P4M+Notebook+5

December 4, 2018

1 Part 5: Algorithmic thinking and primality testing

In this notebook, we go deeper into the nature of *algorithms* -- not only computing things with Python, but reliably computing them efficiently in time and space. We study two algorithms in detail: Floyd's cycle-finding algorithm and Pingala's exponentiation algorithm. We apply the latter to primality-testing, quickly determining whether a very large (hundreds or thousands of digits) number is prime.

We analyze the **expected performance** of these algorithms using Python's `timeit` function for timing and `randint` function to randomize input parameters. We also see how the performance depends on the number of **bits** of the input parameters. In this way, we gently introduce some practical and theoretical issues in computer science.

1.1 Table of Contents

1.2 Calculations in modular arithmetic

1.2.1 The mod (%) operator

For basic modular arithmetic, one can use Python's "mod operator" `%` to obtain remainders. There is a conceptual difference between the "mod" of computer programming and the "mod" of number theorists. In computer programming, "mod" is typically the operator which outputs the remainder after division. So for a computer scientist, "23 mod 5 = 3", because 3 is the remainder after dividing 23 by 5.

Number theorists (starting with Gauss) take a radical conceptual shift. A number theorist would write $23 \equiv 3 \mod 5$, to say that 23 is **congruent** to 3 modulo 5. In this sense "mod 5" (standing for "modulo 5") is a prepositional phrase, describing the "modular world" in which 23 is the same as ("congruent to") 3.

To connect these perspectives, we would say that the computer scientist's statement "23 mod 5 = 3" gives the **natural representative** 3 for the number 23 in the mathematician's "ring of integers mod 5". (Here "ring" is a term from abstract algebra.)

```
In [2]: 23 % 5  # What is the remainder after dividing 23 by 5?  What is the natural represent
```

```
Out[2]: 3
```

The miracle that makes modular arithmetic work is that the end-result of a computation "mod m" is not changed if one works "mod m" along the way. At least this is true if the computation only involves **addition, subtraction, and multiplication**.

```
In [3]: ((17 + 38) * (105 - 193)) % 13  # Do a bunch of stuff, then take the representative mo
```

```
Out[3]: 9
```

```
In [4]: (((17%13) + (38%13)) * ((105%13) - (193%13)) ) % 13 # Working modulo 13 along the way.
```

```
Out[4]: 9
```

It might seem tedious to carry out this "reduction mod m" at every step along the way. But the advantage is that you never have to work with numbers much bigger than the modulus (m) if you can reduce modulo m at each step.

For example, consider the following computation.

```
In [5]: 3**999 # A very large integer.
```

```
Out[5]: 4406902731602688789634850865840481219884740109173827225549734560756095324489016331802
```

```
In [6]: (3**999) % 1000  # What are the last 3 digits of 3 raised to the 999 power?
```

```
Out[6]: 667
```

To compute the last three digits of 3^{999}, Python works with some very large integers along the way. But what if we could reduce modulo 1000 at every step? Then, as Python multiplies terms, it will never have to multiply numbers bigger than 1000. Here is a brute-force implementation.

```
In [7]: P = 1  # The "running product" starts at 1.
        for i in range(999):  # We repeat the following line 999 times, as i traverses the lis
            P = (P * 3)%1000  # We reduce modulo 1000 along the way!
        print(P)
```

```
667
```

In this computation, Python never has to work with long integers. Computations with long integers are time-consuming, and unnecessary if you only care about the result of a computation modulo a small number m.

1.2.2 Performance analysis

The above loop works quickly, but it is far from optimal. Let's carry out some **performance analysis** by writing two `powermod` functions.

```
In [8]: from numpy.random import randint
```

```
In [92]: from random import randint
```

```
In [9]: def powermod_1(base, exponent, modulus):  # The naive approach.
            return (base**exponent) % modulus

In [10]: def powermod_2(base, exponent, modulus):
             P = 1 # Start the running product at 1.
             e = 0
             for i in range(exponent):
                 P = (P * base) % modulus
             return P
```

Now let's compare the performance of these two functions. It's also good to double check the code in `powermod_2` and run it to check the results. The reason is that loops like this are classic sources of Off by One Errors. One has to unravel the loop carefully to be completely certain, and testing is a necessity to avoid bugs!

We can compare the performance of the two functions with identical input parameters below.

```
In [11]: %timeit powermod_1(3,999,1000)

2.56 µs ś 38.8 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)


In [12]: %timeit powermod_2(3,999,1000)

104 µs ś 9.46 µs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
```

The `powermod_1` function is faster (perhaps by a factor of 8-10) than the `powermod_2` function we designed. At least, this is the case for inputs in the 2-3 digit range that we sampled. But why? We reduced the complexity of the calculation by using the mod operation `%` throughout. Here are a few issues one might suspect.

1. The mod operation itself takes a bit of time. Maybe that time added up in `powermod_2`?
2. The Python power operation `**` is highly optimized already, and outperforms our while loop.
3. We used more multiplications than necessary.

It turns out that the mod operation is extremely fast... as in *nanoseconds* (billionths of a second) fast.

```
In [13]: %timeit 1238712 % 1237

17.7 ns ś 2.33 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

So the speed difference is probably not due to the number of mod operations. But the other issues are relevant. The Python developers have worked hard to make Python run fast -- built-in operations like `**` will almost certainly be faster than any function that you write with a loop in Python. The developers have written programs in the **C programming language** (typically) to implement operations like `**` (see the CPython implementation, if you wish); their programs

have been **compiled** into **machine code** -- the basic sets of instructions that your computer understands (and that are not meant for people to understand). When you call a built-in operation like `**`, Python just tells your computer to run the developers' optimized and **precompiled** machine code... this is very fast! When you run your own loop, there is a lot of Python-related overhead slowing things down.

Still, it is unfortunate to use long integers if you ask Python to compute `(3**999) % 1000`. The good news is that such modular exponents are so frequently used that the Python developers have a built-in operation: the `pow` function.

The `pow` function has two versions. The simplest version `pow(b,e)` raises `b` to the `e` power. It is the same as computing `b ** e`. But it also has a modular version! The command `pow(b,e,m)` raises `b` to the `e` modulo `m`, efficiently reducing modulo `m` along the way.

```
In [14]: pow(3,999)  # A long number.

Out[14]: 4406902731602688789634850865840481219884740109173827225549734560756095324489016331802

In [15]: pow(3,999,1000) # The result, modulo 1000.

