      88
47      32
53      108
67      136
73      148
83      168
97      196
103     208
```

```
107     72
113     76
127     256
137     276
157     316
163     328
167     336
173     348
193     388
197     396
```

The general formula for primes ending in 3 or 7 looks to be $\pi(p) = 2(p+1)$. There are some exceptions seen here, which we will return to.

```
for i in srange(200):
    if is_prime(i) and (mod(i,10)==1 or mod(i,10)==9): print i, '\t',
pisano(i)
```

```
11      10
19      18
29      14
31      30
41      40
59      58
61      60
71      70
79      78
89      44
101     50
109     108
131     130
139     46
149     148
151     50
179     178
181     90
191     190
199     22
```

The general formula for primes ending in 1 or 9 looks to be $\pi(p) = p - 1$. There are some exceptions here, as well.

Let's explore what those exceptions seem to reflect.

```
for i in srange(200):
    output = pisano(i)
    if is_prime(i) and (mod(i,10)==1 or mod(i,10)==9):
        if output != i-1:
            print "expected: ", i-1, ", actual: ", output
    elif is_prime(i) and (mod(i,10)==1 or mod(i,10)==9):
        if output != i-1:
            print "expected: ", 2*(i+1), ", actual: ", output
```

```
expected:  28 , actual:  14
expected:  88 , actual:  44
expected:  100 , actual:  50
expected:  138 , actual:  46
expected:  150 , actual:  50
expected:  180 , actual:  90
expected:  198 , actual:  22
```

Within our exceptions, it appears that the actual Pisano period of that prime is some divisor of what our general formulas expect.

Let us formulate a conjecture. Let $g(p) = p - 1$ when p is $\pm 1 \pmod{10}$, and $g(p) = 2(p+1)$ when p is $\pm 3 \pmod{10}$.

Then we conjecture that $\pi(p)$ always divides $g(p)$ for any given prime p.

```
def g(p):
    if mod(p,10)==1 or mod(p,10)==9:
        return p-1
    elif mod(p,10)==3 or mod(p,10)==7:
        return 2*(p+1)

def test_conjecture():
    for i in srange(1000):
        if is_prime(i):
            if g(i) % pisano(i) != 0:
                return "wrong at: " + str(i)
    return "finished with no errors"
```

```
test_conjecture()
```

```
'finished with no errors'
```

Using this conjecture, we can now formulate a much faster implementation which assumes the two general formulas and the rules for exception.

Given some output $\pi(p) = a$, we want to be able to continually check to see if this is indeed the Pisano period of p. To do this, we will first check if $F_

```
def F(n,m):
    # fibonacci sequence begins at 0
    if n==0:
        return 0

    # we use iterated squaring of the [1 1\ 1 0] matrix to quickly
    # find the nth fibonacci number modulo 10^20
    F = matrix(2,2,[1,0,0,1])
    A = matrix(2,2,[1,1,1,0])
    for k in srange(1,n.nbits()+1): # for each '1' in binary, we
        if n & (1 << k-1):          # multiply that index's square
            F=F*A % m               # to the final matrix F
        A=A^2 % m
```

```
    return F[0,1]

def faster_pisano(p):
    if p.is_prime():
        if mod(p,10)==1 or mod(p,10)==9:
            g=p-1
        elif mod(p,10)==3 or mod(p,10)==7:
            g=2*(p+1)
        d_list=g.divisors()
        for d in d_list:
            # We'll check if F_d, F_(d+1) are 0, 1 mod p
            if F(d,p)==0 and F(d+1,p)==1:
                return(d)
        return('Ooops')
```

