



Cryptography in Forensics:

Check Sums, Hash Functions, and the MD5 Algorithm

An Undergraduate Teaching Module



By: Melanie Brown, Champlain College, mbrown@champlain.edu;
Catherine Buell, Fitchburg State University, cbuell@fitchburgstate.edu; and
Alison Marr, Southwestern University, marra@southwestern.edu

Note to teachers: Teacher notes appear in dark red in the module, allowing faculty to pull these notes off the teacher version to create a student version of the module.

Summary of the Module

The MD5 Message-Digest Algorithm (MD5) is one of the current standards for data integrity verification for law enforcement and digital forensics. The algorithm uses a cryptographic function called a hash to produce a 32-character “word” or string from any type of data. This “word” or string is a nearly unique hexadecimal representation of the data. Law enforcement uses these unique signatures to prove the existence of particular data as well as proof that data has not been compromised. This module begins with modular arithmetic, binary and hexadecimal expressions, and bit operations in order to motivate the mathematics and the logic behind hash functions and the MD5 algorithm. The algorithm is considered with both a cryptographic and forensic lens. The module concludes with a discussion of current trends in digital verification and cybersecurity.

Target Audience:

The module can be adapted or divided into sub-modules that are appropriate for many levels of mathematics and computer science. Sections 1-5 cover applicable background material. Section 6 introduces the concept of hashing. Section 7 provide specific background material for MD5. Sections 8 and 9 present the MD5 algorithm and related topics. Section 10 discusses future avenues for hashing.

Prerequisites:

There are none.

Table of Contents

Section 1: Modular Arithmetic	p.4
Section 2: The Real World of Modular Arithmetic: Check Digits and Check Sums	p.8
Section 3: Binary and Bits	p.11
Section 4: ASCII: Computers and Binary	p.16
Section 5: Expressing Numbers in Hexadecimal	p.20
Section 6: Bit Operations	p.23
Section 7: Hashing and the MD5 Algorithm	p.32
Section 8: Uses of MD5 and Future Hashing Functions	p.43
References	p.45
Appendix: Extended Activities	p.46

Section 1: Modular Arithmetic

The basics of the MD5 algorithm, coding systems, and other data integrity verification methods, all start with the same building blocks. The roots of these methods lie in both mathematics and computer science. In these introductory sections, computer science topics like bits, bytes, and binary, as well as mathematical topics like modular arithmetic, base-two arithmetic, and check sums are discussed. Let's start with modular arithmetic.

To motivate the idea of modular arithmetic, consider your watch. If your watch says 5:00 and you have an appointment in 38 hours, what time will your watch read when the appointment starts? Or perhaps it is 11:00 and you have an exam in 42 hours. What time will your watch read as you sit down for the exam? How do we determine these times? What methods could be employed?

There are multiple ways to solve the above questions. One could think in either a 24 hour or a 12 hour day. Let's consider the fact that a watch cycles every 12 hours. If it is 5:00, then in 12 hours the watch will read 5:00 again. In 24 hours, the clock will read 5:00. In 36 hours, the clock will read 5:00. Since we are interested in 38 hours, then the watch will read 7:00 (38 hours is two more hours than 36 hours).

Now, if the world decided that one day was 5 hours long, then a watch might have the digits 1, 2, 3, 4, and 5. Let's explore some examples in this 5-hour-day world.

Example 1.1:

It is 1:00. What time will your watch read in 13 hours if one day is 5 hours long?

Answer: In this new world, 13 hours is equal to 2 days (each five hours) and 3 additional hours. We can write this mathematically as $13 = 5 * 2 + 3$. So, the watch will need to move 3 hours and it will read 4:00.

Your turn!

It is 5:00. What time will your watch read in 15 hours if one day is 5 hours long?

Answer: Since $15 = 5 * 3$, 15 hours is exactly 3 days later and the time will be the same: 5:00.

These are examples of modular arithmetic. We use it every day and we unknowingly interact with it every day through businesses, technology, and internet security.

Equivalence and Modular Arithmetic

Returning to the normal world with 24 hours days and the numbers 1-12 on a watch, we will discuss times that look the same on a watch face. For example, whether it is 1:00 or 13:00 both will appear as 1:00 on a watch face. Similarly, 1:00, 13:00, 25:00, and -11:00 are all same as 1:00 on a watch face.

Note:

$$\begin{aligned}1 &= 12 * 0 + 1 \\13 &= 12 * 1 + 1 \\25 &= 12 * 2 + 1 \\-11 &= 12 * -1 + 1\end{aligned}$$

All of these numbers have a remainder of 1 when divided by 12. Modular arithmetic can be thought of as “remainder arithmetic”. So, when we work “mod 12” meaning “watch face mathematics”, every number is said to be *equivalent* to its remainder when divided by 12. We should say that 1, 13, 25, and -11 are all equivalent to 1. Another way to see if two numbers are equivalent mod 12 is to check to see if the difference is divisible by 12.

So, 3:00, 15:00, and 27:00 are all equivalent mod 12 because they all have a remainder of three when divided by 12. Also $3 - 27 = -24$ and -24 is divisible by 12, so we can say that 3 and 27 are equivalent “mod 12.”

Note, -9:00, seems to makes little sense in the concept of time and watches, but it addresses subtracting time. Here -9:00, would also be equivalent to 3:00 because $-9 = 12 * (-1) + 3$ and $-9 - 3 = -12$ which is divisible by 12. Since grade school, we have known that remainders are always non-negative and always less than the original divisor. Formally, we state this idea as follows.

For any pair of integers (m, n) we can write $m = n * q + r$ where $0 < r < n - 1$, and q is called the *quotient* and must be an integer. We can restate this and say that r is an element in the set $\{0, 1, 2, \dots, n - 1\}$. The set $\{0, 1, 2, \dots, n - 1\}$ defines *equivalence classes*, as all the numbers with the same remainder will be in the same class. Numbers in the same class are *equivalent*. Sometimes we will call the set *equivalence classes mod n*.

This means there are finitely many equivalent classes and all integers fit into one of the classes. You can determine the equivalence class of an integer **mod n** by asking, “What is the remainder when this number is divided by n?” Again, we can check to see if two numbers, a and b , are equivalent **mod n** if they have the same remainder or if $(a - b)$ is divisible by n .

Problems working with *modulus* 7, call it **mod 7**. Now the remainders (the equivalence classes) are $\{0, 1, 2, 3, 4, 5, 6\}$. First, we determine which classes certain numbers belong. Use the notation:

$$13 \equiv 6 \pmod{7}.$$

Read the statement as “13 is equivalent to 6 **mod 7**”. We know this is true because $13 = 7 * 1 + 6$ and the \equiv symbol means equivalent.

Example 1.2:

What is 24 equivalent to **mod 7**?

Answer: $24 = 7 * 3 + 3$ so $24 \equiv 3 \pmod{7}$. So 24 is equivalent to 3 **mod 7**.

Your turn!

What is -7 equivalent to mod 7?

Answer: $-7 = 7 * (-1) + 0$ so $-7 \equiv 0 \pmod{7}$.

Example 1.3:

What is $24 + (-7)^5$ equivalent to **mod 7**? Note: Before performing any arithmetic, replace values with their mod 7 equivalent.

Answer: We could solve this problem by calculating each pieces. Since $(-7)^5 = -16807$, then we can say $24 + (-7)^5 = 24 + (-16807) = -16783 = 7 * (-2398) + 3$, so the expression is equivalent to 3 **mod 7**.

However, we really should use the fact from the previous example that -7 is equivalent to 0 **mod 7**. We also know that $24 \equiv 3 \pmod{7}$ (because $24 = 7 * 3 + 3$). A more efficient answer is to say, $24 + (-7)^5 \equiv 3 + 0^5 \pmod{7} \equiv 3 \pmod{7}$.

Your turn!

