# Miller Rabin, Fast Mod Expo 10/18/2024

Monday, January 22, 2024 1:18 PM

Recap

Fermat's Theorem: if p is prime, gcd(a, p) = 1

$$a^{p-1} \equiv 1 \pmod{p}$$

If we repeatedly exponentiate any a and look at the result mod p, we'll "loop back to the beginning in p-1 steps, guaranteed.

As an example, let's pick p = 7, a = 2, a = 3

| exp | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| a=2 | 1 | 2 | 4 | 1 | 2 | 4 | 1 |
| a=3 | 1 | 3 | 2 | 6 | 4 | 5 | 1 |

Euler's Theorem: A generalization on Fermat's: if gcd(a, n) = 1, then

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

What Euler did is generalize Fermat's result for all integers. He found the guaranteed "looping cycle size" for all integers n, and ended up defining a function that was very useful to get the result, called the Euler Phi Function.

If n is composite, the phi(n) < n-1.
If n is prime, then phi(n) = n-1

Primality Testing…

Straight forward trial division is too slow for large integers (100 digits)

One idea for a property to test is cycle size. If p is prime, it's guaranteed that

$$a^{p-1} \equiv 1 \pmod{p}$$

$$a^1 \equiv 1 \pmod{p}$$

If p is NOT prime, this may or may not be true, but if we find for some value n and another value a that

$$a^{n-1} \not\equiv 1 \pmod{n}$$

Then, that's proof that n is NOT prime.

Our general is as follows when testing if n is prime:

1. Pick a random value of a, 1 < a < n.
2. If gcd(a, n) != 1, return not prime.
3. Calculate a raised to the power n-1 mod n.
    a. if this value is not equal to 1, report n is composite
    b. else, report "IS PROBABLY PRIME"

This algorithm isn't perfect. It's possible it may call a composite number "is probably prime". But it will never call a prime number composite.

This test isn't necessarily that bad, it works in a vast majority of cases. Two issues:

1. Sometimes accidentally, you might pick an a for which the remainder happens to be 1, but n is composite. (How often might this occur?)
2. There are some special numbers, called Carmichael Numbers, that are composite, but for all a such that gcd(a, n) = 1, it happens to be the case that

$$a^{n-1} \equiv 1 \pmod{n}$$

Miller-Rabin came up with a primality that is based on Fermat's Theorem, but "fixes" both of the issues above.

Except Carmichael numbers, the probability that a to the n-1 is 1 mod n, when n is composite is less than 1/2.

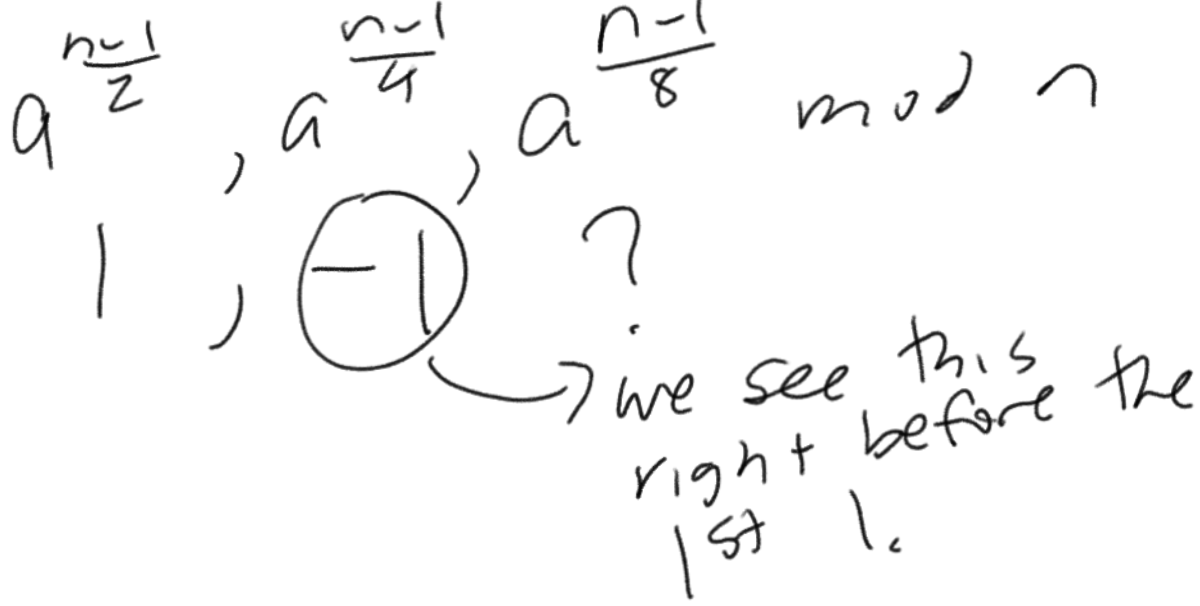
Solution to this: Repeat the test many times with different values of a, so if we repeat it 100 times, the probability of getting a false positive (it tells me "is probably prime" when it's actually composite) ends up being

$$< \left(\frac{1}{2}\right)^{100} \rightarrow \text{really small!}$$

Imagine that your chance of making a free throw is slightly less than 50%, or maybe exactly 50%. You might make one, or you might even make 2 in a row. But it's near impossible for you to make 100 in a row. We fix issue #1 by simply testing over and over again with different values of a. If n is composite, with extremely high probability, one of those values of a will find it out.

