# Number Theory and Security in the Digital Age

Lola Thompson

Ross Program

July 21, 2010

# Introduction

"I have never done anything useful. No discovery of mine has made, or is likely to make, directly or indirectly, for good or ill, the least difference to the amenity of the world." -G. H. Hardy

# Introduction

For centuries, number theory was considered to be the most 'pure' form of mathematics - there were no practical applications, as far as anyone could tell.

However, in the latter half of the $20^{th}$ century, number theory became central to developments in digital security. Today, we will discuss just a few of its applications, including:

# Introduction

For centuries, number theory was considered to be the most 'pure' form of mathematics - there were no practical applications, as far as anyone could tell.

However, in the latter half of the $20^{th}$ century, number theory became central to developments in digital security. Today, we will discuss just a few of its applications, including:

- primality testing/proving

# Introduction

For centuries, number theory was considered to be the most 'pure' form of mathematics - there were no practical applications, as far as anyone could tell.

However, in the latter half of the $20^{th}$ century, number theory became central to developments in digital security. Today, we will discuss just a few of its applications, including:

- primality testing/proving
- public key cryptography

# Introduction

For centuries, number theory was considered to be the most 'pure' form of mathematics - there were no practical applications, as far as anyone could tell.

However, in the latter half of the $20^{th}$ century, number theory became central to developments in digital security. Today, we will discuss just a few of its applications, including:

- primality testing/proving
- public key cryptography
- credit card check digits

# Primality Testing

"The problem of distinguishing prime numbers from composite numbers, and of resolving the latter into their prime factors, is known to be one of the most important and useful in arithmetic... Nevertheless we must confess that all methods that have been proposed thus far are either restricted to very special cases or are so laborious that even for numbers that do not exceed the limits of tables constructed by estimable men, they try the patience of even the practiced calculator." -C. F. Gauss

# Primality Testing

It is easy to tell that 31 is prime and that 33 is not, but what about 60017?

**Fundamental Problem:** Given an integer $n$, determine whether it is prime or composite.

# A Naive Test

What is the most obvious way that you can think of to determine whether a positive integer $n$ is prime?

# A Naive Test

What is the most obvious way that you can think of to determine whether a positive integer $n$ is prime?

- Check all integers up to $\sqrt{n}$ to see if they divide $n$.

  **Example** To determine whether 131 is prime, we just need to check all of the integers up to $\sqrt{131}$ : $2, 3, 4, 5, 6, 7, 8, 9, 10, 11$.

  None of these divide 131, so it must be prime.

# Some Minor Improvements

We know that even numbers greater than 2 are not prime, so an improvement would be:

We know that integers $> 3$ that are congruent to 3 (mod 6) are not prime, so an improvement would be:

# Some Minor Improvements

We know that even numbers greater than 2 are not prime, so an improvement would be:

- Check all **odd** integers up to $\sqrt{n}$ to see if they divide $n$.

We know that integers $> 3$ that are congruent to 3 (mod 6) are not prime, so an improvement would be:

# Some Minor Improvements

We know that even numbers greater than 2 are not prime, so an improvement would be:

- Check all **odd** integers up to $\sqrt{n}$ to see if they divide $n$.

We know that integers $> 3$ that are congruent to 3 (mod 6) are not prime, so an improvement would be:

- Check all integers $\equiv \pm 1 \pmod 6$ up to $\sqrt{n}$ to see if they divide $n$.

  We've already reduced the number of computations to $\frac{1}{3}\sqrt{n}$ steps!

  Can we do any better?

# Two Theorems from Elementary Number Theory

**Theorem (Wilson)**

*If $p$ is prime then $(p-1)! \equiv -1 \pmod{p}$.*

**Theorem (F$\ell$T)**

*If $p$ is prime and $p \nmid a$ then $a^{p-1} \equiv 1 \pmod{p}$.*

Can we use these theorems to detect whether an integer is prime?

If so, how efficient are these as primality criteria?

# F$\ell$T as a Primality Test

The repeated squaring algorithm is quite efficient:

Example: Let's check for $a = 2, p = 91$.

$$2^{90} = (2((2 \cdot (2^5)^2)^2)^2)^2$$

$2^5 \equiv 32\ (91)$ $\quad$ $2^{10} \equiv 23\ (91)$ $\quad$ $2^{11} \equiv -45\ (91)$

$2^{22} \equiv 23\ (91)$ $\quad$ $2^{44} \equiv -17\ (91)$ $\quad$ $2^{45} \equiv -34\ (91)$

$2^{90} \equiv 64\ (91)$

Do you notice anything strange about this last congruence?

