Kernel: SageMath 10.4

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 [2]:
```
p=7
print( [ a^(p-1) for a in range(1,p) ] )
print( [ a^(p-1) % p for a in range(1,p) ] )
```

Out[2]:
```
[1, 64, 729, 4096, 15625, 46656]
[1, 1, 1, 1, 1, 1]
```

In [3]:
```
p=23
print( [ a^(p-1) for a in range(1,p) ] )
print( [ a^(p-1) % p for a in range(1,p) ] )
```

Out[3]:
```
[1, 4194304, 31381059609, 17592186044416, 2384185791015625,
131621703842267136, 3909821048582988049, 7378697629483820646 4,
98477090218361123288 1, 100000000000000000000000, 8140274938683976111332 1,
55206143891243641759334 4, 32118388779548551051573 69,
16398978063355821105872896, 74818276426792144775390625,
30948500982134506872478 1056, 11745628765211484589740 62689,
41304285341123293285177098 24, 13569980418174090907801371961,
41943040000000000000000000000, 12269432738610563294900361284 1,
34142787736421955739664672358 4]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In [4]:
```
p=24
print( [ a^(p-1) for a in range(1,p) ] )
print( [ a^(p-1) % p for a in range(1,p) ] )
```

Out[4]:
```
[1, 8388608, 94143178827, 70368744177664, 11920928955078125,
789730223053602816, 27368747340080916343, 5902958103587056517l2,
8862938119652501095929, 100000000000000000000000,
8954302432552373722465 31, 662473726694923701112 0128,
417539054134131163670457 97, 22958569288698149548222 0544,
112227414640188217163085 9375, 49517601571415210995964 96896,
199675689008595238025590 65713, 74347713614021927913318 776832,
257829627945307727248226 067259, 83886080000000000000000 000000,
257658087510821829192907 5869661, 75114133020128302627262 27918848,
208804679998479120343550 32910567]
[1, 8, 3, 16, 5, 0, 7, 8, 9, 16, 11, 0, 13, 8, 15, 16, 17, 0, 19, 8, 21,
16, 23]
```

How can we use Fermat's little theorem as a primality test?

In [5]:
```
# use Fermat's little theorem to check p for primality using base a
def fltpt(p, a):
    b = a^(p-1) % p
    return b == 1  # compare b with 1 and return the result (True or
False)
```

In [6]:
```
fltpt(23, 5)
```

Out[6]: True

In [7]:
```
fltpt(101, 94)
```

Out[7]: True

In [8]:
```
fltpt(306, 55)
```

Out[8]: False

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 [9]:
```
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[9]:
```
366569385
736705243
458164788
```

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

Out[11]:
```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[11], line 1
----> 1 a**b
File src/cysignals/signals.pyx:341, in cysignals.signals.python_check_interrupt()
KeyboardInterrupt:
```

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

In [12]:
```
# returns 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
```

In [15]:
```
a = randrange(10^20, 10^21)
print(a)
b = randrange(10**20, 10**21)
print(b)
m = randrange(10**20, 10**21)
print(m)
```

Out[15]:
```
62435497333706566822 0
37026472237178458755 4
29651189609245298844 8
```

In [16]:
```
modpow(a, b, m)
```

Out[16]: 120466873282658764800

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. Write your algorithm here:

In [17]:
```
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
- 54673257461630679457

In [18]:
```
fermatPrime(54673257461630679457, 20)
```

Out[18]: True

In [19]:
```
fermatPrime(54673257461630679457*2352521, 20)
```

Out[19]: False

Compare with the built-in `is_prime()` function:

In [20]:
```
is_prime(54673257461630679457)
```

Out[20]: True

Now find your own 50-digit prime

In [0]:

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]: