CSCI 5454 Lecture 27, Primality Testing
Tracy Babb, DJ Sutton
April 16, 2012

---

1 Introduction

A prime numbers is an integer that is not evenly divisible by any other integer except one. Determining whether a given number is prime has been a pure mathematical curiosity ever since Euclid, but the ability to answer this question quickly for very large numbers has become much more important in the last fifty years due to modern public key cryptography. Public key cryptography schemes are usually based on NP-Complete problems, that is, problems for which it is hard to find a solution but easy to check a solution. If we make the problem hard enough, then it is likely that the person who generated the problem-solution pair is the only person in the world who has the solution to that particular problem. This creates a One-way function that can be used for encryption or authentication. Specifically, this can be used to securely encrypt a document because only someone with the solution to the problem can decrypt it, or securely sign a document because only someone with the solution could have created a particular signature.

2 A Little Number Theory

In the discussion of cryptography and primality testing, we will be drawing on several concepts and terms from number theory and abstract algebra.

2.1 Modular Arithmetic

Every grade school child is introduced to modular arithmetic when they learn how to tell time. For example, if it is 11:00 now, what time will it be in 2 hours?. Coming up with the answer of 1:00 requires performing addition modulo 12 [1]. That is, when the number hits 12, it wraps around and starts over. The number where this wrapping occurs is called the modulus. Modular arithmetic involves the the study of the normal arithmetic operations; addition, subtraction, multiplication, division (multiplicative inverse), and exponentiation; in a system where this wrapping occurs.

Mathematically, modular arithmetic can be defined as follows. For integers a, n, q and r if $a = qn + r$ then we write $a \equiv r \pmod{n}$ or "a is congruent to r mod n". From this equation, we see that asking for $a \mod n$ is equivalent to asking for the remainder when a is divided by n. Additionally, we note that since the quotient q is not considered, modular

[1] A subtle but important difference between clock arithmetic and arithmetic modulo 12 is that standard notation uses 12:00 instead of 0:00 to correspond to 0. Typically in modular arithmetic the numbers we deal with are always strictly less than the modulus

congruence is symmetric and transitive. That is, if $a \equiv r \pmod{n}$ then $r \equiv a \pmod{n}$,and if $a \equiv r \pmod{n}$ and $r \equiv b \pmod{n}$ then $a \equiv b \pmod{n}$. If $a \equiv b \pmod{n}$, this also implies $a - b \equiv 0 \pmod{n}$ or $(a - b)$ is divisible by n

Ideas in modular arithmetic usually involve the concept of divisibility and primality. If $n = qa$ for integers n, q, a, then we say $a \mid n$ or a divides n or n is divisible by a, or $n \equiv 0 \pmod{a}$. This implies that if $n \equiv 0 \pmod{a}$, then there exists an integer q such that $a/n = q$. If $a \nmid n$ for all integers $a \in [2, n-1]$, then we say n is prime. If there are two integers p and q such that $a \nmid p$ or $a \nmid q$ for all integers $a \in [2, n-1]$, then we say that p and q are relatively prime or coprime, which is equivalent to saying they have no common factors.

2.2 Computation and Number Theory Algorithms

In most models of computation, we assume that integer arithmetic operations are atomic and take constant time. However, when dealing with very large integers, we must refine this assumption. In reality, our arithmetic operations take time proportional to the number of bits in the binary representation of an integer. It might help here to think of arithmetic operations on the integer n as operation on an array of bits of size $\lceil \lg(n) \rceil$. Since the number of bits in an integer n is $O(\lg(n))$, this becomes the new standard for what it means to have an efficient algorithm. An algorithm with running time $O(\lg(n))$ is considered a linear time algorithm because it is linear in the number of bits in n. Any algorithm that runs in $O(n)$ time is exponential in the number of bits. In fact, any algorithm with running time $O(n^k)$ for a positive constant k is exponential in the number of bits in n and likely to be prohibitively slow for large n.