# The Failure of F$\ell$T

Notice that the last congruence seems to violate F$\ell$T! Thus, the number 91 must not have been prime in the first place.

How efficient was this algorithm?

# The Failure of F$\ell$T

Notice that the last congruence seems to violate F$\ell$T! Thus, the number 91 must not have been prime in the first place.

How efficient was this algorithm?

- The whole process took 7 steps, which is proportional to $\log_2(n)$ when $n = 91$.

# The Failure of F$\ell$T

Notice that the last congruence seems to violate F$\ell$T! Thus, the number 91 must not have been prime in the first place.

How efficient was this algorithm?

- The whole process took 7 steps, which is proportional to $\log_2(n)$ when $n = 91$.
- For contrast, notice that $\frac{1}{3}\sqrt{n} \approx 3.12$, the speed of our (improved) naive algorithm.

# The Failure of F$\ell$T

Notice that the last congruence seems to violate F$\ell$T! Thus, the number 91 must not have been prime in the first place.

How efficient was this algorithm?

- The whole process took 7 steps, which is proportional to $\log_2(n)$ when $n = 91$.
- For contrast, notice that $\frac{1}{3}\sqrt{n} \approx 3.12$, the speed of our (improved) naive algorithm.
- For small values of $n$, the naive algorithm will be a bit quicker. However, when $n$ is big, the F$\ell$T algorithm will be much faster.

# Another F$\ell$T Example

Notice that $3^{90} \equiv 1 \pmod{91}$ seems to obey F$\ell$T, but we just concluded on the previous slide that 91 is not prime. (In fact, $91 = 7 \times 13$)

**Conclusion:** If F$\ell$T fails then $n$ must be composite. But, if it seems to work, then $n$ could be either prime or composite. In this case, we cannot conclude anything!

# Wilson's Theorem as a Primality Test

We just saw that the converse of F$\ell$T is false in general.

However, the converse of Wilson's Theorem is true:

**Theorem**

*If* $(n-1)! \equiv -1 \pmod{n}$ *and* $n > 1$ *then* $n$ *is prime.*

Unfortunately, we have no efficient way to check the Wilson congruence - the naive method of multiplying $n-1$ numbers together would take $n-1$ steps. This is much slower than all of the algorithms that we have discussed so far.

# How Useful Are These Approaches?

- Wilson's Theorem is not useful at all as a primality test. It is too slow to be helpful.

# How Useful Are These Approaches?

- Wilson's Theorem is not useful at all as a primality test. It is too slow to be helpful.

- F$\ell$T is useful if we employ it in the following manner:

  If we pick $a$ randomly, the chance that $a^{p-1} \equiv 1 \pmod{p}$ but $p$ is not prime is $\leq 25\%$.

  If we repeat this process 50 times and always find that the F$\ell$T congruence is satisfied, then the chance that $p$ is not prime is less than $7.9 \times 10^{-29}\%$.

  To put this into context, you are roughly $10^{25}$ times more likely to be struck by lightning this year than you are to incorrectly conclude that $p$ is prime from this test!

  So, we can be reasonably certain that $p$ is prime.

# Exercise

Using F$\ell$T, determine whether or not 60017 is prime. (Note: For this exercise, we will say that 60017 is prime if you are at least 98% certain that it is).

# Primality Testing vs. Primality Proving

As we just saw, F$\ell$T is a good test if we want to convince ourselves that an integer $p$ is prime. However, it doesn't actually prove anything - there is always a (small) chance that $p$ is composite.

If we wanted to **prove** that $p$ is prime then we would need to use a **primality proving algorithm**.

Primality proving algorithms are usually slower than primality tests. So, if we want to be as efficient as possible, we would first use a primality test (like F$\ell$T) to be relatively certain that $p$ is not composite before using a primality proving algorithm.

# Another Approach

## Theorem (Lucas)

*Suppose that $n > 1$ and $a$ are integers with*

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

*and*

$$a^{(n-1)/q} \not\equiv 1 \pmod{n}$$

*for all primes $q \mid (n-1)$. Then $n$ is prime.*

(This follows from the fact that $U_p$ is cyclic, so it must have an element of order $\varphi(p) = p - 1$, but $U_n$ will never have an element of order $n - 1$ if $n$ is composite, since $\varphi(n) < n - 1$ in that case.)