```
%time
pisano(next_prime(4000))
```

```
2000
CPU time: 3.58 s,  Wall time: 3.58 s
```

```
%time
faster_pisano(next_prime(4000))
```

```
2000
CPU time: 0.02 s,  Wall time: 0.02 s
```

It is important that we have a way to continually check whether our conjecture is actually true.

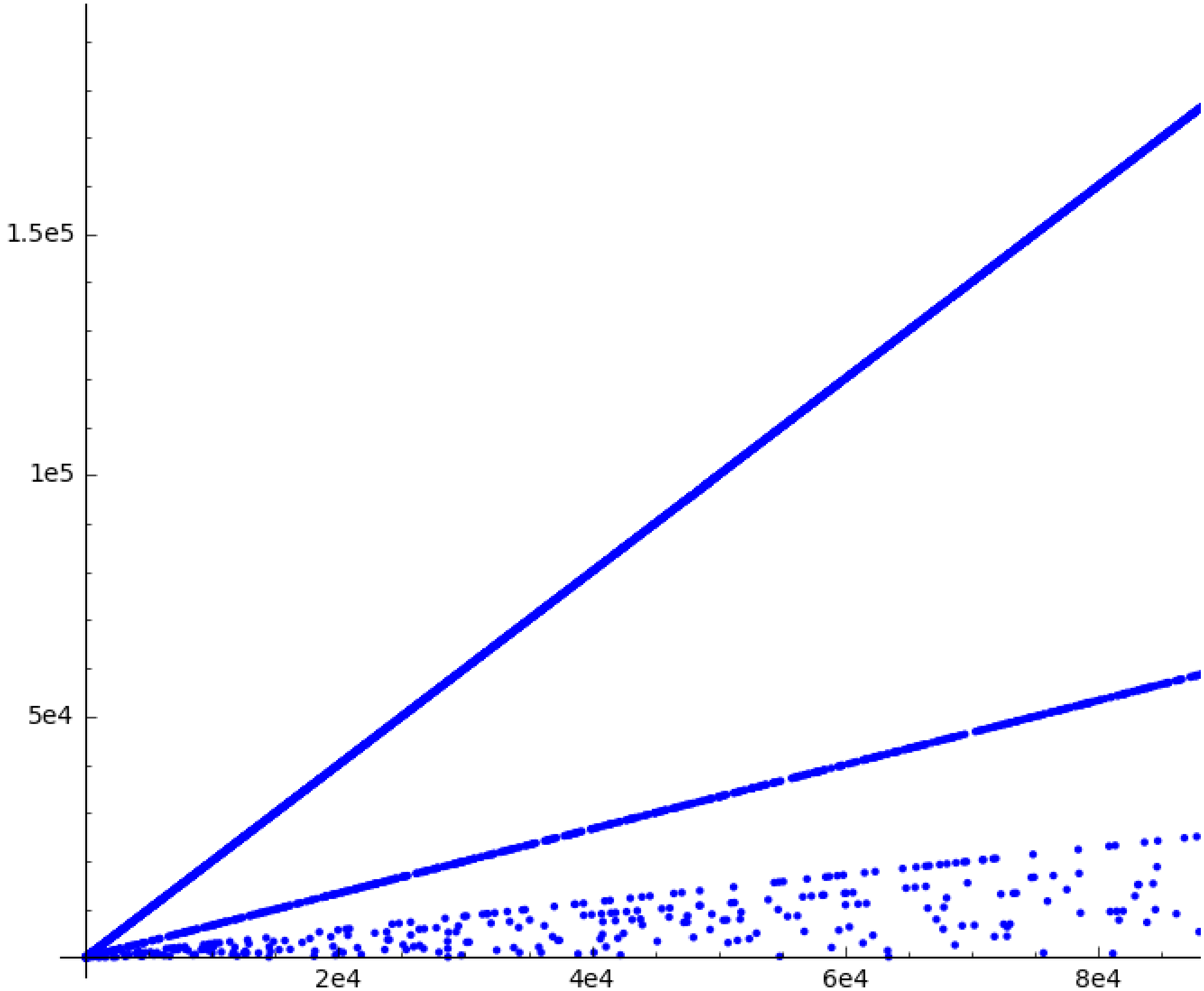
Within the faster implementation, we've added a return for cases where the sequence does not repeat itself after $\pi(m)$ times. That way, we are informed whenever a prime does not follow the two formulas (and exceptions) we've developed.

Let us now take another look at the data, this time using inputs up to 10^5.