Solve the following examples by replacing values with its **mod n** equivalent before doing any arithmetic. Do not use a calculator.

What is $7^2 + (5 * 57)$ equivalent to **mod 48**?

Answer: $49 + (5 * 9) \pmod{48} \equiv 1 + (45) = 46$.

What is $4^3 + 3^2 + 17^{81}$ equivalent to **mod 4**?

Answer: $0^3 + 9 + 1^{81} = 10 \pmod{4} \equiv 2$.

Example 1.4:

What is 9^{2000} equivalent to **mod 80**? (Hint: Write 2000 as $2 * 1000$ and simplify.)

Answer: This problem uses some exponential properties. So,
 $9^{2000} = 9^{2*1000} = (9^2)^{1000} = 81^{1000} \equiv 1^{1000} \bmod 80 \equiv 1 \bmod 80.$

Your turn!

What is $(x + y)^2$ equivalent to **mod 2**? (Hint: Multiply $(x + y)^2$ out.)

Answer: $(x + y)^2 = x^2 + 2xy + y^2 \equiv x^2 + y^2 \bmod 2.$

Homework Exercises Section 1

1. It is 8:00. What time will your watch read in 15 hours if one day is 9 hours long?
2. It is 3:00. What time will your watch read in 9 hours if one day is 5 hours long?
3. What is 82 equivalent to mod 7?
4. Solve the following examples by replacing values with its **mod n** equivalent before doing any arithmetic. Do not use a calculator.
 - a. What is $6^2 + (36 * 55)$ equivalent to **mod 35**?
 - b. What is $4^3 + 30^2 + 16^{81}$ equivalent to **mod 5**?

Section 2: The Real World of Modular Arithmetic: Check Digits and Check Sums

UPCs, ISBNs, and bank accounts numbers are all examples of modular arithmetic in the real world. MD5 and other internet security systems like RSA also use modular arithmetic to disguise, simplify, and verify information.

A UPC (Universal Product Code) is a 12 digit number or barcode that encodes manufacturer information, product information, and a check digit. A check digit is an additional number at the end of the string of digits that can verify if a mistake was made in the previous 11 digits. As we will see later, MD5 also adds digits to the end of a given string. The following is a general example of a UPC.

$$\text{UPC} = d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10} d_{11} c$$

Here $d_1, d_2, d_3, \dots, d_{11}$ encode the product information and c is the check digit. All the digits must be 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

The system used to both determine the check digit and to verify that the values of d_1 through d_{11} are correct alternately multiplies by 3 and 1, adds them together, then determines the value mod 10. First, the following is calculated using the product information:

$$3d_1 + 1d_2 + 3d_3 + 1d_4 + 3d_5 + 1d_6 + 3d_7 + 1d_8 + 3d_9 + 1d_{10} + 3d_{11}$$

Then, the check digit, c , is chosen such that

$$3d_1 + d_2 + 3d_3 + d_4 + 3d_5 + d_6 + 3d_7 + d_8 + 3d_9 + d_{10} + 3d_{11} + c \equiv 0 \text{ mod } 10$$

Example 2.1:

Suppose a product's information can be encoded in the 11 digit string 03600028510. When the UPC is made, the new code will be 03600028510c. Keep in mind it that c must be a single digit and the sum $3d_1 + d_2 + 3d_3 + d_4 + 3d_5 + d_6 + 3d_7 + d_8 + 3d_9 + d_{10} + 3d_{11} + c$ must be equivalent to 0 mod 10. What should c be?

Answer: We can calculate most of the sum above given the known digits and leaving the check digit as c .

$$\begin{aligned} & 3(0) + (3) + 3(6) + 0 + 3(0) + 0 + 3(2) + 8 + 3(5) + 1 + 3(0) + c \\ &= 0 + 3 + 18 + 0 + 0 + 0 + 6 + 8 + 15 + 1 + 0 + c \\ &= 51 + c \end{aligned}$$

For $51 + c$ to be equivalent to 0 mod 10, c must be 9 because $51 + 9 = 60 \equiv 0 \text{ mod } 10$. We can ask ourselves what value of c makes the total a multiple of 10 or we can find $51 + c \bmod 10 \equiv 1 + c \bmod 10$ and again c must be 9.

Your turn!

Is 036000281509 a valid UPC?

Answer:
$$\begin{aligned} & 3(0) + 3 + 3(6) + 0 + 3(0) + 0 + 3(2) + 8 + 3(1) + 5 + 3(0) + 9 \\ &= 0 + 3 + 18 + 0 + 0 + 0 + 6 + 8 + 3 + 5 + 0 + 9 \\ &= 52 \equiv 2 \pmod{10}. \end{aligned}$$

No, this is NOT a valid UPC because the sum is 2 and not 0 **mod 10**.

Bank Accounts are typically 10 digits (9 digits and 1 check digit):

$$n_1 n_2 n_3 n_4 n_5 n_6 n_7 n_8 n_9 c.$$

The check digit is determined and the code verified in a different way than UPC codes. Here they multiply by 7, 3, and 9 alternately.

$$7n_1 + 3n_2 + 9n_3 + 7n_4 + 3n_5 + 9n_6 + 7n_7 + 3n_8 + 9n_9 + c \equiv 0 \pmod{10}$$

Example 2.2:

Compute the check digit for following bank account: 211872438c.

Answer: We can calculate most of the sum above given the known digits and leaving the check digit as c.

$$\begin{aligned} & 7(2) + 3(1) + 9(1) + 7(8) + 3(7) + 9(2) + 7(4) + 3(3) + 9(8) + c \\ &= 14 + 3 + 9 + 56 + 21 + 18 + 28 + 9 + 72 + c \\ &= 230 + c \end{aligned}$$

For this to be a bank account number, $230 + c$ must be equivalent to 0 mod 10. So $c = 0$.

Your turn!

Is the following a correct bank code: 0123456789?

Answer:
$$\begin{aligned} & 7(0) + 3(1) + 9(2) + 7(3) + 3(4) + 9(5) + 7(6) + 3(7) + 9(8) + 9 \\ &= 0 + 3 + 18 + 21 + 12 + 45 + 42 + 21 + 72 + 9 \\ &= 243 \equiv 3 \pmod{10} \end{aligned}$$

This is not a valid bank code because the sum is 3 and not 0 **mod 10**.

Suppose you are given the bank account 211872d461? Can you uniquely determine d?

Answer:
$$\begin{aligned} &7(2) + 3(1) + 9(1) + 7(8) + 3(7) + 9(2) + 7(d) + 3(4) + 9(6) + 1 \\ &= 14 + 3 + 9 + 56 + 21 + 18 + 7d + 12 + 54 + 1 \\ &= 188 + 7d \equiv 8 + 7d \pmod{10} \end{aligned}$$

Since this is a bank account, $8 + 7d \equiv 0 \pmod{10}$. So $d = 6$ because $8 + 7(6) = 50 \equiv 0 \pmod{10}$.

Extended Activity A: Consider some reasons that the banks might use 7, 3, and 9 in their calculations when working mod 10 but not use 2,4,5, etc?

Notes/Hints:

1. Consider if the previous problem came down to solving $8 + 4d \equiv 0 \pmod{10}$ instead of $8 + 7d \equiv 0 \pmod{10}$.
2. The main idea here is the idea of *relatively prime* numbers. In the example, 7, 3 and 9 are all relatively prime to 10.
3. See the Appendix under Extended Activity A for more details.

Homework Exercises Section 2

1. Compute the check digit for following bank account: 311272437c.
2. Is 725439104708 a valid UPC?

Section 3: Binary and Bits

While modular arithmetic is a large part of many cryptographic and security protocols, we also have to remember that all this information is communicated in “computer language.”

Computers are incredibly complicated machines with multi-layered programs, systems, and functions, but under it all are small switches with two settings: “on” and “off” or “0” and “1”. This is why we briefly discuss binary and bits. The word bit is a shorthand for **binary digit**.

