University of Maryland
CMSC456 — Introduction to Cryptography
Professor Jonathan Katz

Lecture 26

1 Generating Primes and Testing Primality

We have seen already a number of examples in which it was necessary to generate large prime numbers. Examples include the RSA cryptosystem (in which it is necessary to generate *two*, random primes and multiply them to obtain a modulus) and perfect message authentication (in which generating one prime p — and not necessarily random — was sufficient). However, we have not yet mentioned whether it is possible to generate such primes *efficiently*. We explore this issue here. (We only give a high-level overview of this topic, and refer the interested reader to [1] for a very readable and more in-depth treatment of the issues here.)

Assume we have some algorithm PrimalityTest which takes as input an integer p and returns "prime" if p is prime and "composite" otherwise (we will discuss examples of such algorithms below). This does not necessarily enable us to generate large (random) primes unless we can say something about the distribution of primes. Let $\pi(x)$ denote the number of primes less than or equal to x. It is known (informally) that $\pi(x) = \frac{x}{\ln x} \approx \frac{x}{\log_2 x}$. We can generate a random k-bit prime using the following algorithm:

```
GeneratePrime($1^k$)
 Repeat until successful:
  Choose random $p \leftarrow \{0,1\}^k$
  If PrimalityTest($p$) outputs "prime" then output $p$ and exit
```

How many different p will we need to test until we find one which is prime? Well, since $\pi(x) \approx \frac{x}{\log_2 x}$, the fraction of primes is $(x/\log_2 x)/x = 1/\log_2 x$ and hence the fraction of k-bit integers which are prime is roughly $1/k$. So we will need to test — on average — k numbers until we find one which is prime. Thus, assuming PrimalityTest runs in polynomial time, the entire GeneratePrime algorithm runs in polynomial time.

1.1 Primality Testing

To complete our algorithm for generating primes, we need to specify a PrimalityTest algorithm. Early in the semester, Prof. Gasarch gave a guest lecture discussing a (very recent) *deterministic* algorithm for testing primality. Unfortunately, this algorithm is slow; in practice, faster *randomized* algorithms are used. As we will see, however, these randomized algorithms have a small probability of returning the wrong answer.

First try. The first algorithm we consider will not completely solve our problem, but will be a good initial step. Before we present this algorithm, let's step back and see what properties we will want our PrimalityTest algorithm to have. Optimally (once we are willing to accept a small probability of error) we would like that for *all* primes, the probability that

PrimalityTest returns "prime" should be high; conversely, for *all* composite numbers, the probability that PrimalityTest returns "composite" should be high. Consider the following algorithm:

```
PrimalityTest($p$)
 for $i = 1$ to $T$    pick random $a \in \{1, \ldots, p-1\}$
   if $\gcd(a, p) \neq 1$ return "composite" and exit
   if $a^{p-1} \neq 1 \bmod p$, return "composite" and exit
 if $p$ has passed all the above tests, return "prime"
```

Note that if T is a constant (or, in general, polynomial in $|p|$) the above algorithm runs in polynomial time. We may note the following features of this algorithm:

Claim *If p is prime, the above algorithm always returns "prime".*

Proof If p is prime, then for all $a \in \mathbb{Z}_p^*$ we have $a^{p-1} = 1 \bmod p$. Thus, p passes every iteration of the above algorithm and therefore the algorithm always outputs "prime". ■

What can we say when p is prime? Unfortunately, there are composite numbers p for which $a^{p-1} = 1 \bmod p$ for all $a \in \mathbb{Z}_p^*$; these numbers are called *Carmichael numbers*. For such numbers, PrimalityTest will output the (incorrect) answer "prime" unless it happens to pick an a for which $\gcd(a, p) \neq 1$; this will happen with low probability. For non-Carmichael numbers, we can claim the following without proof:

Claim *If p is composite and there exists at least one $a \in \mathbb{Z}_p^*$ for which $a^{p-1} \neq 1 \bmod p$ (i.e., p is not a Carmichael number), then each iteration of* PrimalityTest(p) *outputs "composite" with probability at least* $1/2$.

Thus, iterating T times (as done above) ensures that the probability that PrimalityTest will return the wrong answer in this case is at most $1/2^T$. This probability can be made as small as desired by increasing T.