Out[15]: 667

In [16]: pow(3,999) % 1000 # The old way

Out[16]: 667

In [17]: %timeit pow(3,999,1000)

1.23 µs ś 75 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)


In [18]: %timeit pow(3,999) % 1000

2.62 µs ś 213 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

The `pow(b,e,m)` command should give a significant speedup, as compared to the `pow(b,e)` command. Remember that ns stands for nanoseconds (billionths of a second) and µs stands for microseconds (millionths of a second).

Exponentiation runs so quickly because not only is Python reducing modulo `m` along the way, it is performing a surprisingly small number of multiplications. In our loop approach, we computed 3^{999} by multiplying repeatedly. There were 999 multiplications! But consider this carefully -- did we need to perform so many multiplications? Can you compute 3^{999} with far fewer multiplications? What if you can place results in memory along the way?

In the next section, we study a very efficient **algorithm** for computing such exponents. The goal in designing a good algorithm is to create something which runs **quickly**, minimizes the need for **memory**, and runs reliably for all the necessary input values. Often there are trade-offs between speed and memory usage, but our exponentiation algorithm will be excellent in both respects. The ideas go back to Pingala, an Indian mathematician of the 2nd or 3rd century BCE, who developed his ideas to enumerate possible poetic meters (arrangements of long and short syllables into verses of a given length).

You may wonder why it is necessary to learn the algorithm at all, if Python has an optimized algorithm built into its `pow` command. First, it is interesting! But also, we will need to understand the algorithm in finer detail to implement the Miller-Rabin test: a way of quickly testing whether very large numbers are prime.

1.2.3 Exercises

1. Use the `timeit` and `randint` functions to investigate how the speed of the command `pow(a,e,m)` depends on how many digits the numbers `a`, `e`, and `m` have. Note that `randint(10**(d-1), 10**d - 1)` will produce a random integer with `d` digits. If you hold two of these variables fixed, consider how the time changes as the third variable is changed. Which of the three variables has the biggest effect on speed?

```
In [19]: %timeit pow(randint(10**(5-1), 10**5 - 1), randint(10**(5-1), 10**5 - 1), randint(10*
```

```
6.61 ţs ś 249 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

```
In [20]: %timeit pow(randint(10**(5-1), 10**5 - 1), randint(10**(5-1), 10**5 - 1), randint(10*
```

```
8.42 ţs ś 577 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

```
In [21]: %timeit pow(randint(10**(5-1), 10**5 - 1), randint(10**(15-1), 10**15 - 1), randint(1
```

```
9.35 ţs ś 794 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

```
In [22]: %timeit pow(randint(10**(15-1), 10**15 - 1), randint(10**(5-1), 10**5 - 1), randint(1
```

```
6.6 ţs ś 411 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

Changing the exponent has the greatest impact on the speed of the pow() function.

1.3 Modular dynamics

Now we look at dynamical systems, cycle-finding, and Fermat's Little Theorem. When we refer to "dynamics" or "dynamical systems," we are talking about something with *initial conditions* and an *evolution process* by which the state changes. The "state" of the system is typically described by one or more numbers, and the initial conditions describe the starting-state. The evolution process describes how the state changes at every "tick" of the clock.

For example, the Collatz conjecture is about a dynamical system where... 1. The state, at any time, is given by a single integer. 2. We explore various integers as initial conditions. 3. The state changes according to the following rule: even numbers are divided by two; odd numbers are tripled, then incremented by one.

Here is the evolution function for the Collatz conjecture.

```
In [23]: def evolve_Collatz(n):
             if n%2 == 0:
                 return n//2 # Integer division is a good idea!
             else:
                 return 3*n + 1
```

To study the dynamics of the Collatz process, we could use a simple loop.

```
In [24]: state = 7  # The initial condition
         time = 0
         while time < 20:
             print("At time {}, the state is {}.".format(time, state))
             state = evolve_Collatz(state)
             time += 1
```

```
At time 0, the state is 7.
At time 1, the state is 22.
At time 2, the state is 11.
At time 3, the state is 34.
At time 4, the state is 17.
At time 5, the state is 52.
At time 6, the state is 26.
At time 7, the state is 13.
At time 8, the state is 40.
At time 9, the state is 20.
At time 10, the state is 10.
At time 11, the state is 5.
At time 12, the state is 16.
At time 13, the state is 8.
At time 14, the state is 4.
At time 15, the state is 2.
At time 16, the state is 1.
At time 17, the state is 4.
At time 18, the state is 2.
At time 19, the state is 1.
```

To study dynamical systems more flexibly, we can create a more adaptive function.

```
In [25]: def study(init_state, evolve, time_max):
             state = init_state  # The initial condition
             time = 0
             while time < time_max:
                 print("At time {}, the state is {}.".format(time, state))
                 state = evolve(state)
                 time += 1
```

Before studying this function, we take it for a spin.

```
In [26]: study(13,evolve_Collatz, 10)
```

```
At time 0, the state is 13.
At time 1, the state is 40.
At time 2, the state is 20.
At time 3, the state is 10.
At time 4, the state is 5.
At time 5, the state is 16.
```

```
At time 6, the state is 8.
At time 7, the state is 4.
At time 8, the state is 2.
At time 9, the state is 1.
```

We have seen just about everything used in the function `study`. But one parameter might have caught your eye: `evolve` is a parameter -- an input to the function `study`. Not only numbers, lists, and arrays can be inputs to functions. Functions can be inputs to functions! Here `evolve` is a function, which is studied by the function `study`. If we want to study a different dynamical system, we can define a new function to study.

```
In [27]: def evolve_seven(n):
             return (14*n) % 100

In [28]: study(11, evolve_seven, 20) # Start at 11.  Evolve for 20 units of time.
```

```
At time 0, the state is 11.
At time 1, the state is 54.
At time 2, the state is 56.
At time 3, the state is 84.
At time 4, the state is 76.
At time 5, the state is 64.
At time 6, the state is 96.
At time 7, the state is 44.
At time 8, the state is 16.
At time 9, the state is 24.
At time 10, the state is 36.
At time 11, the state is 4.
At time 12, the state is 56.
At time 13, the state is 84.
At time 14, the state is 76.
At time 15, the state is 64.
At time 16, the state is 96.
At time 17, the state is 44.
At time 18, the state is 16.
At time 19, the state is 24.
```