For all primes, the cycle size for all a is a divisor of p - 1. (Note that p - 1 is always even, so we can divide out powers of 2 from p - 1.)
Note that -1 squared is 1.

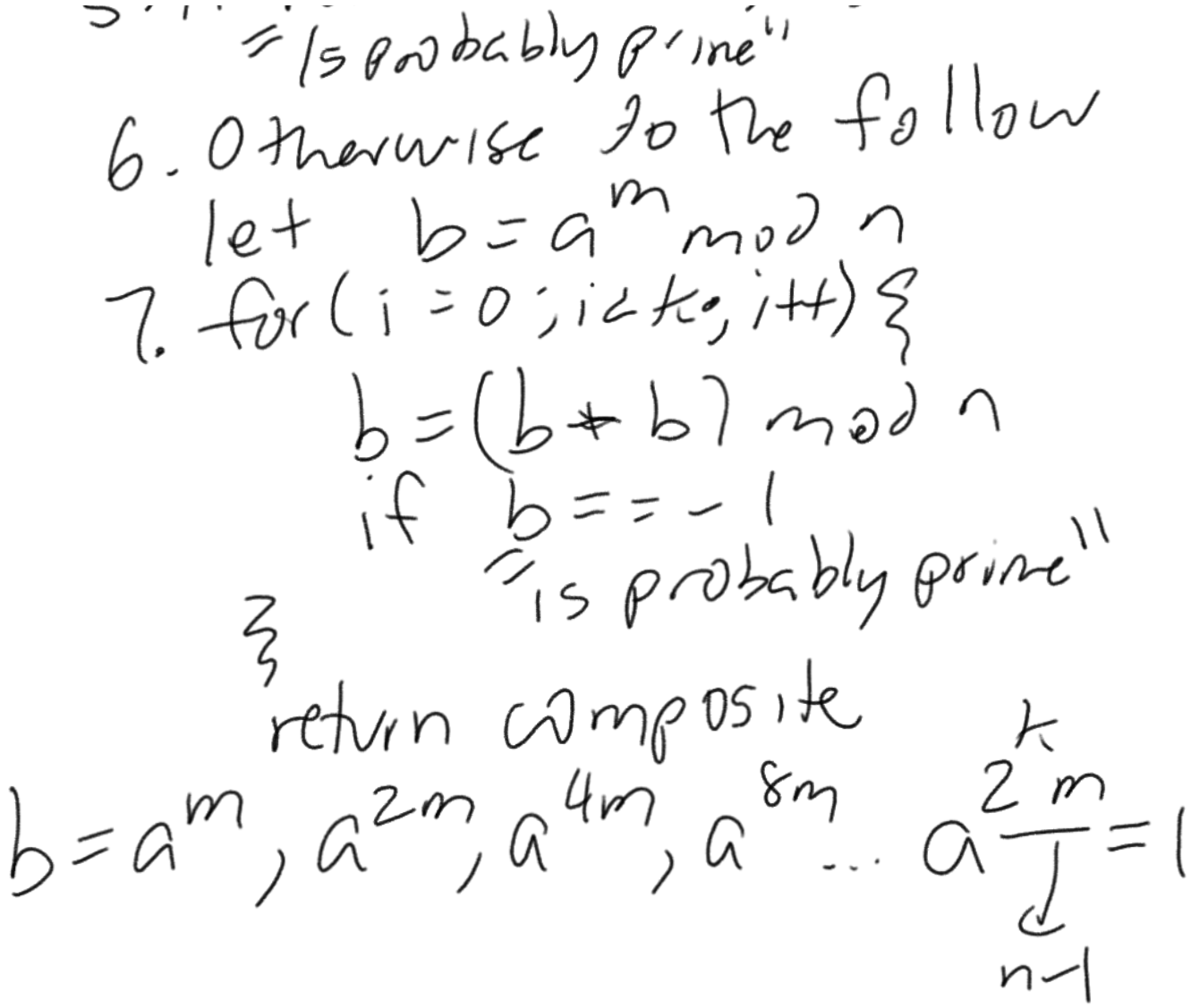
If we try the following:




But, this isn't true of Carmichael numbers. Miller-Rabin utilized this fact.