In summary, the above algorithm has the following properties:

1. If p is prime, then PrimalityTest always returns "prime".
2. If p is a Carmichael number, then PrimalityTest (almost) always returns the incorrect answer "prime".
3. If p is composite and not a Carmichael number, then PrimalityTest returns the correct answer "composite" except with probability at most $1/2^T$.

As a practical matter, Carmichael numbers are relatively rare (although there are infinitely many). So, if trying to generate a random prime (using GeneratePrime as discussed above), the probability of picking a Carmichael number in the first place is low. Still, we would prefer an algorithm with low failure probability for *all* composite numbers. The next algorithm will achieve this.

Improved algorithm. The next algorithm — known as the Miller-Rabin algorithm — will have the properties we desire.

```
MillerRabin($p$)
 Write $p-1$ as $2^e r$ for odd $r$
 for $i = 1$ to $T$
  pick random $a \in \{1, \ldots, p-1\}$
  if $\gcd(a, p) \neq 1$ return "composite" and exit
  if $a^r \neq 1 \bmod p$ and $\nexists k \in [0, e-1]$ such that $a^{2^k r} = -1 \bmod p$
   return "composite" and exit
 output "prime" and exit
```

First, it is important to note that this algorithm runs in polynomial time. Step 1 does *not* involve factoring $(p-1)$; all we are doing is pulling out factors of 2, which is easy. Secondly, checking whether there exists a $k \in [0, e-1]$ such that $a^{2^k r} = -1 \bmod p$ requires checking at most e such values; but e is at most $\log_2 p$ so this is still polynomial time (of course, computing the exponentiation can also be done in polynomial time as the exponent never gets bigger than $p-1$).

Claim *If p passes an iteration of the Miller-Rabin test for a particular value of a, then $a^{p-1} = 1 \bmod p$ and hence p would pass an iteration of* PrimalityTest *for that same a.*

Proof Note that $a^{p-1} = a^{2^e r} \bmod p$. If p passes an iteration of the Miller-Rabin test then either $a^r = 1 \bmod p$ or else there exists a $k \in [0, e-1]$ such that $a^{2^k r} = -1 \bmod p$. In the first case, we have $a^{2^e r} = (a^r)^{2^e} = 1^{2^e} = 1 \bmod p$. In the second case, we have $a^{2^e r} = (a^{2^k r})^{2^{e-k}} = (-1)^{2^{e-k}} \bmod p$. Since $e > k$, we can write this as $((-1)^2)^{2^{e-k-1}} = 1^{2^{e-k-1}} = 1 \bmod p$. ■

In fact, as was the case with PrimalityTest, the Miller-Rabin test always outputs "prime" when given a prime as input.

Claim *If p is prime, the Miller-Rabin algorithm always returns "prime".*

Proof We show that for all $a \in \{1, \ldots, p-1\} = \mathbb{Z}_p^*$, an iteration of the Miller-Rabin test using this value of a succeeds. Fix a. Consider the sequence $(x_0 = a^r, x_1 = a^{2r}, x_2 = a^{2^2 r}, \ldots, x_e = a^{2^e r})$ (where all values are taken modulo p). Since $2^e r = p-1$, we know that the last term in this sequence is 1. Now, if $a^r = 1 \bmod p$ we are done, since this implies that the iteration succeeds. Otherwise, there must be a rightmost element of the sequence (say, x_i) which is *not* 1. But since $x_{i+1} = x_i^2 \bmod p$, and since (by assumption) $x_{i+1} = 1$, we must have $x_i = \pm 1$ (since p is prime). Thus, $x_i = -1 \bmod p$ and, since $x_i = a^{2^i r}$, this implies that the iteration succeeds. ■

We state the following without proof.

Claim *If p is composite — even if p is Carmichael — then each iteration of the Miller-Rabin test outputs "composite" with probability at least* $3/4$.

Thus, in addition to fixing the problem of falsely identifying Carmichael numbers, the Miller-Rabin test also achieves a lower failure probability per iteration!

As above, by adjusting T we can achieve as low an error probability as we desire.

References

[1] L.N. Childs. *A Concrete Introduction to Higher Algebra (2nd Edition)*. Springer, 1995.