If you know a bit of HTML, you can use it to output your data in a prettier format. Here is an example, using the HTML and display functions.

```
In [29]: from IPython.display import HTML, display # Load the necessary packages.

In [30]: def study(init_state, evolve, time_max):
             state = init_state  # The initial condition
             time = 0
             state_list = [None] * time_max # Initialize a list of Nones.  Faster than appendi
             while time < time_max:
```

```
            state_list[time] = state
            state = evolve(state)
            time += 1
        table = [['Time:'] + list(range(time_max)), ['State:'] + state_list] # Put the da
        display(HTML(
        '<table><tr>{}</tr></table>'.format(
            '</tr><tr>'.join(
                '<td>{}</td>'.format('</td><td>'.join(str(_) for _ in row)) for row in tab
            )
        ))
```

```
In [31]: study(11,evolve_seven, 30) # Now it will look pretty!
```

```
<IPython.core.display.HTML object>
```

You might notice that the dynamical system enters a cycle. The state at time 2 is 56, and 10 "ticks" later, the state at time 12 is 56. Since the evolution process does not depend on time, just on state, you'll see another repetition 10 ticks later: the state at time 22 is 56 also. In this way, we find that the dynamics *enter a cycle* of length 10, starting at time 2. (Note that the states at times 0,1 do not repeat!)

It is often useful to find cycles, and this raises interesting problems in computer science. How might we teach the computer to detect cycles, and how can we do this efficiently in time and space? The brute-force method is to save the past-states in a list, and then check each state against the previous states for a repetition. An implementation is below, after we introduce a few more list-tricks.

```
In [32]: 3 in [1,2,3,4,5] # Python has an "in" operator, to detect membership in a list.
```

```
Out[32]: True
```

```
In [33]: 6 in [1,2,3,4,5] # 6 is not "in" the list.
```

```
Out[33]: False
```

```
In [34]: [1,2,3,4,5].index(3) # Python can also find the *index* of an item in a list.
```

```
Out[34]: 2
```

```
In [35]: [1,2,3,4,5].index(6) # A ValueError exception is thrown if the item is not found.
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-35-8476a00e2334> in <module>()
----> 1 [1,2,3,4,5].index(6) # A ValueError exception is thrown if the item is not found.


ValueError: 6 is not in list
```

```
In [ ]: def find_cycle(init_state, evolve, time_max):
            state = init_state  # The initial condition
            state_list = [None] * time_max # Initialize a list of Nones.  Faster than appendin
            time = 0
            cycle_found = False # We have not found a cycle yet!
            while (time < time_max) and (cycle_found == False):
                if state in state_list:
                    cycle_found = True
                    index_found = state_list.index(state) # Where did we find it?
                    time_found = time
                state_list[time] = state
                state = evolve(state)
                time += 1
            table = [['Time:'] + list(range(time)), ['State:'] + state_list[:time]] # Put the
            display(HTML(
            '<table><tr>{}</tr></table>'.format(
                '</tr><tr>'.join(
                    '<td>{}</td>'.format('</td><td>'.join(str(_) for _ in row)) for row in tabl
                )
            ))
            if cycle_found:
                return index_found, time_found

In [ ]: def find_cycle_no_print(init_state, evolve, time_max):
            state = init_state  # The initial condition
            state_list = [None] * time_max # Initialize a list of Nones.  Faster than appendin
            time = 0
            cycle_found = False # We have not found a cycle yet!
            while (time < time_max) and (cycle_found == False):
                if state in state_list:
                    cycle_found = True
                    index_found = state_list.index(state) # Where did we find it?
                    time_found = time
                state_list[time] = state
                state = evolve(state)
                time += 1
            if cycle_found:
                return index_found, time_found

In [ ]: find_cycle_no_print(11,evolve_seven,30)

In [ ]: find_cycle(11,evolve_seven, 30)

In [ ]: find_cycle(-1000, evolve_Collatz,50)
```

This method of cycle detection is reliable -- it will find the moment when a dynamical system enters a cycle, and the first time the cycle repeats. But there are two drawbacks to this "brute force" method. The first drawback is memory -- one must remember all of the previous states in order to search for repetition. This is not a problem in the examples above, but one can imagine

a situation where a cycle is not encountered until after a million repetitions or more. This could cause memory problems!

The second drawback is speed. The list method `LIST.index(n)` searches through a list to find the first occurrence of the item `n`. The amount of time needed is proportional to the length of the list, or rather, the length of the list up to the point where `n` is found (or the end of the list if it's not found). At time 1, one will need to search through 1 item; at time 2, one will need to search through 2 items; at time 3, one will need to search through 3 items, etc.. So to go through 1000 steps of time, one will end up searching (testing equality) $1 + 2 + 3 + \cdots + 1000$ times. This can get slow!

So here we introduce a completely different approach to cycle detection, known (perhaps mistakenly) as Floyd's Algorithm. The idea is to send two creatures, the tortoise and the hare, through the dynamical system at different speeds, and compare their values to each other at each step of time. We implement this in a verbose form below.

```
In [ ]: def Floyd(init_state, evolve, time_max = 1000000):
            tortoise = init_state
            hare = init_state
            # The tortoise and hare start in the same state.
            time = 0
            cycle_found = False
            # Stage 1:  The tortoise and hare go through the dynamical system.
            # The tortoise goes one step per unit of time.
            # The hare goes two steps per unit of time.
            # Repeat until they equal each other (or until time hits time_max)
            while (time < time_max) and (cycle_found == False):
                time = time + 1
                tortoise = evolve(tortoise) # The tortoise goes one step.
                hare = evolve(evolve(hare)) # The hare goes two steps!
                if tortoise == hare:
                    cycle_found = True
            if time == time_max:
                print('Timeout before cycle was found :(')
                return None
            # End of Stage 1.
            # Effectively, the state at "time" equals the state at "2*time" (hare-time)
            # So the cycle length is a divisor of "time",
            # and the first cycle occurs before "time."

            # Stage 2:  Find the *beginning* of the cycle.
            time = 0
            tortoise = init_state # The tortoise starts over.
            # The hare is still the previous value... "time" steps ahead of the tortoise.
            while tortoise != hare:
                time = time + 1
                tortoise = evolve(tortoise)
                hare = evolve(hare)
            # End of Stage 2.
            # The tortoise is at the beginning of the first cycle.
```

```
            # Move the hare back to the location of the tortoise.
            # Set the hare moving forward until it finds the end of the first cycle.
            hare = evolve(tortoise) # One step ahead of the tortoise
            cycle_length = 1
            while tortoise != hare:
                cycle_length = cycle_length + 1
                hare = evolve(hare)
            # Now the tortoise is at the beginning of the first cycle,
            # and the hare is at the end of the first cycle.
            return time, cycle_length, tortoise # Functions can return multiple outputs, as a
```

```
In [37]: def Floyd_no_print(init_state, evolve, time_max = 1000000):
             tortoise = init_state
             hare = init_state
             # The tortoise and hare start in the same state.
             time = 0
             cycle_found = False
             # Stage 1:  The tortoise and hare go through the dynamical system.
             # The tortoise goes one step per unit of time.
             # The hare goes two steps per unit of time.
             # Repeat until they equal each other (or until time hits time_max)
             while (time < time_max) and (cycle_found == False):
                 time = time + 1
                 tortoise = evolve(tortoise) # The tortoise goes one step.
                 hare = evolve(evolve(hare)) # The hare goes two steps!
                 if tortoise == hare:
                     cycle_found = True
             if time == time_max:
                 raise Exception("exceeded max steps")
                 return None
             # End of Stage 1.
             # Effectively, the state at "time" equals the state at "2*time" (hare-time)
             # So the cycle length is a divisor of "time",
             # and the first cycle occurs before "time."

             # Stage 2:  Find the *beginning* of the cycle.
             time = 0
             tortoise = init_state # The tortoise starts over.
             # The hare is still the previous value... "time" steps ahead of the tortoise.
             while tortoise != hare:
                 time = time + 1
                 tortoise = evolve(tortoise)
                 hare = evolve(hare)
             # End of Stage 2.
             # The tortoise is at the beginning of the first cycle.
             # Move the hare back to the location of the tortoise.
             # Set the hare moving forward until it finds the end of the first cycle.
             hare = evolve(tortoise) # One step ahead of the tortoise
```

```
    cycle_length = 1
    while tortoise != hare:
        cycle_length = cycle_length + 1
        hare = evolve(hare)
    # Now the tortoise is at the beginning of the first cycle,
    # and the hare is at the end of the first cycle.
    return time, cycle_length, tortoise # Functions can return multiple outputs, as a
```