3 A Little Cryptography

We will motivate our interest in prime numbers with a demonstration of the RSA encryption scheme. [7] Here we meet the familiar Alice and Bob, two parties who wish to communicate over an unsecured channel. Specifically, Alice wishes to send a message m to Bob without revealing the message to Eve, a third party who can listen to any communication between Alice and Bob. Alice and Bob can communicate securely using the following algorithm

1. Bob picks two large prime numbers p and q and computes $n = pq$.
2. Bob computes $\phi(n) = (p-1)(q-1)$, $\phi(n)$ is Euler's totient function. [2]
3. Bob picks an integer e relatively prime to $\phi(n)$.
4. Bob computes d such that $de \equiv 1 \pmod{\phi)(n)}$, which can be done efficiently using the extended Euclidean algorithm. [4]
5. Bob sends (n, e) to Alice and keeps p, q, d secret

6. Alice computes $c = m^e \pmod{n}$ and sends c to Bob [5]

7. Bob computes $m = c^d \pmod{n}$ to retrieve the original message

All Carol can see is n,e and c. It is therefore impossible for her to retrieve m unless she can invert $c = m^e \pmod{n}$, which is believed to be exponentially hard, or she can compute d, which believed to be as hard as factoring n. In order to make factoring n very hard, Bob must choose large prime numbers for p and q, and he must be fairly certain they are prime. In other words, Bob needs a good primality test.

4 Requirements for Primality Testing

It turns out that many number theory problems that are useful for cryptography involve primes. However, to make these problems useful, we must be able to satisfy a few key requirements.

1. In order to make the problems sufficiently hard, we must choose very large primes. Typical modern key generation algorithms will use primes with 2048 or 4096 bits, which is around six hundred to twelve hundred decimal digits.

2. In order to make it easier to create these problems than it is to solve the problems, we must be able to find these primes quickly, which means that enumerating all the prime numbers is not an option. Instead, the accepted approach is to generate random numbers with the required number of bits until a prime is found [2], which means that we must be able to test for primality quickly.

3. In order to ensure that the problem truly is hard to solve, we want to make sure that the number we find is definitely prime.

Together, these tight requirements motivate the study of efficient algorithms for primality testing.

5 Naïve method

If you wanted to find out if a small number like 91 was prime, what would you do? For most people, the answer to this question is a process called trial division. Is it divisible by 2? Is it divisible by 3? Is it divisible by 5?... etc. How high do you have to check? We can answer this last question by noting that if n is composite, then it must have a least one factor less than or equal to $\sqrt{n}$, which can be proved as follows. If there exist positive integers q, r with $q \leq r$ such that $q \cdot r = n$ and $q, r \neq n$, then we must have $q \geq 2$. Next, rewrite

[2]For numbers around N, approximately $\frac{1}{\ln(N)}$ of numbers that size are prime. This means that the expected number of tries before finding a prime is $O(\ln(n))$, which is on the order of the number of bits in N

$q = \alpha\sqrt{n}, r = \beta\sqrt{n}$ for some constants α, β with $\alpha \leq \beta$. Then, $n = \alpha\beta\sqrt{n}^2$ and so we must have $\alpha\beta = 1$. Since we $\alpha \leq \beta$, we conclude $\alpha \leq 1$ This implies the following algorithm for doing the minimum number of trial divisions necessary to determine if a number is prime. The algorithm exhaustively checks all integers between 2 and $\sqrt{n}$ to see if they divide n via the modulo operator and returns "composite" if it finds any such integer, or "prime" if it passes $i = \sqrt{n}$ without finding a factor

function TRIAL DIVISION(n)
  for $i = 2$ to $\lceil\sqrt{n}\rceil$ **do**
    if $n \equiv 0 \pmod{i}$ **then**
      return composite
    end if
  end for
  return prime
end function

This algorithm only needs to allocate the counter variable i ,and so used constant space. This algorithm will run in $O(\sqrt{n})$ time, because at worst it will iterate through all integers from 2 to $\sqrt{n}$. However, as previously stated, this is considered an exponential running time for large integers, and thus is unacceptable for an efficient primality test.

As an example, an execution trace for this algorithm with $n = 91$ would look as follows

i	$91 \pmod{i}$
2	1
3	1
4	3
5	1
6	1
7	$0 \rightarrow$ Return composite

6 Fermat's Little Theorem

Fermat's little theorem is essential to the idea behind the Miller-Rabin primality test. Fermat's little theorem says that for a prime number p and $a \in \mathbb{N}$ that

$$a^p - a = 0 \pmod{p} \tag{1}$$

Furthermore, if a is not divisible by p, then

$$a^{p-1} - 1 = 0 \pmod{p} \tag{2}$$

The first result here can be shown by induction, and then the second result follows directly from the first result. To show the first result, take $a = 1$. Then

$$1^p - 1 = 0 \pmod{p} \tag{3}$$

Now assume the result holds for some a and show it also holds for $a+1$. That is, show

$$(a+1)^p - (a+1) = 0 \pmod{p} \tag{4}$$

Using the binomial theorem, note that

$$(a+1)^p - (a^p+1) = \sum_{i=1}^{p-1} \binom{p}{i} a^i \tag{5}$$

Also note that $\binom{p}{i}$ is divisible by p $\forall i \in \{1, 2, ..., p-1\}$. Therefore, the quantity on the left hand side of the equation is divisible by p. And therefore, the quantity