```
pisano_list_3=[]
for i in srange(10000):
    if is_prime(10*i+3):
        pisano_list_3.append([10*i+3, faster_pisano(10*i+3)])
    if is_prime(10*i+7):
        pisano_list_3.append([10*i+7, faster_pisano(10*i+7)])
```

```
list_plot(pisano_list_3)
```

2e5



As you can see, none of our inputs broke the conjecture.

The new data continues to support our formulas as the trends for the plot continued as predicted when we increased the input space. Note that the majority of these outputs lie within the two general formulas; that is, the majority of outputs are defined by $g(p)$ alone.

The exceptions, cases where the actual Pisano period was a *nontrivial factor* of $g(p)$, seem to become more sparse as our input space rises.

Prime Powers

Let's now investigate the Pisano periods of prime powers.

```
for i in srange(15):
    if is_prime(i):
        print i, ": ", pisano(i)
        print i^2, ": ", pisano(i^2)
        print i^3, ": ", pisano(i^3), "\n"
```

```
2 :  3
```

```
4 :  6
8 :  12

3 :  8
9 :  24
27 :  72

5 :  20
25 :  100
125 :  500

7 :  16
49 :  112
343 :  784

11 :  10
121 :  110
1331 :  1210

13 :  28
169 :  364
2197 :  4732
```

The periods of these primes seem to each divide the periods their prime powers.

Let's explore this more with $p = 5$.

```
for i in srange(1,6):
    print 5^i, ": ", pisano(5^i)
```

```
5 :  20
25 :  100
125 :  500
625 :  2500
3125 :  12500
```

Each iteration of the prime powers of 5 have a Pisano period that is 5 times more than the period before it.

Let's look at another example, $p = 7$.

```
for i in srange(1,6):
    print 7^i, ": ", pisano(7^i)
```

```
7 :  16
49 :  112
343 :  784
2401 :  5488
16807 :  38416
```

The general formula appears to be $\pi(p^k) = p^{k-1}\pi(p)$, for any power k of a prime p.

Let's run some more data to see how our conjecture holds up.

```
x_1 = next_prime(10^8)
x_2 = x_1^2
x_100 = x_1^100

period_1 = faster_pisano(x_1)
period_2 = x_1 * period_1
period_100 = x_1^99 * period_1
```

We have set x_1 as the next prime that occurs after 10^8. We should expect the first 2 numbers within $F_n \pmod{x_1}$ to repeat after $\pi(x_1)$ iterations.

```
for i in srange(2):
    print F(i,x_1), F(period_1+i,x_1)
```

```
0 0
1 1
```

Within the faster Pisano implementation, it suffices to see, after $\pi(x_1)$ numbers, if $F_n \pmod{x_1}$ begins repeats itself. This is because our input x_1 is prime, and we have seen that all primes up to 10^5 strictly adhere to the conjecture made in the previous section, wherein we are very confident that F_n does not repeat itself anytime sooner than predicted. We shall later make another conjecture about the nature of *composite* numbers.

```
# the output should be wrong, as we're still using the period of p_1
for i in srange(2):
    print F(i,x_2), F(period_1+i,x_2)
```

```
0 9730484381133859
1 9865242890566955
```

```
# the output should match as now our period is p * period_1
for i in srange(2):
    print F(i,x_2), F(period_2+i,x_2)
```

```
0 0
1 1
```

Finally, we have x_{100} set as $(x_1)^{100}$. Using our formula, we should expect the first 2 numbers within $F_n \pmod{(x_1)^{100}}$ to repeat after $\pi((x_1)^{100}) = (x_1)^{99}\pi(x_1)$ iterations.

```
# the output should be wrong, as we're still using the period of p_1
for i in srange(2):
    print F(i,x_100), F(period_1+i,x_100)
```

```
0
9375535169503107524057939362054477002223967509164123362686253306722\
6069574545396317076367413031458822265083347173979860451011041916335\
2001291471126669179328375542161333001419594327847879392885875935781\
7704712162439305369686866406122429075018250914837368738752308248672\
7826940385484983477934877653150512523417617795960498143626736212012\
6096923572261035412425945911988005662201420380921611514684164855406\
9934213334191776822036924324708877510604768993374732595088389504840\
4982249091839328563447024500552224553167484888595265323874070368498\
8170692100435605291000878567872640876734280992720115871713650612713\
```