Binary Notation and Expansion

We live in a decimal number system (sometimes called denary), meaning we have a base-ten system. We use the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and place value to determine the value of a string of digits. We see 145.2 and we know this means $1 * 10^2 + 4 * 10^1 + 5 * 10^0 + 2 * 10^{-1}$. A computer uses binary (base-two). Binary has the digits 0 and 1 and also uses place value to determine the value of a string of digits. We will use the notation XXX_2 to denote a number is written in base 2. If a number is in base 10, then we are not going to write XXX_{10} and instead we will just write XXX .

Converting a value in binary into a decimal number is a matter of expanding using place value. Visually, we can write 1001_2 in a place value table and then calculate.

2^3	2^2	2^1	2^0
1	0	0	1

$$\text{So, } 1001_2 = 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 8 + 1 = 9.$$

However, we need to have a better strategy to make a decimal number into its binary representation. It is a very good exercise because it reminds us about the basics and foundations of the number system, place value, and grouping. When we learned the base-ten system, we needed to know to group by ones (10^0), tens (10^1), or hundreds (10^2), etc. In binary, or base-two, we group by ones (2^0), “twos” (2^1), “fours” (2^2), “eights” (2^3), etc.

Suppose we are asked to describe 140 as a base-ten value. We see this as one hundred (10^2), four tens (10^1), and zero ones (10^0). We start by looking for the largest power of ten that fits into the number. However, in binary, 140 is one 2^7 (128), zero 2^6 (64), zero 2^5 (32), zero 2^4 (16), one 2^3 (8), one 2^2 (4), zero 2^1 (2), and zero 2^0 . We write this as 10001100_2 . We determine the expansion by first finding the largest power of two that could fit into the number.

First, $128 = 2^7$ and $256 = 2^8$, so 2^7 is the largest power of two less than 140. This leaves $140 - 128 = 12$ remaining to be built from powers of two. The next largest power of two that fits in 12 would be $8 = 2^3$ and we have $12 - 8 = 4$ remaining. Since 4 is a power of two (2^2), we are done!

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	1	1	0	0

$$=10001100_2$$

Example 3.1:

Convert 10001_2 into base-ten.

Answer: Like we saw in the original example, we can expand using place value. So,
 $10001_2 = 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 16 + 1 = 17$

Your turn!

Convert 111_2 into base-ten.

Answer: $1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 4 + 2 + 1 = 7$

Convert 00101_2 into base-ten.

Answer: $0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 1 = 5$

Note: It should seem odd to have 00 in the leftmost place; however, sometimes this will happen in binary, especially when the computer forces the user to always use a certain number of bits.

Convert 101_2 into base-ten.

Answer: $1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 1 = 5$

Example 3.2:

Convert 25 into binary.

Answer: Recall, we need to find the highest power of two that fits in 25. The largest power is $2^4 = 16$. We know $25 - 16 = 9$, so we will need an additional 8 (2^3) and a 1 (2^0). So,
 $25 = 16 + 8 + 1 = 2^4 + 2^3 + 2^0 = 11001_2$.

Your turn!

Convert 255 into binary.

Answer: $255 = 128 + 32 + 16 + 8 + 4 + 2 + 1$
 $= 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 1111111_2$

Convert 48 into binary using 8 binary digits (bits).

Answer: $46 = 32 + 8 + 4 + 2 = 2^5 + 2^3 + 2^2 + 2^1 = 00101110_2$

Extended Activity B: There are numerous algorithms to multiply two numbers: the standard algorithm, area method, lattice methods, distribute/rewriting, etc. There is also a method that uses the binary expansion of a number (behind the scenes) to do multiplication. Here are two examples.

The Algorithm:

1. To multiply m and n , create two columns. In one column, start with m , the entry below is the entry above divided by 2. So $m, m/2, m/2/2$, etc. Stop when the value is 1. If any entry is odd, then we use the floor function (take the integer less than or equal to the value). For example, we would say $7/2 = 3$. In the other column, multiply n by 2 and each row will be twice the row before it. Stop when the value in the first column is 1.
2. Cross out any row in the table where there is an even value in the leftmost column.
3. Add any remaining values in the rightmost column, this is mn .

Suppose we wish to multiply 16 and 31. We will create two columns.

16	31
8	62
4	124
2	248
1	496

Cross out all even values in the left-most column and those matched in the other column.

16	31
8	62
4	124
2	248
1	496

Therefore, $16 * 31 = 496$.

Suppose we wish to multiply 31 and 25. We will create two columns.

29	25
14	50
7	100
3	200
1	400

Therefore, $29 * 25 = 25 + 100 + 200 + 400 = 725$.

The idea is that dividing by two and multiplying by two is “easy” or at least easier than the standard algorithm. However, how and why does the algorithm work and what is its connection to the binary expansion of a number?

Notes/Hints:

1. See the Appendix under Extended Activity B for related links and references.

Binary Arithmetic

Arithmetic is interesting and quite fun in binary. In the decimal system, digits representing the same place value are added together, and if a group of ten can be made, then we carry over into the next place value. In binary, the same thing happens; however, we make groups of two.

Example 3.3:

Calculate $101_2 + 1001_2$.

Answer: Looking at the ones column, $1 + 1 = 2$, so we would “group and carry” into the next place value. Then,

$$\begin{array}{r} 101_2 \\ + 1001_2 \\ \hline 1110_2 \end{array}$$

Your turn!

Evaluate $10_2 + 1100_2$.

Answer: 1110_2

Evaluate $11001_2 + 01010_2$.

Answer: 100011_2

Homework Exercises Section 3

1. Convert the following into base-ten
 - a. 1010_2
 - b. 01101_2

2. Convert 321 into binary.
3. Convert 55 into binary using 8 binary digits (bits).
4. Evaluate $011_2 + 1100_2$.
5. Evaluate $11111_2 + 11110_2$.

Section 4: ASCII: Computers and Binary

Translating decimal numbers to binary is only one requirement for data transmission. In fact, numbers, letters, words, and images are just some of the data can all be transferred in binary. One common translation between letters/words and binary is the ASCII system. The American Standard Code for Information Interchange (ASCII) assigns numbers to all uppercase letters, lowercase letters, selected punctuation, transmission controls, and some special characters. There are two codes: Standard ASCII and Extended ASCII.

For example **M** is assigned the number 77. However, this number is translated into binary and transferred, so **M** = 1001101_2 (Standard ASCII) or 01001101_2 (Extended ASCII). In total, there are 128 symbols in Standard ASCII and 256 in Extended ASCII. These values, 128 and 256, are determined by the fact that computers use binary representations for letters, numbers, and symbols and the ASCII systems work with binary strings of different length. The Standard ASCII uses 7-bit strings for all values, so there are $2^7 = 128$ characters because there are 7 digits and two choices for each digit (0 or 1). The Extended ASCII uses 8-bit strings for all values, so by the same reasoning there are $2^8 = 256$ characters/commands. Let's look at the two values for **M**. Note the biggest difference are the number of digits, but the values are the same. Let's look at the place value.

$$\begin{aligned} \mathbf{M} = 1001101_2 &= 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ &= 64 + 8 + 4 + 1 = 77 \end{aligned}$$

$$\begin{aligned} \mathbf{M} = 01001101_2 &= 0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ &= 64 + 8 + 4 + 1 = 77 \end{aligned}$$

Your turn!

The letter **C** has the value of 67. Express **C** in 7-bit Standard ASCII and 8-bit Extended ASCII.

Answer: $67 = 64 + 2 + 1 = 2^6 + 2^1 + 2^0 = 1000011_2$ in Standard ASCII and 01000011_2 in Extended ASCII.

The next page has a table of Extended ASCII binary codes.

In class, it would be a good exercise to erase some of the binary entries and have students determine the binary elements.