# Lucas' Theorem as a Primality Test

In order to use Lucas' Theorem to prove that an integer $n$ is prime, there are two things that we must do:

# Lucas' Theorem as a Primality Test

In order to use Lucas' Theorem to prove that an integer $n$ is prime, there are two things that we must do:

- Find an element $a$ with order $n - 1$.

# Lucas' Theorem as a Primality Test

In order to use Lucas' Theorem to prove that an integer $n$ is prime, there are two things that we must do:

- Find an element $a$ with order $n-1$.
- Find the prime factorization for $n-1$.

# Efficiency Issues

- We can randomly choose an integer $a$ and have a decent chance that $a$ is a generator. If it turns out not to be a generator, then we just pick a different $a$. This algorithm is very fast.

# Efficiency Issues

- We can randomly choose an integer $a$ and have a decent chance that $a$ is a generator. If it turns out not to be a generator, then we just pick a different $a$. This algorithm is very fast.

- Factoring $n-1$ is quite difficult (most of the time).

# A More Efficient Lucas Test

We don't have to factor $n-1$ completely. Factoring a "large enough" portion is usually sufficient.

**Theorem (Proth, Pocklington, Brillhart, Lehmer, & Selfridge)**

*Suppose that $a, F, n > 1$ are integers, $F \mid n-1$, $F > \sqrt{n}$,*

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

*and*

$$\gcd\left(a^{F/q} - 1, n\right) = 1$$

*for all primes $q \mid F$. Then $n$ is prime.*

**Exercise:** Prove!

# Other Primality Tests

The theoretical goal is to create a "fast" algorithm that always works (i.e. it tells us with 100% certainty that an integer is prime).

In 2002, **A**grawal, **K**ayal and **S**axena (**AKS**) published an algorithm for primality proving that is both fast (relatively speaking) and always works.

It solves the theoretical problem but, unfortunately, it doesn't finish off the practical problem - AKS requires many more computations than the probabilistic algorithms that we have discussed.

# Other Primality Tests

One of the fastest algorithms in use today is known as **Elliptic Curve Primality Proving** (ECPP). It uses the same basic idea for the Lucas Primality Proving, but instead of looking at orders of elements in $U_p$, ECPP examines orders of points (mod $p$) on an elliptic curve. However, this is a probabilistic algorithm because randomness is used in choosing the elliptic curve.

The most recent primality tests have relied on some extremely advanced ideas from algebraic number theory and algebraic geometry. These are computationally simple but quite difficult to understand.

# Why Do We Care if $n$ is Prime?

Knowing whether an integer $n$ is prime is useful in cryptography.

In general, it is much more difficult to factor an integer into a product of large primes than it is to multiply large primes together. Many cryptographic systems rely on this fact.

If you were able to quickly factor an integer into a product of two large primes and verify that they were both prime, you would be able to break into most banking systems.

# Public Key Cryptography

The first public key cryptosystem was created in 1976. Previously, secret messages that needed to be decoded required a private key that was available only to the sender and receiver. However, if the key fell into the wrong hands, then the secret message could easily be decoded. As a result, the sender and receiver would have to arrange for a secure exchange of the private key (ex. meeting face-to-face or transporting the key via a trusted courier).

# Public Key Cryptography


The idea behind a public key cryptosystem is that the key is published publicly, but only the receiver knows how to make use of it. The first public key cryptosystem was proposed by **R**ivest, **S**hamir and **A**dleman (**RSA**) in 1977. It is still widely used today.

# RSA: The Setup

Here is how I would set up my own system for receiving encrypted messages using RSA:

- I choose two distinct prime numbers $p$ and $q$.

# RSA: The Setup

Here is how I would set up my own system for receiving encrypted messages using RSA:

- I choose two distinct prime numbers $p$ and $q$.
- I compute $n = pq$.

# RSA: The Setup

Here is how I would set up my own system for receiving encrypted messages using RSA:

- I choose two distinct prime numbers $p$ and $q$.
- I compute $n = pq$.
- I compute $\varphi(pq) = (p-1)(q-1)$.

# RSA: The Setup

Here is how I would set up my own system for receiving encrypted messages using RSA:

- I choose two distinct prime numbers $p$ and $q$.
- I compute $n = pq$.
- I compute $\varphi(pq) = (p-1)(q-1)$.
- I choose an integer $e$ such that $1 < e < \varphi(pq)$, and $e$ and $\varphi(pq)$ share no divisors other than 1 (i.e., $e$ and $\varphi(pq)$ are coprime).

