How to generate big prime numbers — Miller-Rabin

A prime number is a positive integer, greater than 1, that has only two positive divisors: 1 and itself.

Here are the first prime numbers:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ..
There is an infinity of prime numbers ! But as we are going to see, big prime numbers are hard to find.

But big primes numbers are useful for some applications, like cryptography.

The security of several public-key cryptography algorithms is based on the fact that big prime numbers are hard to find. A well known example is the *RSA algorithm*. During the key generation step, you have to choose 2 primes, **p** and **q**, and calculate their product, **n = p*q**. The security of RSA is based on the fact that it's very hard to find **p** and **q** given **n**.

Of course, the bigger **p** and **q** are, the harder it is to find them given **n**.

So let's see how we can generate big prime numbers.

I chose to write the code samples in Python for 3 main reasons:

- I love Python ❤ ❤
- The language is really simple to understand, even for someone who've never coded in Python

- Python handle big numbers natively and it’s really nice, because we will work with numbers far bigger than 32 or 64 bits.

You can find the full code at the end of the article.

The problem of prime numbers

There is no pattern to find prime numbers, so how can we find primes ?

Prime numbers density

π(n) is the number of prime numbers ≤ **n**.
For example **π(10) = 4**, because *2, 3, 5* and *7* are the only primes ≤ **10**.

The *prime number theorem* states that **n / ln(n)** is a good approximation of **π(n)** because when **n** tends to infinity, **π(n) / (n / ln(n)) = 1**.

It means the probability that a randomly chosen number is prime is **1 / ln(n),** because there are **n** positive integers ≤ **n** and approximately **n / ln(n)** primes, and **(n / ln(n)) / n = (1 / ln(n))**

For example, the probability to find a prime number of 1024 bits is **1 / (ln(2^{1024})) = (1 / 710)**

As we know that primes are odd (except 2), we can increase this probability by 2, so in average, to generate a 1024 bits prime number, we have to test **355** numbers randomly generated.

How to test that a number is prime

There is a simple way to be sure that an integer (I will call it **n**) is prime.
We need to divide **n** by each integer **d** such that **1 < d < n**.
If one value of **d** divides **n**, then **n** is *composite*.
Else, the only divisors of **n** are **1** and **n**. So by definition, **n** is *prime*.

There are some easy improvements by the way:

- We only have to test to **sqrt(n)**, because if **n = p*q** (composite), **p ≤ sqrt(n)** or **q ≤ sqrt(n)**.
  In the case where **p = q**, we have **n = p*p** with **p = sqrt(n)**.
  In the other case, where **p != q**, either **p > sqrt(n)** and **q < sqrt(n)**, or **q > sqrt(n)** and **p < sqrt(n)**.
- We only have to test with **2** and all the **odd** numbers to **sqrt(n)** because **2** divides all even numbers, so if **2** doesn’t divide **n**, the others even numbers will not.

```
def is_prime(n):



    # test if n is even



    if n % 2 == 0:



        return False# test each odd number from 3 to sqrt(n)
    for i in range(3, sqrt(n), 2):

        if n % d == 0:

            return False# n is necessarily prime

    return True
```

Runtime complexity: **O(sqrt(n))**

Not bad, but in practice, when **n** is very big (an integer on 1024 bits, or more), it takes a while ..

So how can we generate big primes fastly for cryptography purposes ?

Probabilistic tests

Probabilistic algorithms are much faster that the one we see just before, but we can’t be sure at 100% that **n** is prime.

In fact, a probabilistic test is absolutely right when it says that **n** is composite. But when it says that **n** is prime, there is a (very low) chance that **n** is actually not prime.

Here, we will see a famous probabilistic algorithm, called *Miller-Rabin*. But before, let’s see few maths that will help us to understand how this algorithm works.

> *If you don’t know modular arithmetic, take a look on Google it will take you 5 minutes to understand the basics.*

Fermat primality test

Given **n**, a prime number, and **a**, an integer such that **$1 \le a \le n-1$**,
the *Fermat's little theorem* says that **a^(n-1) = 1 (mod n)**.

So for a integer **n**, we just have to find a value of **a** for which **a^(n-1) != 1 (mod n)** to proove that **n** is composite. Such value for **a** is called a *Fermat witness*.

In the other hand, if we find a value of **a** that verify the Fermat's theorem, we just show that **n** satisfies Fermat's theorem for the base **a**, and appears to be prime. In this case, we said that **n** is *pseudo-prime of base a*.