Extended ASCII Binary	Decimal	Letter	Extended ASCII Binary	Decimal	Letter
01000001	65	A	01001110	78	N
01000010	66	B	01001111	79	O
01000011	67	C	01010000	80	P
01000100	68	D	01010001	81	Q
01000101	69	E	01010010	82	R
01000110	70	F	01010011	83	S
01000111	71	G	01010100	84	T
01001000	72	H	01010101	85	U
01001001	73	I	01010110	86	V
01001010	74	J	01010111	87	W
01001011	75	K	01011000	88	X
01001100	76	L	01011001	89	Y
01001101	77	M	01011010	90	Z

Your turn!

Translate the following binary message using the table.

01010011 01000101 01000011 01010101 01010010 01000101

Answer: **SECURE**

Encode KEY using the table.

Answer: **01001011 01000101 01011001**

You may have heard the word bit or byte before. Usually we hear kilobyte (KB), megabyte (MB), or gigabyte (GB). A *bit* (or **binary digit**) is a single digit (either 0 or 1). So a given letter uses 8 bits. Given that much communication uses 8 bits to represent letters and commands, 8 bits are called a *byte*. To encode KEY, you needed 24 bits or 3 bytes.

You can see how quickly the data can grow. Try converting your first and last name to binary! In later sections, we will be learn about more efficient number systems.

Checksums as Digital Signatures

Utilizing the ASCII code above, we can introduce our first digital signature. MD5, SHA-1, and SHA-2 are examples of digital signatures. These hashes give a “unique” way to identify digital data, hence its signature. One can create a digital signature for a word, a file, or an entire hard drive. One very basic example would be doing an 8 digit checksum for a word.

Consider the word FROG. The value of the word FROG would be

$$F+R+O+G = 70 + 82 + 79 + 71 = 302.$$

Since we want to write this as an 8 bit value, we would take $302 \bmod 256$ which is equivalent to 46. The digital signature would be 00101110 in binary since $46 = 32 + 8 + 4 + 2$.

If instead we replaced the O in FROG with the number 0, then $FR0G = F+R+0+G$. The numerals 0 through 9 are not in the table above; however, they taken on the decimal values 48 through 57, where 0 is worth 48, 1 is worth 49, 2 is worth 50, etc. So

$$FR0G = 70 + 82 + 48 + 71 = 271$$

which is equivalent to $15 \bmod 256$. The digital signature of $FR0G$ is 00001111.

Clearly $FR0G$ and FROG give different signatures. You cannot recreate the original word from the digital signature, but you could tell whether or not the signature could belong to FROG or $FR0G$. We use the word “could” because there is going to be a problem, or what we will later call a collision. To demonstrate this issue, try the following problems:

Your turn!

Find the 8 bit digital signature for ZD.

Answer: $Z+D = 90 + 68 = 178$. The digital signature is 10110010.

Find the 8 bit digital signature for A22DIGITAL22YR

Answer: $A+2+2+D+I+G+I+T+A+L+2+2+Y+R$
 $= 65 + 50 + 50 + 68 + 73 + 71 + 73 + 84 + 65 + 76 + 50 + 50 + 89 + 82$
 $= 178 \bmod 256$.
The digital signature is 10110010.

The previous examples demonstrate that two very different words can have the same signature. We want to know whether we are looking at ZD or at A22DIGITAL22YR. In order for these two strings to have different digital signatures, we will need to create more complicated methods than using a checksum approach.

Example 4.1:

Another fun example of a collision is DORMITORY and DIRTYROOM. You could calculate the checksums to verify that have the same digital signature; however, you don't need too. Can you see why?

Answer: The letters are the same just in a different arrangement.

Your turn!

Find two words (one complicated and one simple like our ZD example) and show that they have the same checksum.

Answers may vary.

Homework Exercises Section 4

1. Translate MAP into binary.
2. Find the 8 bit digital signature for COOL.
3. Find three words that will have the same digital signature where you will know the signatures are the same without any calculation.

Section 5: Expressing Numbers in Hexadecimal

The word *hexadecimal* comes from the Greek word *hex*, meaning “six” and the Latin word *decem*, meaning “ten.” Putting these together, we see that *hexadecimal* means “of sixteen.” Expressing numbers in hexadecimal is very similar to the work we did with binary in Section 3, but instead of writing in base 2, we are going to use base 16. In base 2, we used the symbols 0 and 1. In base 16, we will use 0, 1, 2, ..., 9, 10, 11, 12, 13, 14, 15. However, we cannot actually use 10, 11, 12, 13, 14, or 15 because each number has two digits. Instead, we will use letters:

Decimal	10	11	12	13	14	15
Hexadecimal	a	b	c	d	e	f

Let’s look at $n = 2519$. The digits and places tell us exactly how to obtain the number.

In decimal: $2519 = 2 \cdot 1000 + 5 \cdot 100 + 1 \cdot 10 + 9 \cdot 1 = 2 \cdot 10^3 + 5 \cdot 10^2 + 1 \cdot 10^1 + 9 \cdot 10^0$

When we converted numbers into binary, we used a similar idea, but instead of powers of 10, we used powers of 2. To write 2519 in hexadecimal, we repeat the process, but with powers of 16. The coefficients will be the digits 0, 1, ..., 9, a, b, c, d, e, f . To help with the process, the first few powers of 16 are listed in the table below:

16^2	16^3	16^4	16^5
256	4096	65536	1048576

In hexadecimal: $2519 = 9 \cdot 16^2 + 13 \cdot 16^1 + 7 \cdot 16^0 = 9d7_{16}$

As we did with binary, we will denote the number expressed in hexadecimal with a subscript of 16.

As a review, we can also express 2519 in binary:

$$\begin{aligned} 2519 &= 1 \cdot 2^{11} + 0 \cdot 2^{10} + 0 \cdot 2^9 + 1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 \\ &\quad + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 100111010111_2 \end{aligned}$$

Since $16 = 2^4$, we can quickly convert binary to hexadecimal and vice versa by taking breaking up our binary string into 4-bit pieces and converting each piece to its decimal value and then its hexadecimal value.

In our example above,

2519 =	1001	1101	0111
Decimal	9	13	7
Hex	9	d	7

This is so much faster! It can be easier to convert decimal to hexadecimal by going through binary first, and we will use that process throughout.

Example 5.1:

Express $n = 753$ in hexadecimal.

Answer: We first express 753 in binary:
 $753 = 10\ 1111\ 0001_2$

Observe that the binary word has 10 digits, which is not divisible by 4. We can solve this problem by splitting the string into 4-bit pieces *starting at the right*. In the left-most piece, we will add as many zeroes as necessary to give it length 4. In this case, we need two zeros:

$$753 = 10\ 1111\ 0001_2 = 0010\ 1111\ 0001_2$$

Now we will rewrite each 4-bit piece in decimal (and hexadecimal if necessary):

753 =	0010	1111	0001
Decimal	2	15	1
Hex	2	f	1

Therefore we have $753 = 2f1_{16}$

Example 5.2:

Find the decimal expansion for $b47c_{16}$

Answer: Based on the digit placement,

$$\begin{aligned}
 b47c_{16} &= b \cdot 16^3 + 4 \cdot 16^2 + 7 \cdot 16^1 + c \cdot 16^0 \\
 &= 11 \cdot 16^3 + 4 \cdot 16^2 + 7 \cdot 16^1 + 12 \cdot 16^0 \\
 &= 46204
 \end{aligned}$$

Your Turn!

Give the decimal expansion of each number.

a. $5b3_{16}$

$$\begin{aligned}
 \text{Answer: } 5b3_{16} &= 5 \cdot 16^2 + b \cdot 16 + 3 \cdot 16^0 \\
 &= 5 \cdot 16^2 + 11 \cdot 16 + 3 \cdot 16^0 \\
 &= 1280 + 176 + 3 = 1459
 \end{aligned}$$

b. 27_{16}

Answer: $27_{16} = 2 \cdot 16^1 + 7 \cdot 16^0$
 $= 32 + 7 = 39$

c. $fade_{16}$

Answer: $fade_{16} = f \cdot 16^3 + a \cdot 16^2 + d \cdot 16^1 + e \cdot 10^0$
 $= 15 \cdot 16^3 + 10 \cdot 16^2 + 13 \cdot 16^1 + 14 \cdot 10^0$
 $= 61440 + 2560 + 208 + 14$
 $= 64222$

Your Turn!