Now let's try out Floyd's algorithm. To understand the result, compare it to the study below.

```
In [ ]: Floyd(-1000, evolve_Collatz)
```

```
In [ ]: study(-1000, evolve_Collatz, 50)
```

```
In [ ]: find_cycle(-1000, evolve_Collatz,50)
```

1.4 Exercises

1. Use %timeit to compare the speed of Floyd and find_cycle. But be *very* careful! Before you use %timeit, remove all print commands, so that your computer doesn't try to print a million things to your browser tab! Back up your work first. For your experiments, it's good to have a dynamical system with long cycle-lengths. Try variations of the following:

```
def evolve_fun(n):     return (n*n*n + 17)%1000000
```

2. Suppose that each time you call the `evolve` function, you have to pay a dollar. How much will Floyd's algorithm cost, if the first cycle begins at time t and has length ℓ?

3. Consider the following dynamical system: Let p be a prime number, and let a be an integer between 1 and $p-1$. Begin at an integer k. Evolve by sending an integer z to $(a \cdot z)\%p$. How are the cycle lengths related to $p-1$? Try a few different primes p and integers a to look for patterns.

```
In [ ]: #1
        def evolve_fun(n):
            return (n*n*n + 17)%100000
```

```
In [ ]: %timeit find_cycle_no_print(-1000000, evolve_Collatz, 100)
```

```
In [39]: %timeit Floyd_no_print(-1000000, evolve_Collatz, 100)
```

```
60.5 ţs ś 1.24 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
```

2 2

It takes 2t iterations of evolve to find the find the cycle, then the tortoise starts over so another 2t iterations to find the beginning of the cycle, move hare to location of tortoise then evolve one step so that's one more evolve call, then ℓ iterations from the hare to find the end of the cycle. So overall 4t+ℓ+1

```
In [65]: def evolve_prime_1(z):
             return ((2*z)%17)

         def evolve_prime_2(z):
             return ((2*z)%23)

         def evolve_prime_3(z):
             return ((2*z)%41)

         def evolve_prime_4(z):
             return ((2*z)%53)

In [66]: print(Floyd_no_print(-500, evolve_prime_1, 100))
         print(Floyd_no_print(-500, evolve_prime_2, 100))
         print(Floyd_no_print(-500, evolve_prime_3, 100))
         print(Floyd_no_print(-500, evolve_prime_4, 100))

(1, 8, 3)
(1, 11, 12)
(1, 20, 25)
(1, 52, 7)
```

It looks like the cycle length is a divisor of p-1.

2.1 The Miller-Rabin primality test

2.1.1 Fermat's Little Theorem and the ROO property of primes

Fermat's Little Theorem states that if p is a prime number, and $GCD(a, p) = 1$, then

$$a^{p-1} \equiv 1 \bmod p.$$

It can be proven by showing that the cycle-lengths for the "multiplication-by-a-mod-p" dynamical system are all divisors of $p - 1$.

Under the assumptions above, if we ask Python to compute `(a**(p-1))%p`, or even better, `pow(a,p-1,p)`, the result should be 1. We use and refine this idea to develop a powerful and practical primality test. Let's begin with a few checks of Fermat's Little Theorem.

```
In [67]: pow(3,36,37)  # a = 3, p = 37, p-1 = 36

Out[67]: 1

In [68]: pow(17,100,101) # 101 is prime.

Out[68]: 1

In [69]: pow(303, 100, 101)  # Why won't we get 1?

Out[69]: 0
```

```
In [70]: pow(5,90,91) # What's the answer?

Out[70]: 64

In [71]: pow(7,12318, 12319) # What's the answer?

Out[71]: 1331
```

We can learn something from the previous two examples. Namely, 91 and 12319 are **not** prime numbers. We say that 7 **witnesses** the non-primality of 12319. Moreover, we learned this fact without actually finding a factor of 12319! Indeed, the factors of 12319 are 97 and 127, which have no relationship to the "witness" 7.