But if in a later test, we finally found that this **n** is composite, then, the previous values of **a** for whose **a^(n-1) != 1 (mod n)** are called *Fermat liars*, because they lied about the fact that **n** is actually composite (bouuuuuh, liars).

Carmichael numbers

There are some composite numbers that satisfies
the *Fermat's little theorem* for all possible values of **a**. As you guessed, these numbers are called … *Carmichael numbers* (so much suspense…).

The first 3 are **561**, **1105** and **1729**.

There are only **255** Carmichael numbers **$< 10^8$**,
and **20138200 $< 10^{21}$** !
So if you generate a random number **$n < 10^8$,** the probability that **n** is a Carmichael numbers is **2.55*10^(-6)**.
As you can see, it's very low ! But the Fermat primality test is not perfect, because of these numbers.

Miller-Rabin is more advanced than Fermat’s primality test, and Carmichael numbers are not a problem.

Trivial and nontrivial square root

We define **$p > 2$**, a prime.

We know that **1** and **-1** always give **1** when squared: **$1^2 = (-1)^2 = 1 \pmod{p}$**. They are called *trivial square root*.

But sometimes, there are *nontrivial square root* of **1** modulo **p**.
We define **a**, an integer, to be a *nontrivial square root* of **1 (mod p)** if
$a^2 = 1 \pmod{p}$.

> *For example:*
> ***$3^2 = 9 = 1 \pmod{8}$***
> *so **3** is a non trivial square root of **1** modulo **8**.*

If **1** has a square root other than **1** and **-1** modulo **n** (a nontrivial square root), then **n** must be composite.

Miller-Rabin

The goal of Miller-Rabin is to find a nontrivial square roots of **1** modulo **n**.

Take back the *Fermat’s little theorem*: **$a^{(n-1)} = 1 \pmod{n}$**.

For Miller-Rabin, we need to find **r** and **s** such that **$(n-1) = r*(2^s)$**, with **r** odd.

Then, we pick **a**, an integer in the range **[1, n-1]**.

- If **a^r != 1 (mod n)** and **a^((2^j)r) != -1 (mod n)** for all **j** such that **0 ≤ j ≤ s-1**, then **n** is not prime and **a** is called a *strong witness to compositeness for n*.
- In the other hand, if **a^r = 1 (mod n)** or **a^((2^j)r) = -1 (mod n)** for some **j** such as **0 ≤ j ≤ s-1**, then **n** is said to be a *strong pseudo-prime to the base a*, and **a** is called a *strong liar to primality for n*.

So, how to generate big prime numbers ?

Now, we know all the theory we need to generate prime numbers. So .. let’s do it ! :)

The algorithm

- **Generate a prime candidate**. Say we want a 1024 bits prime number. Start by generating 1024 bits randomly. Set the MSB to 1, to make sure that the number hold on 1024 bits. Set the LSB to 1 to make be sure that it’s an odd number.
- **Test if the generated number is prime** with Miller-Rabin. Run the test many time to make it more efficient.
- If the number is not prime, **restart** from the beginning.

Full code

Here’s my Python implementation.

```
from random import randrange, getrandbitsdef is_prime(n, k=128):



    """ Test if a number is prime           Args:
                n -- int -- the number to test

                k -- int -- the number of tests to do           return True if n is prime

    """

    # Test if n is not even.

    # But care, 2 is prime !

    if n == 2 or n == 3:

        return True

    if n <= 1 or n % 2 == 0:

        return False

    # find r and s

    s = 0

    r = n - 1

    while r & 1 == 0:
        s += 1

        r //= 2

    # do k tests

    for _ in range(k):

        a = randrange(2, n - 1)

        x = pow(a, r, n)

        if x != 1 and x != n - 1:

            j = 1

            while j < s and x != n - 1:

                x = pow(x, 2, n)

                if x == 1:

                    return False

                j += 1
```

```
              if x != n - 1:

                   return False      return Truedef
generate_prime_candidate(length):

     """ Generate an odd integer randomly           Args:

              length -- int -- the length of the number to generate, in bits
return a integer

     """

     # generate random bits

     p = getrandbits(length)

     # apply a mask to set MSB and LSB to 1

     p |= (1 << length - 1) | 1      return pdef
generate_prime_number(length=1024):

     """ Generate a prime           Args:

              length -- int -- length of the prime to generate, in         bits
return a prime

     """
```

```
p = 4

# keep generating while the primality test fail

while not is_prime(p, 128):

      p = generate_prime_candidate(length)

return pprint(generate_prime_number())
```

That’s it ! You now know how to generate big prime numbers !