```
4870936273593739187668290138571361930835207207422568469193374466977\
5987605808466398218720881521545602736466629303082906789806264804477 0\
7079354239522794222041882291664476438813730876556330
1
875485897710339494763050323239290840462005131211857348534384952862 84\
794098774025367460437577817627156418756265465586873921864796510657 33\
832801903345641263760618603994692613066542949818711982634758403196 56\
907446100831813942254528011505013395063680150469043080875386865839 47\
768923824590083001059557597201766744066387089952096062144118381435 54\
370793773187106241996128488594494494758567744864829346898764418292 59\
268128017607783316790797388028094081178471799816482824225767973439 30\
655730580547095640443068951281001788314717182166243483342784883932 87\
019585876382288887198277948430577365768688718612841082397231231362 00\
920419444030559337439717957918905226703723162012341478935736471413 89\
489659205075009851295975002057761224413444334600065332621817049599 13\
5220910462845954156627280496521090994370748085725782
```

```
# the output should still be wrong, as we're using the period of p_2 and not
p_100
for i in srange(2):
    print F(i,x_100), F(period_2+i,x_100)
```

```
0
837683989543395340364277090678802636272440807197468380782076487449 80\
996454288939018659226733380350077696111815932949462940904612028518 68\
204840953445913713153351492719440031957747935238337266783247725205 98\
780160269021964152943084393914450618628215627602460711507195340695 90\
132119345612774764179144064618952748372662671669862683486331440719 92\
572648208845227380116629466461222866656263329308608938341109634793 24\
139635738758779372995390328234255391826725219447159787231416737930 30\
669648374567940817210287019210223865988262453865319600669269888264 11\
186725512806098567766400677483811058308756227648821597860162602880 72\
758707358034686082657733410301332789500937846462083575642164087242 07\
490850734837874865729378076150853823712183087406728601926900917568 82\
3933324313641677692833657335917649557116148537368235
1
892671118751897974168100440086856791413528873553286958111908599377 22\
385061763858879129523852571313977436786117767474040464989790237809 21\
360908769325401282039044951669426629513991986294239118673039527493 37\
236664095678769939165699428395781143251456477075142608355780104339 17\
322461453292293277899690064441357346661762724409417892431129278604 48\
824707643640751187996034115884949898270296671613755690762463404706 90\
190506123773113067718786496758190959383536925178975816168854838851 10\
109064302108553434110552532276937581397122383283261168644932856130 29\
312890659834604200706645758559111032037429605848063762284111779509 36\
494252165104073502011707319608860241512685205123089015363891945077 74\
069455129300799480695346862740613448110282057197588114517550036552 01\
8780072145891382878237335133753158989099487365417180
```

```
# the output should match as now our period is p^14 * period_1
for i in srange(2):
    print F(i,x_100), F(period_100+i,x_100)
```

```
0 0
```

```
1 1
```

Let us take the moment to update our faster implementation for the Pisano periods. First, we will check if p is a prime. If so, we will use the implementation that we have already created. Next, we will check if p is a prime power. Because the Pisano periods of prime powers are given in terms of its prime, we simply need to recurse once to find the Pisano period of that prime. Using this, we can very quickly calculate very large powers.

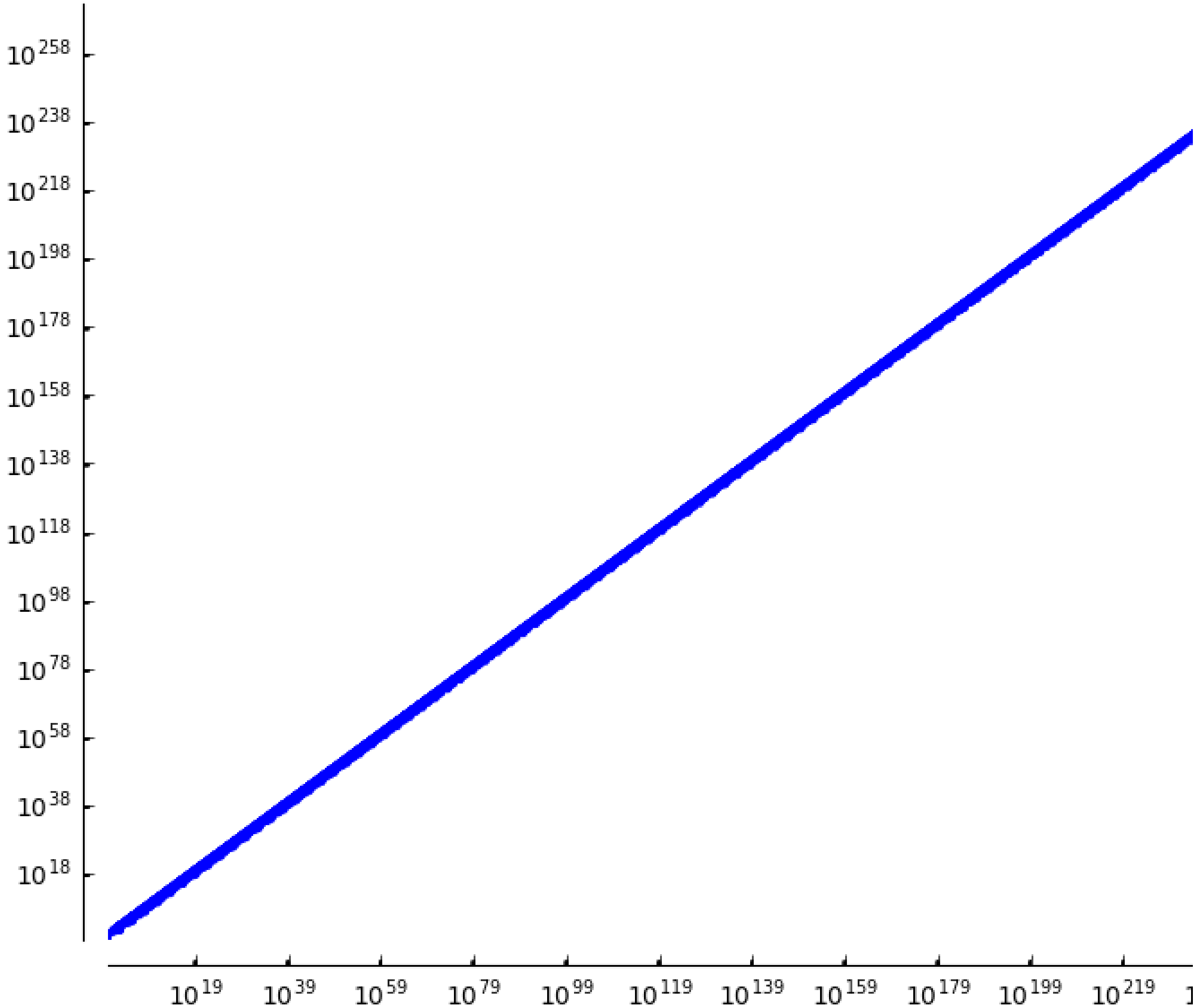
```
def F(n,m):
    # fibonacci sequence begins at 0
    if n==0:
        return 0

    # we use iterated squaring of the [1 1\ 1 0] matrix to quickly
    # find the nth fibonacci number modulo 10^20
    F = matrix(2,2,[1,0,0,1])
    A = matrix(2,2,[1,1,1,0])
    for k in srange(1,n.nbits()+1): # for each '1' in binary, we
        if n & (1 << k-1):          # multiply that index's square
            F=F*A % m               # to the final matrix F
        A=A^2 % m
    return F[0,1]

def faster_pisano(p):
    if p.is_prime():
        if mod(p,10)==1 or mod(p,10)==9:
            g=p-1
        elif mod(p,10)==3 or mod(p,10)==7:
            g=2*(p+1)
        d_list=g.divisors()
        for d in d_list:
            # We'll check if F_d, F_(d+1) are 0, 1 mod p
            if F(d,p)==0 and F(d+1,p)==1:
                return(d)
        return('Ooops')
    elif p.is_prime_power():
        M = factor(m)
        d = M[0][0]^(M[0][1]-1) * pisano(M[0][0])
        # We'll check if F_d, F_(d+1) are 0, 1 mod p
        if F(d,p)==0 and F(d+1,p)==1:
            return(d)
        return('Ooops')
```