In this way, Fermat's Little Theorem -- a statement about prime numbers -- can be turned into a way of discovering that numbers are not prime. After all, if p is not prime, then what are the chances that $a^{p-1} \equiv 1 \bmod p$ by coincidence?

```
In [72]: pow(3,90,91)

Out[72]: 1
```

Well, ok. Sometimes coincidences happen. We say that 3 is a **bad witness** for 91, since 91 is not prime, but $3^{90} \equiv 1 \bmod 91$. But we could try multiple bases (witnesses). We can expect that someone (some base) will witness the nonprimality. Indeed, for the non-prime 91 there are many good witnesses (ones that detect the nonprimality).

```
In [73]: for witness in range(1,20):
             flt = pow(witness, 90, 91)
             if flt == 1:
                 print("{} is a bad witness.".format(witness))
             else:
                 print("{} raised to the 90th power equals {}, mod 91".format(witness, flt))

1 is a bad witness.
2 raised to the 90th power equals 64, mod 91
3 is a bad witness.
4 is a bad witness.
5 raised to the 90th power equals 64, mod 91
6 raised to the 90th power equals 64, mod 91
7 raised to the 90th power equals 77, mod 91
8 raised to the 90th power equals 64, mod 91
9 is a bad witness.
10 is a bad witness.
11 raised to the 90th power equals 64, mod 91
12 is a bad witness.
13 raised to the 90th power equals 78, mod 91
14 raised to the 90th power equals 14, mod 91
15 raised to the 90th power equals 64, mod 91
16 is a bad witness.
17 is a bad witness.
18 raised to the 90th power equals 64, mod 91
19 raised to the 90th power equals 64, mod 91
```

For some numbers -- the Carmichael numbers -- there are more bad witnesses than good witnesses. For example, take the Carmichael number 41041, which is not prime ($41041 = 7 \cdot 11 \cdot 13 \cdot 41$).

```
In [76]: for witness in range(1,20):
             flt = pow(witness, 41040, 41041)
             if flt == 1:
                 print("{} is a bad witness.".format(witness))
             else:
                 print("{} raised to the 41040th power equals {}, mod 41041".format(witness, fl
```

```
1 is a bad witness.
2 is a bad witness.
3 is a bad witness.
4 is a bad witness.
5 is a bad witness.
6 is a bad witness.
7 raised to the 41040th power equals 29316, mod 41041
8 is a bad witness.
9 is a bad witness.
10 is a bad witness.
11 raised to the 41040th power equals 18656, mod 41041
12 is a bad witness.
13 raised to the 41040th power equals 22100, mod 41041
14 raised to the 41040th power equals 29316, mod 41041
15 is a bad witness.
16 is a bad witness.
17 is a bad witness.
18 is a bad witness.
19 is a bad witness.
```

For Carmichael numbers, it turns out that finding a good witness is just as difficult as finding a factor. Although Carmichael numbers are rare, they demonstrate that Fermat's Little Theorem by itself is not a great way to be certain of primality. Effectively, Fermat's Little Theorem can often be used to quickly prove that a number **is not prime**... but it is not so good if we want to be sure that a number **is prime**.

The Miller-Rabin primality test will refine the Fermat's Little Theorem test, by cleverly taking advantage of another property of prime numbers. We call this the ROO (Roots Of One) property: if p is a prime number, and $x^2 \equiv 1 \bmod p$, then $x \equiv 1$ or $x \equiv -1 \bmod p$.

```
In [77]: for x in range(41):
             if x*x % 41 == 1:
                 print("{} squared is congruent to 1, mod 41.".format(x))  # What numbers do y
```

```
1 squared is congruent to 1, mod 41.
40 squared is congruent to 1, mod 41.
```

Note that we use "natural representatives" when doing modular arithmetic in Python. So the only numbers whose square is 1 mod 41 are 1 and 40. (Note that 40 is the natural representative of -1, mod 41). If we consider the "square roots of 1" with a composite modulus, we find more (as long as the modulus has at least two odd prime factors).

```
In [78]: for x in range(91):
             if x*x % 91 == 1:
                 print("{} squared is congruent to 1, mod 91.".format(x))  # What numbers do y
```

```
1 squared is congruent to 1, mod 91.
27 squared is congruent to 1, mod 91.
64 squared is congruent to 1, mod 91.
90 squared is congruent to 1, mod 91.
```

We have described two properties of prime numbers, and therefore two possible indicators that a number is not prime.

1. If p is a number which violates Fermat's Little Theorem, then p is not prime.
2. If p is a number which violates the ROO property, then p is not prime.

The Miller Rabin test will combine these indicators. But first we have to introduce an ancient algorithm for exponentiation.

2.1.2 Pingala's exponentiation algorithm

If we wish to compute 5^{90} mod 91, without the `pow` command, we don't have to carry out 90 multiplications. Instead, we carry out **Pingala's algorithm**. To understand this algorithm, we begin with the desired exponent (e.g. $e = 90$), and carry out a series of steps: replace e by $e/2$ if e is even, and replace e by $(e-1)/2$ if e is odd. Repeat this until the exponent is decreased to zero. It's a process like the Collatz $3n+1$ process... but it continually makes positive integers smaller.

The following function carries out this process on any input e.

```
In [74]: def Pingala(e):
             current_number = e
             while current_number > 0:
                 if current_number%2 == 0:
                     current_number = current_number // 2
                     print("Exponent {} BIT 0".format(current_number))
                 if current_number%2 == 1:
                     current_number = (current_number - 1) // 2
                     print("Exponent {} BIT 1".format(current_number))
```

```
In [75]: Pingala(90)
```

```
Exponent 45 BIT 0
Exponent 22 BIT 1
Exponent 11 BIT 0
Exponent 5 BIT 1
```

```
Exponent 2 BIT 1
Exponent 1 BIT 0
Exponent 0 BIT 1
```

The codes "BIT 1" and "BIT 0" tell us what happened at each step, and allow the process to be reversed. In a line with BIT 0, the exponent gets **doubled** as one goes **up** one line (e.g., from 11 to 22). In a line with BIT 1, the exponent gets **doubled then increased by 1** as one goes **up** one line (e.g., from 2 to 5).

We can use these BIT codes in order to compute an exponent. Below, we follow the BIT codes to compute 5^{90}.

```
In [79]: n = 1  # This is where we start.
         n = n*n * 5 # BIT 1 is interpreted as square-then-multiply-by-5, since the exponent i
         n = n*n  # BIT 0 is interpreted as squaring, since the exponent is doubled.
         n = n*n * 5 # BIT 1
         n = n*n * 5 # BIT 1 again.
         n = n*n # BIT 0
         n = n*n * 5 # BIT 1
         n = n*n # BIT 0

In [80]: print(n)  # What we just computed.
         print(5**90) # I hope these match!!

807793566946316088741610050849573099185363389551639556884765625
807793566946316088741610050849573099185363389551639556884765625
```

Note that along the way, we carried out 11 multiplications (count the * symbols), and didn't have to remember too many numbers along the way. So this process was efficient in both time and memory. We just followed the BIT code. The number of multiplications is bounded by twice the number of BITs, since each BIT requires at most two multiplications (squaring then multiplication by 5) to execute.