Give the hexadecimal expansion of each number.

a. 9715

Answer: $9715 = 10010111110011_2$
 $= 10 \ 0101 \ 1111 \ 0011_2$
 $= 0010 \ 0101 \ 1111 \ 0011_2$
 $= 2 \quad 7 \quad 15 \quad 3$
 $= 27f3_{16}$

b. 1001110100111_2

Answer: $1001110100111_2 = 1 \ 0011 \ 1010 \ 0111_2$
 $= 0001 \ 0011 \ 1010 \ 0111_2$
 $= 1 \quad 3 \quad 10 \quad 7$
 $= 13a7_{16}$

Homework Exercises Section 5

1. Give the decimal expansion of each number.

a. $3b7_{16}$

b. 92_{16}

c. fed_{16}

2. Give the hexadecimal expansion of each number.

a. 2371

b. 100110011001011_2

Section 6: Bit Operations; Cryptography versus Hashing

The security and integrity of the MD5 Hash Algorithm lies in the complex systems of steps required to transform the data. However, each step is fairly simple and can be described in terms of *bit operations*. These bit operations give us different ways to view arithmetic operations on binary strings. In “normal arithmetic,” we perform operations using the entire numbers:

To add 23 and 79, we would see that in the ones place, $3 + 9 = 12$, so we would record the 2 and carry the 1. Then $1 + 2 + 7 = 10$ and

$$\begin{array}{r} 23 \\ +79 \\ \hline 102 \end{array}$$

To multiply 3 and 74, we can use a few different methods:

$$\begin{array}{r} 74 \\ \times 3 \\ \hline 222 \end{array}$$

or

$$3(74) = 3(70 + 4) = 3(70) + 3(4) = 210 + 12 = 222$$

Even the binary arithmetic we did earlier used the entire number:

$$\begin{array}{r} 101_2 \\ +1001_2 \\ \hline 1110_2 \end{array}$$

Bit operations do not use the entire number. Instead, we perform the arithmetic bit-by-bit. To perform a bit operation, we first need to convert any decimal numbers into binary. We will explore four bit operations: AND, OR, XOR, and NOT. The operations AND, OR, and XOR each compares two bits and returns a new bit. The operation NOT only requires one bit to return a new bit. In computer science courses, you might hear people refer to AND, OR, and XOR as *binary operators*. In this context, “binary” refers to the two bits required for inputs. A computer scientist might also refer to NOT as a *unary operator* because it only requires one bit as an input. Since each operation is bit-wise, we need to make sure all strings are the same length, that is, they each have the same number of bits.

For each of these definitions, lowercase letters will represent bits and uppercase letters will represent strings.

AND: The AND operation compares two bits. If both bits are 1, then it returns a 1. Otherwise it returns a 0.

There are many ways to interpret this operation. For example, we can express AND as a piece-wise function. For two bits x and y ,

$$x \text{ AND } y = \begin{cases} 1 & \text{if } x = y = 1 \\ 0 & \text{if } x = 0 \text{ or } y = 0 \end{cases}$$

Another way to view the AND operation is using a truth table where 1 = True and 0 = False. We read truth tables just like we read addition tables:

AND	1	0
1	1	0
0	0	0

Using the table, we can read

$$1 \text{ AND } 0 = 0$$

We also see that AND has the same *commutative* property as addition and multiplication do, so

$$1 \text{ AND } 0 = 0 \text{ AND } 1 = 0$$

Why do we call the table above a Truth Table? We can determine whether a compound statement is True or False based on whether each simple statement is True or False. Consider the following statements:

- A. The sky is blue
- B. The Earth is round
- C. The Moon is made of cheese

We know that statements A and B are true, but C is false. The statement, “The sky is blue AND the Earth is round” is true because we know each simple statement is true. The statement, “The Earth is round AND the Moon is made of cheese” is false because the second part is false. Two true statements combine to make another true statement, but if one part is false, then the entire statement is false. We can also rewrite these two statements in terms of 1 for True and 0 for False:

$$\text{“The sky is blue AND the Earth is round”} = \text{True} \Leftrightarrow 1 \text{ AND } 1 = 1$$

$$\text{“The Earth is round AND the Moon is made of cheese”} = \text{False} \Leftrightarrow 1 \text{ AND } 0 = 0$$

This use of AND is also the same as how it is used in probability. Suppose an event is “Drawing a red Jack” from a standard 52-card bridge deck. When drawing a single card, we need it to be both Red AND Jack to satisfy the event.

Now that we are more comfortable with the idea of AND, let's try using it as a bit operation.

Example 6.1:

Let $X = 10110$ and $Y = 11010$. We first note that each string has 5 bits, so they have equal length. To compute $X \text{ AND } Y$, we compare bits in the same position. Since this is performed bit-wise, we can start on the left or we can start on the right. Starting on the right is probably habit for most people, so we can start there:

$$\begin{aligned} 0 \text{ AND } 0 &= 0 \\ 1 \text{ AND } 1 &= 1 \\ 1 \text{ AND } 0 &= 0 \\ 0 \text{ AND } 1 &= 0 \\ 1 \text{ AND } 1 &= 1 \end{aligned}$$

Putting it together, we have

$$\begin{array}{r} 10110 \\ \text{AND } 11010 \\ \hline 10010 \end{array}$$

As actual numbers, $= 10110_2 = 22$, $Y = 11010_2 = 26$ and $X \text{ AND } Y = 10010_2 = 18$. The AND operation is bit-wise, so the entire number rarely gives meaningful information.

Your turn!

Let $X = 11010010$ and $Y = 10111001$. Compute $X \text{ AND } Y$

Answer:

$$\begin{array}{r} 11010010 \\ \text{AND } 10111001 \\ \hline 10010000 \end{array}$$

OR: The OR operation compares two bits. If at least one bit is a 1, then it returns a 1. If both bits are 0, it returns a 0. Just as with AND, we can express it as a piecewise function:

$$x \text{ OR } y = \begin{cases} 1 & \text{if } x = 1 \text{ or } y = 1 \\ 0 & \text{if } x = y = 0 \end{cases}$$

The truth table for OR is:

OR	1	0
1	1	1
0	1	0

We see that OR is also commutative. With OR statements, we only need one part to be true to make the entire statement true. Saying that "The Earth is round OR the Moon is made of cheese" is a true statement because either "The Earth is round" is true or "The Moon is made of cheese" is true, and we know the Earth is indeed round.

This is the same idea we see with the OR in probability. Going back to our standard bridge deck, suppose an event is “Drawing a red OR a Jack.” Drawing the 2 of Diamonds would satisfy the event because it is a red card, even though it is not a Jack. Drawing the Jack of Clubs would satisfy the event because it is a Jack, even though it is a black card.

Example 6.2:

Now let’s try using OR as a bit operation. Let $X = 10110$ and $Y = 11010$. Both strings are the same length, so we can proceed. To compute $X \text{ OR } Y$, we compare bits in the same position. Again, the starting side does not matter, but we will start on the right because of long-ingrained habit:

$$\begin{aligned} 0 \text{ OR } 0 &= 0 \\ 1 \text{ OR } 1 &= 1 \\ 1 \text{ OR } 0 &= 1 \\ 0 \text{ OR } 1 &= 1 \\ 1 \text{ OR } 1 &= 1 \end{aligned}$$

Putting it together, we have

$$\begin{array}{r} 10110 \\ \text{OR } 11010 \\ \hline 11110 \end{array}$$

Your turn!

