Kernel: SageMath 10.7

Detecting Large Primes

MATH 242 Modern Computational Math

Sieve methods such as Eratosthenes or Sundaram are efficient for generating lists of primes up to a hundred million or so. However, what if we need to find primes that are far larger? For example, certain encryption methods depend on the availability of prime numbers with *hundreds* of digits. Neither trial division nor sieve methods are useful at this scale, so how can we find such primes? Today we will examine one method of finding large primes.

Fermat's Little Theorem

Fermat's little theorem says that if p is prime and a is not divisible by p, then $a^{n-1} \equiv 1 \pmod p$.

Verify Fermat's little theorem for some primes p. What happens if p is not prime?

In [1]:

```
# primes
p = 17
print(f"p = {p}:")
print( [ a^(p-1) % p for a in range(1,p)  ] )

p = 89
print(f"\np = {p}:")
print( [ a^(p-1) % p for a in range(1,p)  ] )

p = 151
print(f"\np = {p}:")
print( [ a^(p-1) % p for a in range(1,p)  ] )
```

Out[1]:

```
p = 17:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

p = 89:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

p = 151:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In [2]:

```
# composites
p = 18
print(f"p = {p}:")
print( [ a^(p-1) % p for a in range(1,p)  ] )

p = 35
print(f"\np = {p}:")
print( [ a^(p-1) % p for a in range(1,p)  ] )

p = 105
print(f"\np = {p}:")
print( [ a^(p-1) % p for a in range(1,p)  ] )
```

Out[2]:

```
p = 18:
[1, 14, 9, 16, 11, 0, 13, 8, 9, 10, 5, 0, 7, 2, 9, 4, 17]

p = 35:
[1, 9, 4, 11, 30, 1, 14, 29, 16, 25, 11, 9, 29, 21, 15, 16, 4, 4, 16, 15, 21, 29, 9, 11,
```

```
25, 16, 29, 14, 1, 30, 11, 4, 9, 1]

p = 105:
[1, 46, 51, 16, 25, 36, 91, 1, 81, 100, 16, 81, 1, 91, 15, 46, 16, 51, 46, 85, 21, 1, 46,
51, 100, 46, 36, 91, 1, 60, 16, 16, 81, 1, 70, 36, 46, 16, 51, 25, 1, 21, 1, 46, 30, 16,
46, 36, 91, 85, 81, 16, 16, 81, 85, 91, 36, 46, 16, 30, 46, 1, 21, 1, 25, 51, 16, 46, 36,
70, 1, 81, 16, 16, 60, 1, 91, 36, 46, 100, 51, 46, 1, 21, 85, 46, 51, 16, 46, 15, 91, 1,
81, 16, 100, 81, 1, 91, 36, 25, 16, 51, 46, 1]
```

Fermat's little theorem as a primality test

Fermat's little theorem provides a way to check whether a number n is prime without having to consider divisors of n. How could this work? Does Fermat's little theorem allow us to conclude *for sure* that n is prime? Does it allow us to conclude *for sure* that n is composite?

In [4]:
```
# fermat's little theorem primality test
# uses base a to check whether n might be prime
# input: integers n and a, with 1 < a < n
# output: False indicates that n is composite; True indicates that n is probably prime
def fltpt(n, a):
    return a^(n-1) % n == 1
```

Try it out:

In [5]:
```
fltpt(97, 12)
```

Out[5]:
```
True
```

In [7]:
```
fltpt(105,8)
```

Out[7]:
```
True
```

In [8]:
```
fltpt(36, 7)
```

Out[8]:
```
False
```

Does this method let you check whether BIG integers are prime?

In [9]:
```
fltpt(1038455544353, 34598236)
```

Out[9]:
```
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
Cell In[9], line 1
----> 1 fltpt(Integer(1038455544353), Integer(34598236))
Cell In[4], line 6, in fltpt(n, a)
      5 def fltpt(n, a):
----> 6     return a**(n-Integer(1)) % n == Integer(1)
File /ext/sage/10.7/src/sage/rings/integer.pyx:2210, in sage.rings.integer.Integer.__pow__()
   2208
   2209         if type(left) is type(right):
-> 2210             return (<Integer>left)._pow_(right)
   2211         elif isinstance(left, Element):
   2212             return coercion_model.bin_op(left, right, operator.pow)
File /ext/sage/10.7/src/sage/rings/integer.pyx:2274, in sage.rings.integer.Integer._pow_()
   2272
   2273         if mpz_fits_slong_p(exp):
-> 2274             return self._pow_long(mpz_get_si(exp))
   2275
   2276         # Raising to an exponent which doesn't fit in a long overflows
File /ext/sage/10.7/src/sage/rings/integer.pyx:2306, in
sage.rings.integer.Integer._pow_long()
   2304 if n > 0:
   2305     x = PY_NEW(Integer)
-> 2306     sig_on()
```