Why did we call the code a BIT code? It's because the code consists precisely of the bits (binary digits) of the exponent 90! Since computers store numbers in binary, the computer "knows" the BIT code as soon as it knows the exponent. In Python, the `bin` command recovers the binary expansion of a number.

```
In [81]: bin(90)  # Compare this to the sequence of bits, from bottom up.

Out[81]: '0b1011010'
```

Python outputs binary expansions as strings, beginning with `'0b'`. To summarize, we can compute an exponent like b^e by the following process:

Pingala's Exponentiation Algorithm

1. Set the number to 1.
2. Read the bits of e, from left to right.

a. When the bit is zero, square the number.

b. When the bit is one, square the number, then multiply by b.

3. Output the resulting number.

```
In [82]: def pow_Pingala(base,exponent):
             result = 1
             bitstring = bin(exponent)[2:] # Chop off the '0b' part of the binary expansion of
             for bit in bitstring: # Iterates through the "letters" of the string.  Here the l
                 if bit == '0':
                     result = result*result
                 if bit == '1':
                     result = result*result * base
             return result

In [83]: pow_Pingala(5,90)

Out[83]: 807793566946316088741610050849573099185363389551639556884765625
```

It is straightforward to modify Pingala's algorithm to compute exponents in modular arithmetic. Just reduce along the way.

```
In [84]: def powmod_Pingala(base,exponent,modulus):
             result = 1
             bitstring = bin(exponent)[2:] # Chop off the '0b' part of the binary expansion of
             for bit in bitstring: # Iterates through the "letters" of the string.  Here the l
                 if bit == '0':
                     result = (result*result) % modulus
                 if bit == '1':
                     result = (result*result * base) % modulus
             return result

In [85]: powmod_Pingala(5,90,91)

Out[85]: 64
```

Let's compare the performance of our new modular exponentiation algorithm.

```
In [86]: %timeit powmod_Pingala(3,999,1000)  # Pingala's algorithm, modding along the way.

1.91 µs ś 17.8 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)


In [87]: %timeit powermod_1(3,999,1000)  # Raise to the power, then mod, using Python built-in

2.51 µs ś 20.5 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)


In [88]: %timeit powermod_2(3,999,1000)  #  Multiply 999 times, modding along the way.
```

```
90.6 ţs ś 3.4 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
```

```
In [89]: %timeit pow(3,999,1000)  #  Use the Python built-in modular exponent.

1.17 ţs ś 2.61 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
```

The fully built-in modular exponentiation `pow(b,e,m)` command is probably the fastest. But our implementation of Pingala's algorithm isn't bad -- it probably beats the simple `(b**e) % m` command (in the `powermod_1` function), and it's certainly faster than our naive loop in `powermod_2`.

One can quantify the efficiency of these algorithms by analyzing how the **time** depends on the **size** of the input parameters. For the sake of exposition, let us keep the base and modulus constant, and consider how the time varies with the size of the exponent.

As a function of the exponent e, our `powmod_Pingala` algorithm required some number of multiplications, bounded by twice the number of bits of e. The number of bits of e is approximately $\log_2(e)$. The size of the numbers multiplied is bounded by the size of the (constant) modulus. In this way, the time taken by the `powmod_Pingala` algorithm should be $O(\log(e))$, meaning bounded by a constant times the logarithm of the exponent.

Contrast this with the slow `powermod_2` algorithm, which performs e multiplications, and has thus has runtime $O(e)$.

2.1.3 The Miller-Rabin test

Pingala's algorithm is effective for computing exponents, in ordinary arithmetic or in modular arithmetic. In this way, we can look for violations of Fermat's Little Theorem as before, to find witnesses to non-primality. But if we look more closely at the algorithm... we can sometimes find violations of the ROO property of primes. This strengthens the primality test.

To see this, we create out a "verbose" version of Pingala's algorithm for modular exponentiation.

```
In [90]: def powmod_verbose(base, exponent, modulus):
             result = 1
             print("Computing {} raised to {}, modulo {}.".format(base, exponent, modulus))
             print("The current number is {}".format(result))
             bitstring = bin(exponent)[2:] # Chop off the '0b' part of the binary expansion of
             for bit in bitstring: # Iterates through the "letters" of the string.  Here the l
                 sq_result = result*result % modulus  # We need to compute this in any case.
                 if bit == '0':
                     print("BIT 0:  {} squared is congruent to {}, mod {}".format(result, sq_re
                     result = sq_result
                 if bit == '1':
                     newresult = (sq_result * base) % modulus
                     print("BIT 1:  {} squared times {} is congruent to {}, mod {}".format(resu
                     result = newresult
             return result

In [91]: powmod_verbose(2,560,561)  # 561 is a Carmichael number.
```

```
Computing 2 raised to 560, modulo 561.
The current number is 1
BIT 1:  1 squared times 2 is congruent to 2, mod 561
BIT 0:  2 squared is congruent to 4, mod 561
BIT 0:  4 squared is congruent to 16, mod 561
BIT 0:  16 squared is congruent to 256, mod 561
BIT 1:  256 squared times 2 is congruent to 359, mod 561
BIT 1:  359 squared times 2 is congruent to 263, mod 561
BIT 0:  263 squared is congruent to 166, mod 561
BIT 0:  166 squared is congruent to 67, mod 561
BIT 0:  67 squared is congruent to 1, mod 561
BIT 0:  1 squared is congruent to 1, mod 561
```

```
Out[91]: 1
```

The function has displayed every step in Pingala's algorithm. The final result is that $2^{560} \equiv 1$ mod 561. So in this sense, 2 is a bad witness. For 561 is not prime (3 is a factor), but it does not violate Fermat's Little Theorem when 2 is the base.

But within the verbose output above, there is a violation of the ROO property. The penultimate line states that "67 squared is congruent to 1, mod 561". But if 561 were prime, only 1 and 560 are square roots of 1. Hence this penultimate line implies that 561 is not prime (again, without finding a factor!).

This underlies the Miller-Rabin test. We carry out Pingala's exponentiation algorithm to compute b^{p-1} modulo p. If we find a violation of ROO along the way, then the test number p is not prime. And if, at the end, the computation does not yield 1, we have found a Fermat's Little Theorem (FLT) violation, and the test number p is not prime.