# RSA: The Setup

Here is how I would set up my own system for receiving encrypted messages using RSA:

- I choose two distinct prime numbers $p$ and $q$.
- I compute $n = pq$.
- I compute $\varphi(pq) = (p-1)(q-1)$.
- I choose an integer $e$ such that $1 < e < \varphi(pq)$, and $e$ and $\varphi(pq)$ share no divisors other than 1 (i.e., $e$ and $\varphi(pq)$ are coprime).
- I find the integer $d$ which satisfies the congruence

$$de \equiv 1 \pmod{\varphi(pq)}.$$

**Public:** $n, e$ **Private:** $p, q, d$

# RSA Encryption

Suppose that Jen wants to send me a message but she doesn't want Dr. Shapiro to read it. How can she use the 'key' $(n, e)$ that I have made public in order to encode her message in a way that, if it falls into the wrong hands (i.e. Dr. Shapiro picks it up), it still can't be read?

# RSA Encryption

- I would transmit my public key $(n, e)$ to Jen and keep my private key secret.

# RSA Encryption

- I would transmit my public key $(n, e)$ to Jen and keep my private key secret.

- Jen would turn her message into an integer $M$ between 0 and $n$ (for example, she could assign $A = 1$, $B = 2$,..., $Z = 26$).

# RSA Encryption

- I would transmit my public key $(n, e)$ to Jen and keep my private key secret.
- Jen would turn her message into an integer $M$ between 0 and $n$ (for example, she could assign $A = 1$, $B = 2$,..., $Z = 26$).
- In order to encode her message $M$, Jen would compute

$$E = M^e \pmod{n}$$

and send $E$ to me.

# RSA Decryption

In order to decode Jen's message, I would simply raise $E^d \pmod{n}$.

Why does this do the job?

$$E^d \equiv (M^e)^d \equiv M^{ed} \equiv M^{(\text{multiple of } \varphi(n))+1} \pmod{n}$$

$$\equiv 1 \times M \equiv M \pmod{n}.$$

Since both $M$ and $E^d$ lie between 0 and $n$, they must be equal! Now, I can convert $M$ back from a string of numbers into a string of letters.

# An RSA Example

- Choose two prime numbers: $p = 61$ and $q = 53$.

# An RSA Example

- Choose two prime numbers: $p = 61$ and $q = 53$.
- Compute $n = pq = 61 \cdot 53 = 3233$.

# An RSA Example

- Choose two prime numbers: $p = 61$ and $q = 53$.
- Compute $n = pq = 61 \cdot 53 = 3233$.
- Compute the $\varphi$ of product:
  $\varphi(61 \cdot 53) = \varphi(61) \cdot \varphi(53) = (61 - 1) \cdot (53 - 1) = 3120$.

# An RSA Example

- Choose two prime numbers: $p = 61$ and $q = 53$.
- Compute $n = pq = 61 \cdot 53 = 3233$.
- Compute the $\varphi$ of product:
  $\varphi(61 \cdot 53) = \varphi(61) \cdot \varphi(53) = (61 - 1) \cdot (53 - 1) = 3120$.
- Choose any number $e > 1$ that is coprime to 3120, ex. $e = 17$.

# An RSA Example

- Choose two prime numbers: $p = 61$ and $q = 53$.
- Compute $n = pq = 61 \cdot 53 = 3233$.
- Compute the $\varphi$ of product:
  $\varphi(61 \cdot 53) = \varphi(61) \cdot \varphi(53) = (61 - 1) \cdot (53 - 1) = 3120$.
- Choose any number $e > 1$ that is coprime to 3120, ex. $e = 17$.
- Compute $d$ such that

$$de \equiv 1 \ (\mathrm{mod}\ \varphi(pq)).$$

For example, if we use Euclid's algorithm, we see that $d = 2753$.

Thus, our public key is $(n = 3233, e = 17)$.

# An RSA Example

Public key: $(n = 3233, e = 17)$.

- Suppose Jen wants to send the message "Hi." Then, using the assignment $A = 1, B = 2, ..., Z = 26$, we see that Hi $= 89$.

# An RSA Example

Public key: $(n = 3233, e = 17)$.

- Suppose Jen wants to send the message "Hi." Then, using the assignment $A = 1, B = 2, ..., Z = 26$, we see that Hi $= 89$.