```
   2307     mpz_pow_ui(x.value, self.value, n)
   2308     sig_off()
FloatingPointError: Floating point exception
```

Modular exponentiation

In order to find large primes, we need to be able to compute $b^e \pmod m$ for large integers b, e, and m. For example, these integers might have *hundreds* of digits.

Observe that computing $b^e \pmod m$ is impractical if b, e, and m have even nine digits.

In [10]:
```
a = randrange(10**8, 10**9)
print(a)
b = randrange(10**8, 10**9)
print(b)
m = randrange(10**8, 10**9)
print(m)
```

Out[10]:
```
746189099
339742896
587809530
```

In [11]:
```
a^b % m
```

Out[11]:
```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[11], line 1
----> 1 a**b % m
File signals.pyx:355, in cysignals.signals.python_check_interrupt()
KeyboardInterrupt:
```

The following algorithm uses repeated squaring to efficiently compute $b^e \pmod m$ for huge integers.

In [13]:
```
# function modpow
# input: integers b, e, and m
# output: b^e (mod m)
def modpow(b, e, m):
    # initialize result
    result = 1

    # main loop
    while e > 0:
        # if x is odd, then multiply result by b and reduce mod m
        if e % 2 == 1:
            result = result*b % m

        # square b and reduce mod m
        b = b*b % m

        # divide x by 2 and ignore the remainder
        e = e // 2

    # done
    return result
```

Try it out! Observe that you can now compute $b^e \pmod m$ for very large integers b, e, and m.

In [14]:
```
modpow(a,b,m)
```

Out[14]:
```
156168181
```

In [16]:
```
a = randrange(10**50, 10**51)
print(a)
b = randrange(10**50, 10**51)
```

```
print(b)
m = randrange(10**50, 10**51)
print(m)

modpow(a, b, m)
```

Out[16]:
```
28406834683139601841662961257341896446755418586635791457766862832617186409360134823276834674625621477098771315252985563554833385967195498920565202988189884706207265601651336950717748665998527292835929876 73
```

Probabilistic Primality Testing

We can use Fermat's little theorem and our **modpow** function to determine, with high probability of being correct, whether large integers are prime.

Given an integer n, choose an integer $1 < a < n$ and compute $a^{n-1} \pmod{n}$. Repeat this several times. If any of these trials result in a number other than 1, then n is not prime, so return **False**. Otherwise, n is probably prime, so return **True**.

Implement this algorithm below.

In [19]:
```
# function fermatPrime
# input: integer n to test, and integer numTrials
# output: False means n is composite; True means n is probably prime
def fermatPrime(n, numTrials):
    # repeat numTrials times
    for i in range(numTrials):

        # choose random integer a between 2 and n-1
        a = randrange(2, n)

        # compute b = a^(n-1)   (mod n)
        b = modpow(a, n-1, n)

        # if b is not 1, then return False
        if b != 1:
            return False

    # if loop finishes, then return True
    return True
```

Test your algorithm. Here are some known primes:

- 104393
- 3267000013
- 5467325746163067945 7

In [0]:

In [0]:

In [0]:

Now find your own 50-digit prime

In [21]:
```
for i in range(500):
    n = randrange(10^49, 10^50)
    if fermatPrime(n, 20):
```

```
        print(f"{n} is probably prime")
        print(f"SageMath is_prime(n) returns {is_prime(n)}")
        print("\n")
```

Out[21]:
```
819728562394768994420155311709092737355661601281 27 is probably prime
SageMath is_prime(n) returns True


196306532080310101942186720210890523294121695048 63 is probably prime
SageMath is_prime(n) returns True
```

In [0]:

In [0]:

Probabilistic primality testing might fail

If there is a composite integer n such that $a^n \equiv a \pmod{n}$ for all positive integers a less than n, then our primality test based on Fermat's little theorem might mistakenly identify n as a prime. Unfortunately, there are such composite integers. Can you find one?

In [0]:

In [0]:

In [0]: