Generating large primes

Saad Mneimneh

1 A simple paradigm

Generating a large prime is an important step in the RSA algorithm. We can generate a large prime by repeatedly selecting a large random integer and testing it for primality. We stop when a prime number is found. Therefore, assume that a large integer n is given, and choose a rational number $\alpha < 1$ to perform the following algorithm:

repeat
  generate a random integer $p \in [n^\alpha, n]$
until p is prime

There are two important aspects of the above algorithm. First, it is randomized in the sense that our random choices determine its running time (through the number of iterations). Second, it relies heavily on a method for primality testing, which will dominate the running time of a single iteration (making a random choice takes $O(\log n)$ time since it requires generating that many bits).

2 Analysis of number of iterations

Let us first determine the probability of choosing a prime in $[n^\alpha, n]$. We use a famous result in number theory:

Prime number theorem
Let $\pi(n)$ be the number of primes $\leq n$. Then

$$\pi(n) \sim \frac{n}{\ln n}$$

This is an asymptotic characterization, i.e. $\lim_{n\to\infty} \frac{\pi(n) \ln n}{n} = 1$. Since we are dealing with large values of n, making the approximation $\pi(n) \approx \frac{n}{\ln n}$ is valid. Therefore,

$$P(p \in [n^\alpha, n] \text{ is prime}) \approx \frac{\frac{n}{\ln n} - \frac{n^\alpha}{\alpha \ln n}}{n - n^\alpha} = \frac{n - n^\alpha/\alpha}{n - n^\alpha} \cdot \frac{1}{\ln n} = \frac{1 - n^{\alpha-1}/\alpha}{1 - n^{\alpha-1}} \cdot \frac{1}{\ln n}$$

Since $\alpha < 1$, $n^{\alpha-1} \approx 0$ for large values of n, and hence

$$P(p \in [n^\alpha, n] \text{ is prime}) \approx \frac{1}{\ln n}$$

Let $p_1, p_2, \ldots, p_m$ be a sequence of randomly generated numbers such that p_m is the first prime. Then m has a geometric distribution and $E[m] = \ln n$. Therefore, the expected running time in terms of iterations is $O(\log n)$. Note that this is an expected running time, not a deterministic one. Yet, we can show that the running time is $O(f(n) \log n)$ with high probability for every increasing function $f(n)$. The probability that a prime number is not found in $f(n) \ln n$ trials is

$$\Big(1 - \frac{1}{\ln n}\Big)^{f(n) \ln n} = \Big[\Big(1 - \frac{1}{\ln n}\Big)^{\ln n}\Big]^{f(n)} \sim e^{-f(n)} \to 0$$

3 Primality testing

Given a positive integer n, we would like to test if it is prime. Here's a trivial algorithm for this task based on finding a divisor for n:

```
for i ← 2 to n
    if n ≡ 0 (mod i)
      then return false
return true
```

The running time of this algorithm is $O(n)$ which is exponential in the size of n ($\log n$ bits). A simple improvement can be achieve by observing that if n is composite, it must have a divisor $\leq \sqrt{n}$. This is because a composite number is the product of two smaller numbers; therefore, $n = ab$ where $a \leq b < n$. Hence, $a \leq \sqrt{n}$.

```
for i ← 2 to ⌊√n⌋
    if n ≡ 0 (mod i)
      then return false
return true
```

But the running time is still exponential in the size of n. To obtain a substantial improvement, we rely on another famous result in number theory:

Fermat's little theorem
If n is prime and $1 \leq a < n$, then $a^{n-1} \equiv 1 (\mathrm{mod} n)$.

Therefore, if for some $1 \leq a < n$, $a^{n-1} \not\equiv 1 (\mathrm{mod} n)$, then a is a *witness* for the compositeness of n. However, if $a^{n-1} \equiv 1 (\mathrm{mod} n)$, Fermat's little theorem does not assert the primality of n. For example, $2^{341-1} \equiv 1 (\mathrm{mod}\, 341)$; however, $341 = 11 \cdot 31$ is not prime. Fermat's theorem can be strengthened to show that n is prime if every $1 \leq a < n$ satisfies $a^{n-1} \equiv 1 (\mathrm{mod} b)$, but we are not interested in an algorithm that tests every $1 \leq a < n$ for obvious reasons. One idea is to test few values for a randomly chosen in $[1, n-1]$.

for $i \leftarrow 1$ to k
    choose a random $a \in [1, n-1]$
    if $a^{n-1} \not\equiv 1 \pmod{n}$
      then return false
return true

We will appreciate the effectiveness of this test by proving the following fact: Let $1 \leq a < n$ be such that $\gcd(n, a) = 1$ (this also means that a has a multiplicative inverse modulo n). If $a^{n-1} \not\equiv 1 (\mathrm{mod}\, n)$, then the same is true for at least $(n-1)/2$ values in $[1, n-1]$. Let $b^{n-1} \equiv 1 (\mathrm{mod} n)$, then

$$(ab)^{n-1} \equiv a^{n-1} b^{n-1} \equiv a^{n-1} \not\equiv 1 (\mathrm{mod} n)$$

Furthermore, if $ab \equiv ac (\mathrm{mod} n)$, then $b = c$ (simply multiply each side by the multiplicative inverse of a). We have just proved that every integer $b \in [1, n-1]$ that passes the Fermat test is associated with a unique integer $ab \bmod n$ that fails the test. Therefore, at least half the integers in $[1, n-1]$ fail the test. The probability of not choosing one of them is $\frac{1}{2^k}$ which is minuscule if $k = 50$ or $k = 100$.