```
pisano_list_powers=[]
p = 10
for i in srange(100):
    p = next_prime(p)
    for j in srange(100):
        pisano_list_powers.append([p^j, p^(j-1) * faster_pisano(p)])
```

```
list_plot_loglog(pisano_list_powers)
```



Because the Pisano period of prime powers grow exponentially with the base of its prime, we have calculated periods on the magnitude of $10^2 50$. This is certainly an improvement over our naive implementation.

Our conjecture is supported by the fact that we have not encountered any 'Oops' while running our code. This means that, for each prime power, our given period correctly cycled as expected in the $F_n \pmod{m}$ sequence.

Composite Numbers

Are there any behaviors that are evident between the Pisano periods of the composite numbers that are not prime powers?

Let's find out. Take, for example, the number 105.

```
pisano(105)
```

```
80
```

```
factor(105)
```

```
3 * 5 * 7
```

The factors of 105 are $3, 5,$ and 7. Let's examine if there is any correlation between their Pisano periods and $\pi(105)$.

```
print pisano(3), pisano(5), pisano(7)
```

```
8 20 16
```

It looks like $\pi(3), \pi(5),$ and $\pi(7)$ all divide $\pi(105)$.

In fact, $\pi(105)$ is the least common multiple of $\pi(3), \pi(5),$ and $\pi(7)$.

```
lcm([pisano(3), pisano(5), pisano(7)])
```

```
80
```

Let's look at another composite number.

```
pisano(50)
```

```
300
```

```
lcm([pisano(x[0]^x[1]) for x in factor(50)])
```

```
300
```

The same behavior remains.

Finally, let's consider how composite numbers behave with factors that are prime powers.

```
pisano(2^6 * 3^4)
```

```
864
```

```
lcm([pisano(x[0]^x[1]) for x in factor(2^6 * 3^4)])
```

```
864
```

We have enough to form a conjecture. We conjecture that, for any composite number n, $\pi(n)$ is equal to the least common multiple of $\pi(a_1), \pi(a_2), \pi(a_3)\ldots$, where a_i are the prime power factors of n.

Updating our implementation one more time, we have the following code.

```
def faster_pisano(m):

    if m==1:
        return 1

    elif is_prime(m):
        if m==2:
            return 3
        elif m==5:
            return 20
        elif mod(m,10)==1 or mod(m,10)==9:
            d_list=(m-1).divisors()
            for d in d_list:
                # We'll check if F_d, F_(d+1) are 0, 1 mod m
                if F(d,m)==0 and F(d+1,m)==1:
```

```
                    return(d)
        else:
            d_list=(2*(m+1)).divisors()
            for d in d_list:
                # We'll check if F_d, F_(d+1) are 0, 1 mod m
                if F(d,m)==0 and F(d+1,m)==1:
                    return(d)

    # for a prime power in the form p^k, the pisano period of that
    # prime power is p^(k-1) * pisano(p)
    elif is_prime_power(m):
        M = factor(m)
        return M[0][0]^(M[0][1]-1) * pisano(M[0][0])

    # for numbers that are neither prime or a prime power, we find
    # the pisano period by taking the least common multiple of the
    # periods of its factors
    else:
        total = 1
        for f in factor(m):
            total = lcm(total,pisano(f[0]^f[1]))
        if F(total,m)==0 and F(total+1,m)==1:
                return total
        return('Ooops')
```

Just like in the previous versions, our function will print an error if the period it predicts does not actually cycle correctly.

Using this implementation, we can compute the periods of Pisano numbers very efficiently.

```
# checks to see if the two pisano implementations are equal
def check_pisanos():
    for i in srange(1,1000):
        if not pisano(i) == faster_pisano(i):
            return i
    return "finished without errors"
```