- To encrypt the message, Jen would send

$$E \equiv M^e \equiv 89^{17} \pmod{3233} \equiv 99 \pmod{3233}.$$

# An RSA Example

Public key: $(n = 3233, e = 17)$.

- Suppose Jen wants to send the message "Hi." Then, using the assignment $A = 1, B = 2, ..., Z = 26$, we see that Hi $= 89$.
- To encrypt the message, Jen would send

$$E \equiv M^e \equiv 89^{17} \pmod{3233} \equiv 99 \pmod{3233}.$$

- To decrypt the message, I would compute

$$E^d \equiv 99^{2753} \equiv 89 \pmod{3233}.$$

# An RSA Example

Public key: $(n = 3233, e = 17)$.

- Suppose Jen wants to send the message "Hi." Then, using the assignment $A = 1, B = 2, ..., Z = 26$, we see that Hi $= 89$.
- To encrypt the message, Jen would send

$$E \equiv M^e \equiv 89^{17} \pmod{3233} \equiv 99 \pmod{3233}.$$

- To decrypt the message, I would compute

$$E^d \equiv 99^{2753} \equiv 89 \pmod{3233}.$$

- Using the assignment $A = 1, B = 2, .., Z = 26$, I can conclude that Jen sent me the message "Hi."

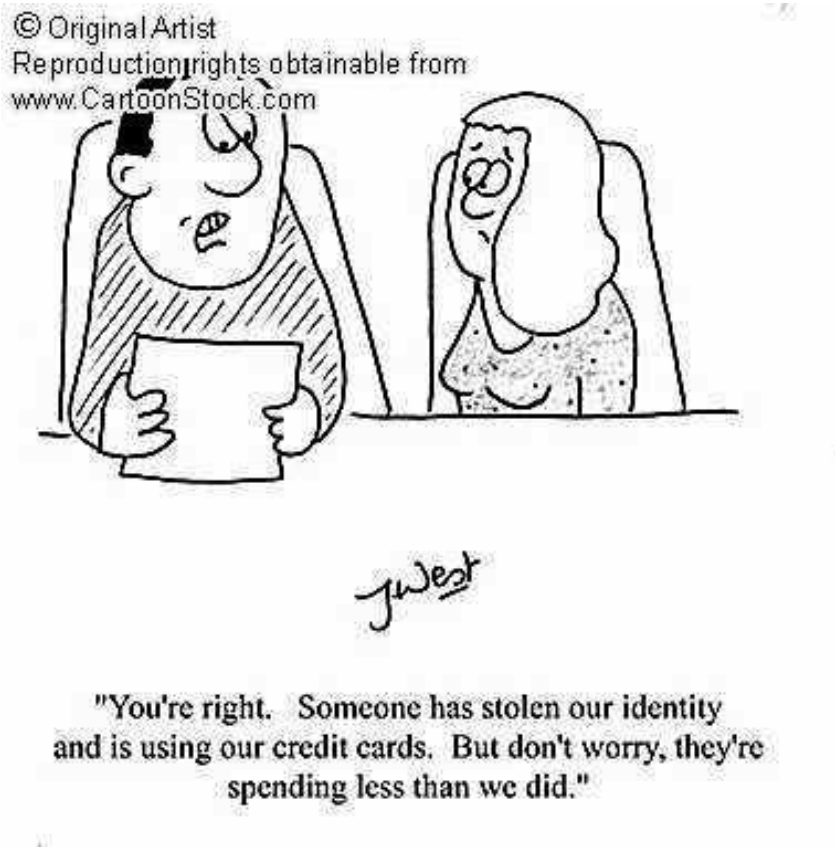
# Why is RSA Effective?

Without knowing $d$, it would be very difficult for Dr. Shapiro to decrypt Jen's message. Remember, the only information that he has is $(n, e)$. Since

$$d \equiv e^{-1} \pmod{(p-1)(q-1)}$$

then, in order to find $d$, he would have to be able to find both $p$ and $q$ by factoring $n$. As we discussed earlier, factoring $n$ into a product of two large primes is quite hard.

# Credit Card Check Digits




The first credit card (the Diners Club Card) made its debut in 1950. By 1954, the first credit card security-related patent had been submitted by Hans Peter Luhn. He created his algorithm in order to detect accidental errors in credit card digits, but it also has proven handy in detecting credit card fraud.

# Luhn's Algorithm

Modular arithmetic gives us a quick way to determine that a credit card number is fake:

- Label the rightmost digit with an $X$. This digit will henceforth be called the check digit.