The Miller-Rabin test utilizes this fact. Here is the algorithm for testing if n is prime:

1. Write $n-1 = 2^k m$, $m \in$ Odd
2. Pick random a, $1 < a < n$.
3. If $\gcd(a,n) \neq 1$, → composite
4. Compute $a^m \mod n$
5. If remainder is 1, say "probably prime"




Fast Modular Exponentiation

In order to get this to work, we have to be able to calculate a modular exponent quickly.

The regular for loop is too slow:

```
res = 1
for (int i=0; i<n-1; i++)
    res = (res*a)%n;
```

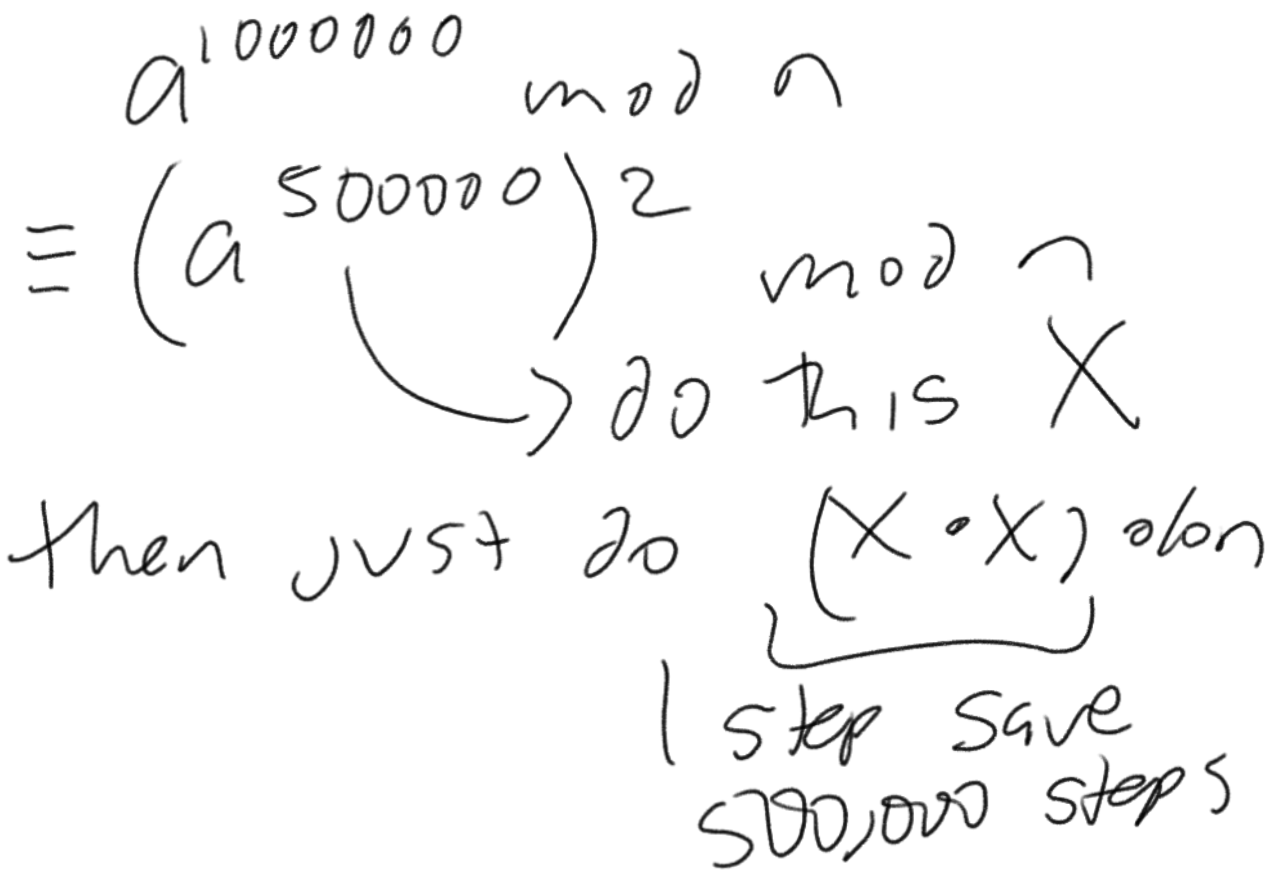
run time O(n), and that's too slow for us because n might be very large (hundred digits or more)

Quick note:

Fast Modular Exponentiation is built into Python

pow(base, exp, mod)

Idea is as follows: Consider computing




Our format:

Input
---
First line has an integer n, representing the number of cases
Each case is on a single line and has 3 space separated positive integer:
b, e, n

Output
---
For each case, on a line by itself, output the remainder when b raised to the power e is divided by n.

Standard Input, Standard Outut

```
// n>1
mypow(b,e,n):
    if e==0:
        return 1
```

return 1

if e%2 == 0:

tmp = mypow(b, e//2, n)

return (tmp * tmp) %n

return (b * mypow(b, e-1, n)) %n

$b^e = b^1 \times b^{e-1}$ ✓