```
check_pisanos()
```

```
'finished without errors'
```

```
%time
pisano(10000)
```

```
15000
CPU time: 20.81 s,  Wall time: 20.82 s
```

```
%time
faster_pisano(10000)
```

```
15000
CPU time: 0.08 s,  Wall time: 0.08 s
```

```
%time
faster_pisano(7^1000)
```

```
28645866056359275842458396168748210426628230287565398204879885886193\
61363011156947027402035528034760301682649589750721752818948330680297\
66312664773190524753897177397135225609107712820411045877978784033609\
36298438422287454462551548166147485838031069064783105985027218838118\
54119044333537987109483938540015627867772495752154019487774565521245\
94710527918971263029698851179731539507150852522985337031053426747455\
62597100737696025041093254580921163037648421611542909825288780157702\
20163251290224781697355445385656981426730369144262635476688638178615\
55203741384405826597342814449854513550690459637571074389484199653968\
59695555640995520684073872569414939434751111379012785342348620461134\
63158360563590873973064333870588634048026432334364469569443307338300\
57510428849861319000379874680461211886369474340167047282121283134423\
544752649305465617671498514288
CPU time: 0.00 s,  Wall time: 0.00 s
```

Conclusion

We have gone through three conjectures that each formulate an efficient solution to finding the Pisano periods of certain numbers.

$F_n \pmod{m}$ is used to output the last 20 digits of F_n when $m = 10^{20}$.

```
def F(n,m):

    # fibonacci sequence begins at 0
    if n==0:
        return 0

    # we use iterated squaring of the [1 1\ 1 0] matrix to quickly
    # find the nth fibonacci number modulo m
    F = matrix(2,2,[1,0,0,1])
    A = matrix(2,2,[1,1,1,0])
    for k in srange(1,n.nbits()+1): # for each '1' in binary, we
        if n & (1 << k-1):          # multiply that index's square
            F=F*A % m               # to the final matrix F
        A=A^2 % m
    return F[0,1]
```

Our approach to finding the Pisano period of m goes as follows.

1. If m is a prime number, then $\pi(m)$ is a divisor of $g(m)$, where $g(m) = m - 1$ when m is $\pm 1 \pmod{10}$ and $g(m) = 2(m+1)$ when m is $\pm 3 \pmod{10}$.
2. If m is a prime power of the form p^k, then $\pi(m) = p^{k-1}\pi(p)$.
3. If m is a composite number that is not a prime power, then $\pi(m)$ is equal to the least common multiple of its factors.

```
def faster_pisano(m):

    if m==1:
        return 1

    elif is_prime(m):
        if m==2:
            return 3
        elif m==5:
            return 20
        elif mod(m,10)==1 or mod(m,10)==9:
            d_list=(m-1).divisors()
            for d in d_list:
                # We'll check if F_d, F_(d+1) are 0, 1 mod m
                if F(d,m)==0 and F(d+1,m)==1:
                    return(d)
        else:
            d_list=(2*(m+1)).divisors()
            for d in d_list:
                # We'll check if F_d, F_(d+1) are 0, 1 mod m
                if F(d,m)==0 and F(d+1,m)==1:
                    return(d)

    # for a prime power in the form p^k, the pisano period of that
    # prime power is p^(k-1) * faster_pisano(p)
    elif is_prime_power(m):
        M = factor(m)
        return M[0][0]^(M[0][1]-1) * faster_pisano(M[0][0])

    # for numbers that are neither prime or a prime power, we find
    # the pisano period by taking the least common multiple of the
    # periods of its factors
    else:
        total = 1
        for f in factor(m):
            total = lcm(total,faster_pisano(f[0]^f[1]))
        return total
```

```
def fast_F(n,m):
    n = n % faster_pisano(m)
    return F(n,m)
```

Using this approach, we gain impressive control over the input space for $F_n \pmod{10^{20}}$.

```
%time
# naive approach
F(7^245738,10^20)
```

    54442120219743276449

```
CPU time: 37.67 s,  Wall time: 37.68 s
```

```
%time
# our approach
fast_F(7^245738,10^20)
```

```
54442120219743276449
CPU time: 0.29 s,  Wall time: 0.29 s
```

For example, we can easily compute the first 20 digits of F_n for values of n up to 7^{CUID}.

```
%time
fast_F(7^24573857,10^20)
```

```
45594537426400012813
CPU time: 0.71 s,  Wall time: 0.74 s
```

We can even compute larger numbers. For example, 7 multiplied by itself a billion times.

```
%time
fast_F(7^(10^9),10^20)
```

```
37274596000000000001
CPU time: 32.35 s,  Wall time: 32.36 s
```