The function below implements the Miller-Rabin test on a number p, using a given base.

```
In [93]: def Miller_Rabin(p, base):
             '''
             Tests whether p is prime, using the given base.
             The result False implies that p is definitely not prime.
             The result True implies that p **might** be prime.
             It is not a perfect test!
             '''
             result = 1
             exponent = p-1
             modulus = p
             bitstring = bin(exponent)[2:] # Chop off the '0b' part of the binary expansion of
             for bit in bitstring: # Iterates through the "letters" of the string.  Here the l
                 sq_result = result*result % modulus  # We need to compute this in any case.
                 if sq_result == 1:
                     if (result != 1) and (result != exponent):  # Note that exponent is congr
                         return False  # a ROO violation occurred, so p is not prime
                 if bit == '0':
                     result = sq_result
                 if bit == '1':
```

```
                result = (sq_result * base) % modulus
        if result != 1:
            return False  # a FLT violation occurred, so p is not prime.

        return True  # If we made it this far, no violation occurred and p might be prime
```

```
In [94]:  Miller_Rabin(101,6)
```

```
Out[94]: True
```

How good is the Miller-Rabin test? Will this modest improvement (looking for ROO violations) improve the reliability of witnesses? Let's see how many witnesses observe the nonprimality of 41041.

```
In [95]: for witness in range(2,20):
            MR = Miller_Rabin(41041, witness) #
            if MR:
                print("{} is a bad witness.".format(witness))
            else:
                print("{} detects that 41041 is not prime.".format(witness))
```

```
2 detects that 41041 is not prime.
3 detects that 41041 is not prime.
4 detects that 41041 is not prime.
5 detects that 41041 is not prime.
6 detects that 41041 is not prime.
7 detects that 41041 is not prime.
8 detects that 41041 is not prime.
9 detects that 41041 is not prime.
10 detects that 41041 is not prime.
11 detects that 41041 is not prime.
12 detects that 41041 is not prime.
13 detects that 41041 is not prime.
14 detects that 41041 is not prime.
15 detects that 41041 is not prime.
16 is a bad witness.
17 detects that 41041 is not prime.
18 detects that 41041 is not prime.
19 detects that 41041 is not prime.
```

In fact, one can prove that at least 3/4 of the witnesses will detect the non-primality of any non-prime. Thus, if you keep on asking witnesses at random, your chances of detecting non-primality increase exponentially! In fact, the witness 2 suffices to check whether any number is prime or not up to 2047. In other words, if $p < 2047$, then p is prime if and only if `Miller_Rabin(p,2)` is `True`. Just using the witnesses 2 and 3 suffice to check primality for numbers up to a million (1373653, to be precise, according to Wikipedia.)

The general strategy behind the Miller-Rabin test then is to use just a few witnesses for smallish potential primes (say, up to 2^{64}). For larger numbers, try some number x (like 20 or 50) random

bases. If the tested number **is** composite, then the probability of all witnesses reporting `True` is is less than $1/4^x$. With 50 random witnesses, the chance that a composite number tests as prime is less than 10^{-30}.

Note that these are statements about **conditional probability**. In more formal language,

$$\text{Prob (tests prime } | \text{ is composite)} < \frac{1}{4^{\#\text{witnesses}}}.$$

As those who study medical testing know, this probability differs from the probability that most people care about: the probability that a number is prime, given that it tests prime. The relationship between the two probabilities is given by Bayes Theorem, and depends on the **prevalence** of primes among the sample. If our sample consists of numbers of absolute value about N, then the prevalence of primes will be about $1/\log(N)$, and the probability of primality given a positive test result can be approximated.

$$\text{Prob (is prime } | \text{ tests prime)} > 1 - \frac{\log(N) - 1}{4^{\#\text{witnesses}}}.$$

As one chooses more witnesses, this probability becomes extremely close to 1.

```
In [96]: from mpmath import *
         # The mpmath package allows us to compute with arbitrary precision!
         # It has specialized functions for log, sin, exp, etc.., with arbitrary precision.
         # It is probably installed with your version of Python.

         def prob_prime(N, witnesses):
             '''
             Conservatively estimates the probability of primality, given a positive test resu
             N is an approximation of the size of the tested number.
             witnesses is the number of witnesses.
             '''
             mp.dps = witnesses # mp.dps is the number of digits of precision.  We adapt this
             prob_prime = 1 - (log(N) - 1) / (4**witnesses)
             print(str(100*prob_prime)+"% chance of primality") # Use str to convert mpmath fl

In [97]: prob_prime(10**100, 50) # Chance of primality with 50 witnesses, if a 100-digit numbe

99.999999999999999999999999981914692482445195916316% chance of primality
```

We implement the Miller-Rabin test for primality in the `is_prime` function below.

```
In [98]: def is_prime(p, witnesses=50):  # witnesses is a parameter with a default value.
             '''
             Tests whether a positive integer p is prime.
             For p < 2^64, the test is deterministic, using known good witnesses.
             Good witnesses come from a table at Wikipedia's article on the Miller-Rabin test,
             based on research by Pomerance, Selfridge and Wagstaff, Jaeschke, Jiang and Deng.
             For larger p, a number (by default, 50) of witnesses are chosen at random.
             '''
```

```
    if (p%2 == 0): # Might as well take care of even numbers at the outset!
        if p == 2:
            return True
        else:
            return False

    if p > 2**64:  # We use the probabilistic test for large p.
        trial = 0
        while trial < witnesses:
            trial = trial + 1
            witness = randint(2,p-2) # A good range for possible witnesses
            if Miller_Rabin(p,witness) == False:
                return False
        return True

    else:  # We use a determinisic test for p <= 2**64.
        verdict = Miller_Rabin(p,2)
        if p < 2047:
            return verdict # The witness 2 suffices.
        verdict = verdict and Miller_Rabin(p,3)
        if p < 1373653:
            return verdict # The witnesses 2 and 3 suffice.
        verdict = verdict and Miller_Rabin(p,5)
        if p < 25326001:
            return verdict # The witnesses 2,3,5 suffice.
        verdict = verdict and Miller_Rabin(p,7)
        if p < 3215031751:
            return verdict # The witnesses 2,3,5,7 suffice.
        verdict = verdict and Miller_Rabin(p,11)
        if p < 2152302898747:
            return verdict # The witnesses 2,3,5,7,11 suffice.
        verdict = verdict and Miller_Rabin(p,13)
        if p < 3474749660383:
            return verdict # The witnesses 2,3,5,7,11,13 suffice.
        verdict = verdict and Miller_Rabin(p,17)
        if p < 341550071728321:
            return verdict # The witnesses 2,3,5,7,11,17 suffice.
        verdict = verdict and Miller_Rabin(p,19) and Miller_Rabin(p,23)
        if p < 3825123056546413051:
            return verdict # The witnesses 2,3,5,7,11,17,19,23 suffice.
        verdict = verdict and Miller_Rabin(p,29) and Miller_Rabin(p,31) and Miller_Rab
        return verdict # The witnesses 2,3,5,7,11,17,19,23,29,31,37 suffice for testi
```