But what if n is composite but $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$ for every $a \in [1, n-1]$ that is relatively prime to n? Such n is called a Carmichael number. Carmichael numbers are rare, the smallest one is $561 = 3 \cdot 11 \cdot 17$. The Fermat test above can be modified to make at least $(n-1)/2$ values in $[1, n-1]$ fail the test for Carmichael numbers. This modification is know as the Miller-Rabin algorithm for primality testing. Their idea is to first compute $a^u \bmod n$ instead of $a^{n-1} \bmod n$, where $n - 1 = 2^t u$ and u is odd. Given a^u, a^{n-1} is then computed through the following sequence:

$$a^u = a^{2^0 u}, a^{2^1 u}, a^{2^2 u}, \ldots, a^{2^t u} = a^{n-1}$$

Each term can be obtained by squaring the previous one (all modulo n of course). This idea is based on the following observation: If n is prime, then either $a^u \equiv 1$ or some term in the above sequence is $\equiv -1$. Here's a proof:

$$x^2 \equiv 1 (\mathrm{mod} n) \Rightarrow x^2 - 1 \equiv 0 (\mathrm{mod} n) \Rightarrow (x-1)(x+1) \equiv 0 (\mathrm{mod} n)$$

Therefore, n divides either $(x-1)$ or $(x+1)$. The only way this can happen is for $x \equiv 1 (\mathrm{mod} n)$ or $x \equiv -1 (\mathrm{mod} n)$. In other words, there is no root of 1 that is not 1 or -1. Consequently, since $a^{n-1} \equiv 1$, either $a^u \equiv 1$ or some term in the sequence is $\equiv -1$. Therefore, if $a^{2^i u} \not\equiv \pm 1$ and $a^{2^{i+1}} \equiv 1$, n must be composite.

For example, $561 - 1 = 2^4 35$. For $a = 7$, $a^{35} \equiv 241 (\mathrm{mod}\, 561)$. This will give the following sequence:

$$241 \quad 298 \quad 166 \quad 67 \quad 1$$

Therefore, $67^2 \equiv 1 (\mathrm{mod}\ 561)$, an indication that 561 cannot be prime. Some number theory (that we omit) tells us that at least $(n-1)/2$ values for $a \in [1, n-1]$ must reveal such an instance (a non-primitive root of 1).

Miller-Rabin
let $n-1 = 2^t u$ (u odd)
for $i \leftarrow 1$ to k
    choose a random $a \in [1, n-1]$
    $x_0 \leftarrow a^u \bmod n$
    for $j \leftarrow 1$ to t
        $x_j = x_{j-1}^2 \bmod n$
        if $x_j = 1$ and $x_{j-1} \not\equiv \pm 1$
          then return false
    if $x_t \neq 1$
      then return false
return true

Since $t = O(\log n)$, the running time of this algorithm is dominated by the computation of $a^u \bmod n$, which at the surface, requires $O(n)$ multiplications. The next section illustrates how this can be computed in $O(\log n)$ time.

4 Repeated squaring

There is a recursive algorithm to compute a^u using only $O(\log u)$ multiplications.

$$a^u = \begin{cases} 1 & u = 0 \\ (a^{u/2})^2 & u \text{ even} \\ a \cdot a^{u-1} & u \text{ odd} \end{cases}$$

Consider the sequence $\{u\}$ given by the values of u over the entire execution of this algorithm:

$$u_0, u_1, \ldots, u_m = 0$$

where $u_0 = u$ and either $u_i = u_{i-1}/2$ or $u_i = u_{i-1} - 1$.

Consider the sequence $\{b\}$ obtained from $\{u\}$ by dropping the odd terms:

$$b_0, b_1, \ldots, b_r = 0$$

where $b_0 \leq u$ and $b_i \leq b_{i-1}/2$, and compare this sequence to the sequence $\{c\}$:

$$c_0 = u, c_1 = u/2, c_2 = u/4, \ldots, c_k$$

where $1 \leq c_k < 2$. It is trivial to show that $b_i \leq c_i$ and, therefore, $b_k = 0$ if it exists, hence $r \leq k$. Since $m \leq 2r + 1$, $m \leq 2k + 1$.

$$c_k 2^k = u \Rightarrow k = \log u - \log c_k$$

Therefore, $k \leq \lfloor \log u \rfloor$ and $m \leq 2\lfloor \log u \rfloor + 1$ which shows that the algorithm requires $O(\log u)$ multiplications. However, to be fair we must make sure that all intermediate numbers do not exceed n; otherwise, multiplications cannot be considered to be comparable in complexity. This can be achieved by an easy fix since what we need eventually is $a^u \bmod n$ anyway.

$$a^u \bmod n = \begin{cases} 1 & u = 0 \\ (a^{u/2})^2 \bmod n & u \text{ even} \\ a \cdot a^{u-1} \bmod n & u \text{ odd} \end{cases}$$