$$((a+1)^p - (a^p+1)) + (a^p - a) = (a+1)^p - (a+1) \tag{6}$$

is also divisible by p since it is just the sum of the quantity that was just shown to be divisible by p and the quantity assumed to be divisible by p in the induction hypothesis. So the result also holds for $a+1$ and therefore holds for all $a \in \mathbb{N}$ if p is prime. The second result is easy to show from the first result. From the first result we have

$$\frac{a^p - a}{p} = c \tag{7}$$

where c is a constant integer. This also gives

$$\frac{a(a^{p-1} - 1)}{p} = c \tag{8}$$

So either a or $(a^{p-1} - 1)$ must be divisible by p. So if a is not divisible by p, then we have the second result. [10]

$$a^{p-1} - 1 = 0 \pmod{p} \tag{9}$$

6.1 Fermat Test for Primality and Pseudoprimes

Fermat's little theorem makes a statement that is provably true if p is prime, or more generally if a and p are relatively prime. However, if we wish to use it to find prime numbers, we must use the contrapositive. That is, for any integers a and p, if $a^{p-1} \neq 1$(mod p) then p is definitely not prime. On the other hand, if $a^{p-1} = 1$(mod p), then a might be prime. This leads to a randomized test for primality which computes the value of a^{p-1}(mod p) for many different a. If this is ever different than 1, then the number is certainly not prime.

function FERMAT TEST(n,k)
  for i=1 through k **do**
    pick a random integer a such that $1 \leq a \leq n-1$
    $x \leftarrow a^{n-1} \pmod{n}$
    if $x \neq 1$ **then**
      return composite

end if
end for
return probably prime
end function

As an example, an execution trace for this algorithm with $n = 9$ might look as follows

a	a^{n-1}	$a^{n-1} \pmod{n}$
8	16777216	1
2	256	$4 \rightarrow$ Return composite

As we test more values for a, and continue to find that $a^{p-1} = 1 \pmod{p}$ it becomes increasingly likely that p is prime. However, it is not possible in general to guarantee that this is true with a reasonably small number different values for a. Indeed, there is a known set of numbers known as Fermat pseudoprimes or Carmichael numbers [3] for which $a^{p-1} = 1 \pmod{p}$ is true for all integers a relatively prime to p. The first such number is 561, but it has also been proven that there are an infinite number of Fermat pseudoprimes. [1]

The Fermat primality test gives some of the important things we need in a good primality test. Since exponentiation mod p can be performed with $O(lg(p))$ operations, the test can be performed quickly even for large p. If the test passes for several independent values of a, it could be a good indication that p is truly prime. However, the existence of Fermat pseudoprimes means that this cannot be a reliable method for verifying that a number is prime with arbitrarily high probability. The general framework and procedure of the test are instead used to create other, more reliable tests, such as the Miller-Rabin algorithm.

7 Miller-Rabin Algorithm

For the Miller-Rabin primality test [6], we start by noting that any odd number n can be written in the form

$$n = 2^r s + 1 \tag{10}$$

where s is an odd integer and r is also an integer. It is easy to see that this has a unique solution for r and s from the fundamental theorem of arithmetic (recall every number has it's own unique prime factorization). Since n is odd, $n - 1$ is even, and therefore the prime factorization for $n - 1$ has a specific number of 2s and some other set of prime numbers whose product will be odd. To find the values of r and s we can simply divide $n - 1$ by 2 until we have an odd number. The number of divisions by 2 will give the value of r, and the resulting odd number will give the value of s.
Next, choose a random integer a such that $1 \leq a \leq n-1$. One of the following two conditions must hold if n is prime.

$$a^s = 1 \pmod{n} \tag{11}$$

or

$$a^{2js} = -1 \pmod{n} \tag{12}$$

for some j such that $0 \leq j \leq r-1$. There are a few interesting properties of the conditions described. First, if n is prime then one of the two conditions given will hold for all numbers a such that a such that $1 \leq a \leq n-1$. So we can check the conditions for several numbers a. If one of the two conditions hold for all a that we check, the number n is most likely prime. If the conditions fail at some point, n must be composite. Monier and Rabin have shown that a composite number will pass the test for no more than 1/4 of the possible numbers a. Therefore,

$$P(\{n \text{ found prime after 1 iteration}\}|\{n \text{ composite}\}) \leq \frac{1}{4} \tag{13}$$