In [122]:
```
def is_prime_trial(n):
    '''
    Checks whether the argument n is a prime number.
    Uses a brute force search for factors between 1 and n.
```

```
            '''
            j = 2
            root_n = sqrt(n)
            while j <= root_n:  # j will proceed through the list of numbers 2,3,... up to s
                if n%j == 0:  # is n divisible by j?
                    return False
                j = j + 1  # There's a Python abbreviation for this:  j += 1.
            return True
```

```
In [99]: is_prime(10000000000000666000000000000001)  # This is Belphegor's prime.
```

```
Out[99]: True
```

How fast is our new `is_prime` function? Let's give it a try.

```
In [100]: %timeit is_prime(234987928347928347928347928734987398792837491)
```

```
97.7 ţs ś 1.48 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
```

```
In [101]: %timeit is_prime(10000000000000666000000000000001)
```

```
2.61 ms ś 16.6 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

The results will probably on the order of a millisecond, perhaps even a tenth of a millisecond (10^{-4} seconds) for non-primes! That's much faster than looking for factors, for numbers of this size. In this way, we can test primality of numbers of hundreds of digits!

For an application, let's find some Mersenne primes. Recall that a Mersenne prime is a prime of the form $2^p - 1$. Note that when $2^p - 1$ is prime, it must be the case that p is a prime too. We will quickly find the Mersenne primes with p up to 1000 below!

```
In [111]: for p in range(1,1000):
              if is_prime(p):  # We only need to check these p.
                  M = 2**p - 1 # A candidate for a Mersenne prime.
                  if is_prime(M):
                      print("2^{} - 1 = {} is a Mersenne prime.\n".format(p,M))
```

```
2^2 - 1 = 3 is a Mersenne prime.

2^3 - 1 = 7 is a Mersenne prime.

2^5 - 1 = 31 is a Mersenne prime.

2^7 - 1 = 127 is a Mersenne prime.

2^13 - 1 = 8191 is a Mersenne prime.

2^17 - 1 = 131071 is a Mersenne prime.
```

```
2^19 - 1 = 524287 is a Mersenne prime.

2^31 - 1 = 2147483647 is a Mersenne prime.

2^61 - 1 = 2305843009213693951 is a Mersenne prime.

2^89 - 1 = 618970019642690137449562111 is a Mersenne prime.

2^107 - 1 = 162259276829213363391578010288127 is a Mersenne prime.

2^127 - 1 = 170141183460469231731687303715884105727 is a Mersenne prime.

2^521 - 1 = 6864797660130609714981900799081393217269435300143305409394463459185543183397656052
```

2.1.4 Exercises

1. If $2^p - 1$ is a Mersenne prime, then Euclid proved that $(2^p - 1) \cdot 2^{p-1}$ is a perfect number. Find all the (even) perfect numbers up to 2^{1000}. (Note: nobody has ever found an odd perfect number. Euler proved that all even perfect numbers arise from Mersenne primes by Euclid's recipe.)

2. The Fermat sequence is the sequence of numbers 3, 5, 257, 65537, etc., of the form $2^{2^n} + 1$ for $n \geq 0$. Test the primality of these numbers for n up to 10.

3. Why does the is_prime function (using Miller-Rabin) run more quickly on non-primes than it does on primes?

4. Compare the performance of the new `is_prime` function to "trial division" (looking for factors up to the square root of the test number). Which is faster for small numbers (1-digit, 2-digits, 3-digits, etc.)? Adapt the `is_prime` function to perform trial division for small numbers in order to optimize performance.

5. Estimate the probability that a randomly chosen 10-digit number is prime, by running is_prime on a large number of samples. How does this probability vary as the number of digits increases (e.g., from 10 digits to 11 digits to 12 digits, etc., onto 20 digits)?

```
In [110]: #1
          for p in range(1,1000):
              if is_prime(p):
                  M = 2**p - 1
                  if is_prime(M):
                      print("{} is a perfect number\n".format(M*2**(p-1)))
```

```
6 is a perfect number
```

```
28 is a perfect number

496 is a perfect number

8128 is a perfect number

33550336 is a perfect number

8589869056 is a perfect number

137438691328 is a perfect number

2305843008139952128 is a perfect number

2658455991569831744654692615953842176 is a perfect number

191561942608236107294793378084303638130997321548169216 is a perfect number

13164036458569648337239753460458722910223472318386943117783728128 is a perfect number

1447401115466452442794637312608598848157367749147483588906635434913119915212
8 is a perfect numb

2356272345726734706578954899670990498847754785839260071014302759750633728317862223973036553960
2

1410537837067120690632079580860631898814867435147156678388386759999548677426523801141041933290
3
```

```
In [109]: #2
          for n in range(10):
              check = (2**2**n) + 1
              if is_prime(check):
                  print('{} is prime'.format(check))
```

```
3 is prime
5 is prime
17 is prime
257 is prime
65537 is prime
```

3 3

Because the algorithm relies on getting to the end of it to tell if a number is prime. So if it doesn't get to the end of the algorithm then it's not prime.

```
In [124]: #4
          %timeit is_prime(115)
```

```
1.87 µs ś 39.6 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

```
In [126]: %timeit is_prime(7)
```

```
1.2 µs ś 20.2 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
```

```
In [125]: %timeit is_prime_trial(115)
```

```
11.7 µs ś 169 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

```
In [127]: %timeit is_prime_trial(7)
```

```
11.5 µs ś 479 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

The Miller-Rabin is_prime function is faster than the trial division is_prime function in every case.

```
In [162]: #5
          def test_percent_primes(iterations, digit_length):
              primes = 0

              for n in range(iterations):
                  if is_prime(randint(10**(digit_length-1), 10**digit_length - 1)):
                      primes += 1

              print('digit length = {}, {:.3f}% are prime'.format(digit_length, (primes/total)*
```

```
In [163]: for n in range(10,21):
              test_percent_primes(100000,n)
```

```
digit length = 10, 4.513% are prime
digit length = 11, 4.191% are prime
digit length = 12, 3.796% are prime
digit length = 13, 3.510% are prime
digit length = 14, 3.181% are prime
digit length = 15, 2.982% are prime
digit length = 16, 2.713% are prime
digit length = 17, 2.559% are prime
digit length = 18, 2.462% are prime
digit length = 19, 2.320% are prime
digit length = 20, 2.156% are prime
```

The number of random primes decreases as the length of the digits increases.