# Luhn's Algorithm

Modular arithmetic gives us a quick way to determine that a credit card number is fake:

- Label the rightmost digit with an $X$. This digit will henceforth be called the check digit.
- Counting from the check digit and moving left, double the value of every second digit.

# Luhn's Algorithm

Modular arithmetic gives us a quick way to determine that a credit card number is fake:

- Label the rightmost digit with an $X$. This digit will henceforth be called the check digit.
- Counting from the check digit and moving left, double the value of every second digit.
- Sum the digits of the products together with the undoubled digits from the original number.

# Luhn's Algorithm

Modular arithmetic gives us a quick way to determine that a credit card number is fake:

- Label the rightmost digit with an $X$. This digit will henceforth be called the check digit.
- Counting from the check digit and moving left, double the value of every second digit.
- Sum the digits of the products together with the undoubled digits from the original number.
- Reduce this sum (mod 10). If the answer is not $\equiv -X \pmod{10}$ then your credit card number is fake!

# Example

Let's check to see whether the following can be an actual credit card number:

4147 2020 3679 7544

- Label the rightmost digit with an $X$. **4147 2020 3679 754X**

# Example

Let's check to see whether the following can be an actual credit card number:

4147 2020 3679 7544

- Label the rightmost digit with an $X$. **4147 2020 3679 754X**
- Counting from the check digit and moving left, double the value of every second digit. **8187 4040 66(14)9 (14)58X**

# Example

Let's check to see whether the following can be an actual credit card number:

4147 2020 3679 7544

- Label the rightmost digit with an $X$. **4147 2020 3679 754X**
- Counting from the check digit and moving left, double the value of every second digit. **8187 4040 66(14)9 (14)58X**
- Sum the digits of the products together with the undoubled digits from the original number. **76**

# Example

Let's check to see whether the following can be an actual credit card number:

4147 2020 3679 7544

- Label the rightmost digit with an $X$. **4147 2020 3679 754X**
- Counting from the check digit and moving left, double the value of every second digit. **8187 4040 66(14)9 (14)58X**
- Sum the digits of the products together with the undoubled digits from the original number. **76**
- Reduce this sum (mod 10). **6**

Since $6 \equiv -4 \equiv -X \pmod{10}$, then the credit card number might be valid.

# Trouble with Transpositions

Luhn's algorithm detects single digit substitutions (ex. accidentally writing a "7" instead of an "1") and **most** transpositions of digits.

There is one exception, however:

$$4147\ 202\mathbf{0}\ 367\mathbf{9}\ 7547$$

$$4147\ 202\mathbf{9}\ 367\mathbf{0}\ 7547$$

Here we've transposed a 0 with a 9, but both credit card numbers produce the same result when we apply Luhn's algorithm.

Why does this happen?

# Trouble with Transpositions

Luhn's algorithm detects single digit substitutions (ex. accidentally writing a "7" instead of an "1") and **most** transpositions of digits.

There is one exception, however:

$$4147\ 202\mathbf{0}\ 367\mathbf{9}\ 7547$$

$$4147\ 202\mathbf{9}\ 367\mathbf{0}\ 7547$$

Here we've transposed a 0 with a 9, but both credit card numbers produce the same result when we apply Luhn's algorithm.

Why does this happen?

- Notice that if 9 is in an even position (moving left from the check digit) then it will be doubled, resulting in 18 with $1 + 8 = 9$ as the sum of its digits. The 0 will be unaffected since it is in an odd position. On the other hand, if 0 were in an even position, its value doubled would be 0 and when added to 9, the sum is still 9. Either way, the sum is 9.

# Bad News/Good News

**The Bad News:** It's very easy for thieves to use modular arithmetic to engineer the check digit in a fraudulent credit card number so that it satisfies our congruence condition.

**The Good News:** Luhn's algorithm is still used by computers as an initial check to distinguish potentially valid credit cards from random collections of digits. However, before Amazon.com will send you the package that you ordered, they will use more sophisticated techniques to verify your identity.

# Further Reading

D. Bressoud, *Factorization and Primality Testing,* Springer, New York, 1989.

R. Crandall and C. Pomerance, *Prime numbers: a computational perspective,* 2nd ed., Springer, New York, 2005.

C. Pomerance, *Primality Testing: Variations on a Theme of Lucas,* Proceedings of the 13th Meeting of the Fibonacci Association, Congressus Numerantium 201 (2010), 301-312.