First, note that the Miller-Rabin primality test can technically be turned into a deterministic test by performing the test for $\lceil n/4 \rceil + 1$ numbers a. If the test is passed for all numbers a then n must be prime. However, for any numbers that are large enough for us to care about testing the primality of, $n/4$ will become large very quickly and the time to test all $\lceil n/4 \rceil + 1$ numbers will become impractical. Nevertheless, since the probability of the test being passed for a given number a given that n is a composite number is bounded by 1/4, we can quickly bound the probability of some composite number n passing the test after m different iterations, where A is the event that the test is passed for all m iterations.

$$P(\{n \text{ found prime after } m \text{ independent iterations}\}|\{n \text{ is composite}\}) < \frac{1}{4^m} = \frac{1}{2^{2m}} \tag{14}$$

While

$$P(\{\text{n found prime after m independent iterations}\}|\{\text{n is prime}\}) = 1 \tag{15}$$

So if we claim that a number n is prime after m iterations of the algorithm, we are wrong with probability less than $1/2^{2m}$, which becomes very small very quickly. For just 10 iterations we have $1/2^{20}$. However, note that one of the problems with these probabilities is that, as small as they are, we could have a situation where, for the application we are considering, we must be correct in our answer of whether or not the number is prime. Contrasting this to something like manufacturing hockey sticks for the NHL, if only 90% of our sticks meet the rules on the size of a stick, it's easy to just measure them after we manufacture them and throw out the ones that don't fall within regulation. However, if we are setting up an RSA public key for sensitive information, and the numbers we produce that we think are prime are actually composite, then our product of supposed product of two primes is actually much easier to factor. So in this case, if we are only right 90% of the time, this is very bad. Fortunately, $1/2^{2m}$ decreases quickly enough that we don't have to test a lot of numbers a to get our probability small, but the point still remains that if we want to generate an RSA public key for very sensitive data then we really want to be sure that our numbers are in fact prime.

Note that while this is a probabilistic test, no composite numbers have been found that pass the test. Wolfram MathWorld states that if there are any composite numbers that actually pass the test, then they occur with such small probability that it is essentially negligible. Wolfram compares this probability to the probability of a hardware error on a computer performing the test. [13]

7.1 Pseudocode

The pseudocode for the Miller-Rabin algorithm is shown here.

function ISPRIME(n,k)
  find r and s such that $n-1=2^r s$
  for i=1 through k **do**
    pick a random integer a such that $1 \leq a \leq n-1$
    for j=1 through $r-1$ **do**
      $x = a^{2^j s}$
      if $x \pmod n = -1$ **then**
        do next loop
      end if
    end for
    $x = a^s$
    if abs(x)=1 **then**
      do next loop
    else
      return composite
    end if
  end for
  return probably prime
end function

7.2 Proof of Correctness

The proof of correctness for the Miller-Rabin algorithm comes from Fermat's little theorem. First we show a small result that is necessary for the proof. Assume that, for some n prime and some x

$$x^2 - 1 = 0 \pmod n \tag{16}$$

Then by factoring the left hand side

$$(x+1)(x-1) = 0 \pmod n \tag{17}$$

Therefore, either (x+1) or (x-1) must be divisible by n. Recall for n prime

$$a^{n-1} - 1 = 0 \pmod n \tag{18}$$

as long as a is not divisible by n. Since the Miller-Rabin algorithm uses a such that $1 \leq a \leq n-1$ it is not possible for a to be divisible by n. Therefore, writing $n-1$ in the form $2^r s$ that we use for the Miller-Rabin algorithm, we have

$$a^{2^r s} - 1 = 0 \pmod n \tag{19}$$

Or

$$\left(a^{2^{r-1} s}\right)^2 - 1 = 0 \pmod n \tag{20}$$

So by the fact just shown ($x^2 - 1 = 0 \pmod{n}$ means either $x - 1$ or $x + 1$ is divisible by n), we have that either

$$a^{2^{r-1}s} - 1 \quad \text{or} \quad a^{2^{r-1}s} + 1 \tag{21}$$

must be divisible by n. If the first one holds, we can take the square root of $a^{2^{r-1}s} - 1$ and check to see if one of the two conditions hold. We can continue to take square roots until either $a^{2^{r-1}s} + 1$ is divisible by n (which is then an indication that n is prime, or until we have

$$a^{2^{r-r}s} \pm 1 = a^s \pm 1 = 0 \pmod{n} \tag{22}$$

So this shows that if n is prime, then either

$$a^s - 1 = 0 \pmod{n} \tag{23}$$

or

$$a^{2^j s} + 1 = 0 \pmod{n} \tag{24}$$

for some j such that $0 \leq j \leq r - 1$. So if we can find an a such that none of these conditions hold then we know that n is composite. Otherwise, n may be prime. [13]