Let $X = 11010010$ and $Y = 10111001$. Compute $X \text{ OR } Y$

Answer:

$$\begin{array}{r} 11010010 \\ \text{OR } 10111001 \\ \hline 11111011 \end{array}$$

XOR: The meaning of “OR” can be a bit ambiguous in the case where both bits are true. If the compound statement is true, then it is unclear whether both simple statements are true or just one of them. The term “XOR” stands for “Exclusive Or” because it behaves like OR, but it excludes the case where $x \text{ AND } y$ is true. As a piecewise function,

$$x \text{ XOR } y = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

In other words, XOR compares two bits and returns a True if the two bits are different and a False if the two bits are the same. We can think of XOR representing “one or the other, but not both.”

As a table, this operation can be represented

XOR	1	0
1	0	1
0	1	0

Example 6.3:

Let $X = 10110$ and $Y = 11010$. Both strings are the same length, so we can proceed. To compute $X \text{ XOR } Y$, we compare bits in the same position. Again, the starting side does not matter, but we will start on the right because of long-ingrained habit:

$$\begin{aligned} 0 \text{ XOR } 0 &= 0 \\ 1 \text{ XOR } 1 &= 0 \\ 1 \text{ XOR } 0 &= 1 \\ 0 \text{ XOR } 1 &= 1 \\ 1 \text{ XOR } 1 &= 0 \end{aligned}$$

Putting it together, we have

$$\begin{array}{r} 10110 \\ \text{XOR } 11010 \\ \hline 01100 \end{array}$$

Your turn!

Let $X = 11010010$ and $Y = 10111001$. Compute $X \text{ XOR } Y$

Answer:

$$\begin{array}{r} 11010010 \\ \text{XOR } 10111001 \\ \hline 01101011 \end{array}$$

Observe that in both examples, the answers differ from those of OR in the positions where both bits were the same.

NOT: The NOT operation looks at a single bit and returns another single bit. We can view NOT as a *bit flip* operation because each bit is going to “flip” values so that a 0 becomes a 1 and a 1 becomes a 0:

$$\begin{aligned} \text{NOT } 1 &= 0 \\ \text{NOT } 0 &= 1 \end{aligned}$$

Performing $\text{NOT } X$ gives us the *bit complement* of X .

Example 6.4:

Let $X = 10110$ and $Y = 11010$. To obtain $NOT\ X$, we flip each bit in X :

$$NOT\ X = 01001$$

Likewise, we can compute $NOT\ Y = 00101$.

Your turn!

Let $X = 11010010$ and $Y = 10111001$.

- a. Compute $NOT\ X$

Answer: $NOT\ X = 00101101$

- b. Compute $NOT\ Y$

Answer: $NOT\ Y = 01000110$

Combining the Operations: When combining the operations, we need to be careful with the order in which we perform them. The NOT operation takes highest priority. To compute $NOT\ X\ AND\ Z$, we interpret it as $(NOT\ X)\ AND\ Z$. AND, OR, and XOR are each commutative and associative. As always, perform the operation in parentheses first.

Let's try some examples. For each example, we will use the following strings of length 5:

$$X = 10110$$

$$Y = 11010$$

$$Z = 01101$$

Example 6.5:

Evaluate $(X\ AND\ Y)\ OR\ (NOT\ X\ AND\ Z)$.

Answer: To evaluate this, we will perform the NOT first, then the expressions in parentheses, and then finally the OR in the middle.

1. $NOT\ X = 01001$

2. Parentheses

a. $X\ AND\ Y = 10110\ AND\ 11010$:

$$\begin{array}{r} 10110 \\ AND\ 11010 \\ \hline 10010 \end{array}$$

b. $(NOT\ X)\ AND\ Z = 01001\ AND\ 01101$:

$$\begin{array}{r} 01001 \\ AND\ 01101 \\ \hline 01001 \end{array}$$

3. The OR: $(X\ AND\ Y)\ OR\ (NOT\ X\ AND\ Z) = 10010\ OR\ 01001$:

$$\begin{array}{r} 10010 \\ OR\ 01001 \\ \hline 11011 \end{array}$$

Example 6.6:

Evaluate $(X\ AND\ Z)\ OR\ (Y\ AND\ NOT\ Z)$.

Answer: Again, we must first find NOT Z: $NOT\ Z = 10010$.

For the parentheses,
 $X\ AND\ Z = 10110\ AND\ 01101$:

$$\begin{array}{r} 10110 \\ AND\ 01101 \\ \hline 00100 \end{array}$$

$Y\ AND\ (NOT\ Z) = 11010\ AND\ 10010$:

$$\begin{array}{r} 11010 \\ AND\ 10010 \\ \hline 10010 \end{array}$$

Now we can combine everything:
 $(X\ AND\ Z)\ OR\ (Y\ AND\ NOT\ Z) = 00100\ OR\ 10010$:

$$\begin{array}{r} 00100 \\ OR\ 10010 \\ \hline 10110 \end{array}$$

Example 6.7:

Evaluate $X\ XOR\ Y\ XOR\ Z$

Answer: Since XOR is associative, the grouping does not matter. We can either evaluate $(X \text{ XOR } Y) \text{ XOR } Z$ or we can evaluate $X \text{ XOR } (Y \text{ XOR } Z)$. Or we can do both. (That was a joke on the ambiguity of OR. If you didn't get the joke, take a few minutes to review the OR operation discussed a few pages back.) We will actually do both so that we can see that the grouping does not affect the final answer:

Option A: $(X \text{ XOR } Y) \text{ XOR } Z = 00001$

$$X \text{ XOR } Y = 10110 \text{ XOR } 11010:$$

$$\begin{array}{r} 10110 \\ \text{XOR } 11010 \\ \hline 01100 \end{array}$$

$$(X \text{ XOR } Y) \text{ XOR } Z = 01101 \text{ XOR } 01101:$$

$$\begin{array}{r} 01100 \\ \text{XOR } 01101 \\ \hline 00001 \end{array}$$

Option B: $X \text{ XOR } (Y \text{ XOR } Z) = 00001$

$$(Y \text{ XOR } Z) = 11010 \text{ XOR } 01101:$$

$$\begin{array}{r} 11010 \\ \text{XOR } 01101 \\ \hline 10111 \end{array}$$

$$X \text{ XOR } (Y \text{ XOR } Z) = 10110 \text{ XOR } 10111:$$

$$\begin{array}{r} 10110 \\ \text{XOR } 10111 \\ \hline 00001 \end{array}$$

Your turn!

Evaluate $Y \text{ XOR } (X \text{ OR } \text{NOT } Z)$

Answer: 01100

$$\begin{aligned} Y \text{ XOR } (X \text{ OR } \text{NOT } Z) &= 11010 \text{ XOR } (10110 \text{ OR } \text{NOT } 01101) \\ &= 11010 \text{ XOR } (10110 \text{ OR } 10010) \\ &= 11010 \text{ XOR } 10110 \\ &= 01100 \end{aligned}$$

Homework for Section 6

1. Let $X = 100110101$ and $Y = 010110111$.
 - a. Compute $X \text{ AND } Y$
 - b. Compute $X \text{ OR } Y$
 - c. Compute $X \text{ XOR } Y$
 - d. Compute $\text{NOT } X$
 - e. Compute $\text{NOT } Y$
2. Let $X = 10011$, $Y = 01011$, and $Z = 10010$. Evaluate $Y \text{ XOR } (X \text{ OR } \text{NOT } Z)$

Section 7: Hashing and the MD5 Algorithm

Cryptography versus Hashing:

When you make a secure purchase online, you want to be sure of two things:

1. Your credit card information is safe and secure as it is being transmitted to the website.
2. The website to which you are sending your credit card information is the actual website you intended to use.

These goals require two different types of secrecy and security.