7.3 Time and space Usage for Miller-Rabin

The first step of determining r, s such that $2^r + s = n - 1$ is done by performing r divisions by 2. Since $2^{\lg(n)} = n > n - 1$, we have $r < \lg(n)$ and so this part runs in $O(\lg(n))$ time. Next, we examine the inner for loop. This loop performs a constant number of operations and will be executed at most $r - 1$ times for a running time of $O(r - 1) = O(\lg(n))$ per execution of the outer loop. After the inner loop, an additional $O(1)$ operations are performed inside the outer loop. The outer loop is executed k times for a total outer loop running time of $k(O(\lg(n)) + O(1)) = O(k\lg(n))$. Therefore the total running time of the whole algorithm is $O(\lg(n)) + O(k\lg(n)) = O(k\lg(n))$. If we hold k constant, which holds the probability of a false positive constant as n increases, then the running time is strictly $O(\lg(n))$, which means that the Miller-Rabin test meets the requirement for efficiency.

In terms of space, the Miller-Rabin algorithm does not allocate any arrays and uses a constant number of temporary variables, so the space requirement is $O(1)$. Thus the algorithm is efficient in space as well.

7.4 Miller-Rabin Example

As an example, an execution trace of one iteration of the outer loop for this algorithm with $n = 561$ might look as follows

Pick $a = 2$
Compute $n - 1 = 2^r s$

r	$\frac{n-1}{2^r}$
0	560
1	280
2	140
3	70
4	35

Thus $560 = 2^4 \cdot 35$, so $r = 4, s = 35$
Now perform the inner loop:

j	$2^j s$	$a^{2^j \cdot s} \pmod n$
1	70	166
2	140	67
3	280	1

None of these are equal to 560, so move on to the final check
Compute $a^s \pmod n = 263$. Since $263 \neq 1$ and $263 \neq 560$, we conclude that 561 must be composite.

8 Variations

Note that the algorithms described here have been what are known as probabilistic tests. The probabilistic algorithms use properties that must hold for prime numbers and can hold for composite numbers. If a number is prime, it must pass the test since the properties must hold for a prime number. If a number is composite it will pass the test with some probability. There are also some algorithms that are known as determistic. These algorithms return with absolute certainty whether a number is prime or composite. The deterministic algorithms take much longer to run, which is why we are mainly interested in probabilistic algorithms. The best deterministic primality test is the elliptic curve primality proving. As of 2009, the largest certified prime using this technique was a 20562 digit number. [9]
The Lucas pseudoprime test is a probabilistic test and is used by Mathematica along with the Miller-Rabin test to test for primality of a number. [12]
The Lucas-Lehmer test is an efficient probabilistic test. [11]
The AKS primality test was the first deterministic algorithm that could run in polynomial time. Original running time was $O(ln^{12+\epsilon}p)$ and has since been reduced to $O(ln^{6+\epsilon}p)$. [8]

References

[1] W. R. Alford, Andrew Granville, and Carl Pomerance. There are infinitely many carmichael numbers. *The Annals of Mathematics*, 139(3):pp. 703–722, 1994.

[2] Jeffrey Bach, Eric; Shallit. *Algorithmic Number Theory (Vol I: Efficient Algorithms)*. The MIT Press, 1966.

[3] R. D. Carmichael. On composite numbers p which satisfy the fermat congruence $a^{P-1} \equiv 1 \bmod p$. *The American Mathematical Monthly*, 19(2):pp. 22–27, 1912.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. 31.2 Greatest Common Divisor pp. 933-938.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. 31.6 Powers of an element pp. 954-958.

[6] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128 – 138, 1980.

[7] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[8] Eric W. Weisstein. *AKS Primality Test* From MathWorld–A Wolfram Web Resource., April 2012.

[9] Eric W. Weisstein. *Elliptic Curve Primality Proving* From MathWorld–A Wolfram Web Resource., April 2012.

[10] Eric W. Weisstein. *Fermat's Little Theorem* From MathWorld–A Wolfram Web Resource., April 2012. http://mathworld.wolfram.com/FermatsLittleTheorem.html.

[11] Eric W. Weisstein. *Lucas-Lehmer Test* From MathWorld–A Wolfram Web Resource., April 2012.

[12] Eric W. Weisstein. *Lucas Pseudoprime* From MathWorld–A Wolfram Web Resource., April 2012.

[13] Eric W. Weisstein. *Rabin-Miller Strong Pseudoprime Test* From MathWorld–A Wolfram Web Resource., April 2012.