Keeping your credit card information secure requires *encryption*. When we *encrypt* data, we transform the data in a way that keeps it secret from those who don't need to know it, but the data can be un-transformed by the intended recipient and read. *Cryptography* is the study of different methods of encryption. To determine whether the website is authentic, services like Verisign and Symantec will verify the website's *hash value*. A *hash function* gives us a way to see if data has been tampered with. It is a mathematical function that transforms data into a unique value, and small changes in the data should result in large changes to the value. To preserve the integrity of the data, hashing should not be reversible. If it were, fraudulent websites could reverse-engineer the function to give a "safe" value, and your credit card information would be in the hands of the wrong people.

In short, encryption should be reversible if you have the proper key, but hashing should not. We can think of a hash functions like hash browns: We recognize that the hash browns came from a potato that was shredded, but there is no way to reconstruct the original potato from the shredded bits. This is also related to the idea of invertible functions. The function $f(x) = 3x - 7$ could be an encryption function because its inverse $f^{-1}(x) = \frac{x+7}{3}$ is also a function. That is, we could use f to encrypt a message and use f^{-1} to decrypt the message. This same function would not work as a hash function because all changes in the data values give equally sized changes in the function output. The function $g(x) = x^2$ would not make a good encryption function because its inverse is not a function (it fails the Horizontal Line Test). For example, g would encrypt 2 as 4, but when we try to decrypt the 4, we're not sure if it came from 2 or -2. The function g would also make a poor hash function because both 2 and -2 give a function value of 4, which is contrary to our desire for a function that transforms each piece of data into a unique value.

Examples:

- i. Transforming a message by shifting each letter forward by one position in the alphabet is a form of encryption. With this, A becomes B, B becomes C, and ultimately Z becomes A. To reverse the process, you could shift each letter in the encrypted message backward by one position. A person who sees your encrypted

message would not be able to read the message without knowing how it was encrypted.

- ii. The check sums you computed in Section 2 are a mild hash function. The check sum is sensitive to changes in the data, which is why `frog` and `fr0g` give such different values. However, there are frequent *collisions* where two strings have the same check sum. For example, the `DIRTYROOM` and `DORMITORY` give a collision because they have the same checksum. We don't want collisions in good hash functions because we can't determine the true source of the value.
- iii. Writing symbols in ASCII, like you did in Section 4, is mild encryption. It does transform the data, but the method is not very secretive.
- iv. Many websites use a method of encryption called RSA. RSA is named for its creators, Ronald Rivest, Adi Shamir, and Leonard Adleman. It is an example of *public key cryptography* because it uses two secret keys, one that is made public, and another that is kept secret. RSA uses modular arithmetic and is secure because factoring large numbers is computationally difficult.
- v. We will focus on the MD5 Hashing Algorithm, which was a popular method of hashing.

MD5 is the fifth iteration of the Message Digest algorithm and was developed in 1991 by Ronald Rivest of MIT (and one of the creators of RSA) to replace MD4. The MD5 algorithm produces a 128-bit (16-byte) hash value, typically expressed as a 32 digit hexadecimal number. This hash value can be thought of as a checksum or digital fingerprint. Each file has a fairly unique MD5 hash and we can see if two files are identical by comparing their MD5 hash values. The slightest change in a file will result in very different MD5 hash values (as we'll see here). We'll also discuss the steps involved in this algorithm as part of this module.

Let's look at the steps of the algorithm:

STEP 1: Our algorithm begins by converting all of our text to ASCII as discussed in Section 4.

Let's remind ourselves of this process with an example.

Example 7.1:

Encode **THE** in ASCII.

Answer: 01010100 01001000 01000101

STEP 2: Our next step in the MD5 algorithm is to pad the message to get it to a set length (in bits).

The goal will be to get a message of length that is congruent to 448 (mod 512). Even if our message begins out being congruent to 448 (mod 512), we will still pad the message.

Consider the question: What is the difference between 512 and 448? **Answer: 64.**

To pad the message, begin by placing a 1 at the end of the message. Then, add as many 0s as are needed to get the message to be of length congruent to 448 (mod 512). So, in other words, you will always add a 1 and at least one 0.

Example 7.2:

If we encode **THE** in ASCII we get 01010100 01001000 01000101. How many bits need to be added in order to pad this message? In particular, how many 1s and 0s will that be?

Answer: Add 424 bits since the message is currently 24 bits. So, that's 1 one and 423 zeros.

Your turn!

(Note: this example has some lowercase letters which are not shown in the table in Section 4. However, their values can be easily looked up.)

If we encode **The fluffy dog walked across the street on its way home.** in ASCII we get:

01010100 01101000 01100101 00100000 01100110 01101100 01110101 01100110
01100110 01111001 00100000 01100100 01101111 01100111 00100000 01110111
01100001 01101100 01101011 01100101 01100100 00100000 01100001 01100011
01110010 01101111 01110011 01110011 00100000 01110100 01101000 01100101
00100000 01110011 01110100 01110010 01100101 01100101 01110100 00100000
01101111 01101110 00100000 01101001 01110100 01110011 00100000 01110111
01100001 01111001 00100000 01101000 01101111 01101101 01100101 00101110

How many bits need to be added in order to pad this message? In particular, how many 1s and 0s will that be?

Answer: Add 512 bits since the message is currently 448 bits. So, that's 1 one and 511 zeros.

STEP 3: As you saw in an earlier question, our padded messages are currently 64 bits shy of being congruent to 0 mod 512. We would like our final padded message to have length exactly congruent to 0 mod 512. We use those remaining 64 bits to append the length of the message.

To do this, we begin by counting the bit length of the original message (before we padded the message). We then add the 64-bit representation of that number to the end of the message. If the message happens to be bigger than 2^{64} bits long, then we only append the lower order bits of the bit-length.

When we append the bits, we list the **bytes** (i.e. 8 bits) in order from most significant to least significant. We do not reverse the order of the bits within the bytes; we only reverse the order of the bytes. For example, if our original message had bit length 16, we'd add 00010000 00000000 00000000 00000000 00000000 00000000 00000000 to the end of the padded message because 16 is represented with 7 bytes of 0s and then 00010000.

Example 7.3:

Giving the least significant bytes first, what would be the 64-bit representation of the length for a message consisting of 40 bits?

Answer: Note that $40 = 0101000_2$, thus if we make that 64-bits long we get,
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00101000.
Now, we reverse the order of the bytes. There are 8 bytes total. We begin with the last byte and rewrite from last byte to first byte:
00101000 00000000 00000000 00000000
00000000 00000000 00000000 00000000.

Your turn!

Giving the least significant bytes first, what would be the 64-bit representation of the length for a message consisting of 264 bits?

Answer: Noting that $264 = 00000001\ 00001000_2$ we get,
00000000 00000000 00000000 00000000
00000000 00000000 00000001 00001000
And now, reversing the byte order we get,
00001000 00000001 00000000 00000000
00000000 00000000 00000000 00000000.

Example 7.4:

Add the bit length to these messages and write down the final padded message (after padding from step 2 and step 3).

01010100 01001000 01000101

Answer:

01010100 01001000 01000101 10000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00011000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Your Turn!

Add the bit length to this message and write down the final padded message (after padding from step 2 and step 3).

01010100 01101000 01100101 00100000 01100110 01101100 01110101 01100110
01100110 01111001 00100000 01100100 01101111 01100111 00100000 01110111
01100001 01101100 01101011 01100101 01100100 00100000 01100001 01100011
01110010 01101111 01110011 01110011 00100000 01110100 01101000 01100101
00100000 01110011 01110100 01110010 01100101 01100101 01110100 00100000
01101111 01101110 00100000 01101001 01110100 01110011 00100000 01110111
01100001 01111001 00100000 01101000 01101111 01101101 01100101 00101110

Answer:

01010100 01101000 01100101 00100000 01100110 01101100 01110101 01100110
01100110 01111001 00100000 01100100 01101111 01100111 00100000 01110111
01100001 01101100 01101011 01100101 01100100 00100000 01100001 01100011
01110010 01101111 01110011 01110011 00100000 01110100 01101000 01100101
00100000 01110011 01110100 01110010 01100101 01100101 01110100 00100000
01101111 01101110 00100000 01101001 01110100 01110011 00100000 01110111
01100001 01111001 00100000 01101000 01101111 01101101 01100101 00101110
10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
11000000 00000001 00000000 00000000 00000000 00000000 00000000 00000000

In order to make this process easier, an online MD5 demo that performs all these steps can be found at <https://sites.google.com/a/falcons.fitchburgstate.edu/cbuell/home>.

Example 7.5:

Try the first three steps of this algorithm (converting to ASCII, padding the message and adding the length of the message) on the message **One if by land.** using the online MD5 demo and copy and paste your results below.

Answer:

01001111 01101110 01100101 00100000 01101001 01100110 00100000 01100010
01111001 00100000 01101100 01100001 01101110 01100100 00101110 10000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
01111000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Your Turn!

Try the first three steps of this algorithm (converting to ASCII, padding the message and adding the length of the message) on the message **PaSsWord** using the online MD5 demo and copy and paste your results below.

Answer:

01010000 01100001 01010011 01110011 01010111 01101111 01110010 01100100
10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

STEP 4: We now need to initialize the four words that will form our final MD5 output. These initialized values are always the same and are given as hexadecimal values, which you studied in section 5. Recall that the lowercase letters represent hexadecimal digits:

$$a = 10, b = 11, c = 12, d = 13, e = 14, \text{ and } f = 15.$$

Let

$A = 01\ 23\ 45\ 67$

$B = 89\ ab\ cd\ ef$

$C = fe\ dc\ ba\ 98$

$D = 76\ 54\ 32\ 10$

Notice the symmetry involved in defining these initial words. Also, note that the MD5 algorithm reads these strings in reverse pair-wise order. So, for example A is treated as 67452301 throughout the algorithm.

STEP 5: The first part of this step is to break our message into blocks of 512 bits each.

Example 7.5:

How many blocks are there for each of the examples we did previously? Keep in mind this calculation is done after the padding done in steps 2 and 3.

a) THE

01010100 01001000 01000101

Answer: 1

b) The fluffy dog walked across the street on its way home.

00001010 00001001 01010100 01101000 01100101 00100000 01101000 01100001
01110000 01110000 01111001 00100000 01100100 01101111 01100111 00100000
01110111 01100001 01101100 01101011 01100101 01100100 00100000 01100001
01100011 01110010 01101111 01110011 01110011 00100000 01110100 01101000
01100101 00100000 01110011 01110100 01110010 01100101 01100101 01110100
00100000 01101111 01101110 00100000 01101001 01110100 01110011 00100000
01110111 01100001 01111001 00100000 01101000 01101111 01101101 01100101

Answer: 2

c) What a life.

01010111 01101000 01100001 01110100 00100000 01100001 00100000 01101100
01101001 01100110 01100101 00101110

Answer: 1

We will now run the entire MD5 algorithm for these three examples. We'll go through the entire MD5 algorithm just once for Examples 1 and 3 since each example has only one block of length 512 bits. However, in Example 2, we'll use MD5 twice, since there are two blocks.

STEP 6: Next, we define four functions using the binary operations we learned about in Section 7.

$F(X,Y,Z) = (X \text{ AND } Y) \text{ OR } (\text{NOT } X \text{ AND } Z)$

$G(X,Y,Z) = (X \text{ AND } Z) \text{ OR } (Y \text{ AND } \text{NOT } Z)$

$H(X,Y,Z) = X \text{ XOR } Y \text{ XOR } Z$

$I(X,Y,Z) = Y \text{ XOR } (X \text{ OR } \text{NOT } Z)$

One other binary operation the algorithm uses will be \lll . This is often written as $X \lll s$ where X is a binary string and s is an integer. This shifts all the digits of X to the left by s units, filling in any holes on the right with zeros and dropping all the extra numbers shifted past the first original digit.

Example 7.6:

Perform the following shift operation: 00001010 00001001 01010100 01101000 $\lll 15$

Answer: 10101010 00110100 00000000 00000000

Your turn!

Perform the following shift operations:

a) 01010111 01101000 01100001 01110100 $\lll 22$

b) 01101000 01101111 01101101 01100101 $\lll 7$

Answer:

a) 01011101 00000000 00000000 00000000

b) 00110111 10110110 10110010 10000000

Now, we'll create a function T which takes any of the numbers 1 through 64 as inputs and has the integer part of $T[i] = 2^{32} |\sin(i)|$ (where i is in radians) as the output. This function helps scramble the bits.

Example 7.7: Calculate $T[1]$.

Answer: $T[1] = 3,614,090,360$

Your Turn! Calculate $T[32]$ and $T[64]$.

Answer: $T[32] = 2,368,359,562$ and $T[64] = 3,860,291,034$

STEP 7: We are now ready to see the 4-round algorithm that is used to create the MD5 hash. The following pseudocode comes directly from Ron Rivest's original memo on the MD5 hash in 1992:

Do the following:

```
/* Process each 16-word block. */  
For i = 0 to N/16-1 do
```

```
  /* Copy block i into X. */  
  For j = 0 to 15 do  
    Set  $X[j]$  to  $M[i*16+j]$ .  
  end /* of loop on j */
```

```
  /* Save A as AA, B as BB, C as CC, and D as DD. */  
  AA = A  
  BB = B  
  CC = C  
  DD = D
```

```
  /* Round 1. */
```

```
  /* Let [abcd k s i] denote the operation  
   $a = b + ((a + F(b,c,d) + X[k] + T[i]) \lll s)$ . */
```

```
  /* Do the following 16 operations. */
```

```
  [ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4] [ABCD 4 7 5] [DABC 5 12 6]  
  [CDAB 6 17 7] [BCDA 7 22 8] [ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22  
  12] [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
```

```
  /* Round 2. */
```

```
  /* Let [abcd k s i] denote the operation  
   $a = b + ((a + G(b,c,d) + X[k] + T[i]) \lll s)$ . */
```



```
/* Do the following 16 operations. */
```

```
[ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20] [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24] [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28] [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
```

```
/* Round 3. */
```

```
/* Let [abcd k s t] denote the operation
```

```
a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
```

```
/* Do the following 16 operations. */
```

```
[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36] [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40] [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44] [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
```

```
/* Round 4. */
```

```
/* Let [abcd k s t] denote the operation
```

```
a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
```

```
/* Do the following 16 operations. */
```

```
[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52] [ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56] [ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60] [ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]
```

```
/* Then perform the following additions. (That is increment each of the four registers by the value it had before this block was started.) */
```

```
A = A + AA
```

```
B = B + BB
```

```
C = C + CC
```

```
D = D + DD
```

```
end /* of loop on i */
```

So, each time we send a 512-bit block word through the algorithm, we'll get a new A, B, C, and D and we'll use those for the next 512-bit block word. We'll do this until all of the 512-bit block words have been done. Note that inside of this algorithm we use binary values, bit operations, and then convert back to hexadecimal. These are all concepts we learned in previous sections.

Finally, we need to report the outcome of the MD5 algorithm. You will have ABCD, but we don't forget about the need to report the output with low-order byte first.

Example 7.8:

Let's try converting the final MD5 output to the final result of MD5. Suppose after all rounds of MD5, A=ab 12 c7 4d, B=01 34 78 21, C=71 04 ad 11, and D=90 45 ab cd. What will be the final result of MD5?

Answer: 4dc712ab2178340111ad0471cdab4590

Homework Exercises Section 7

1. Encode **What a life.** in ASCII. Then, determine how many bits need to be added in order to pad this message? In particular, how many 1s and 0s will that be?
2. Giving the least significant bytes first, what would be the 64-bit representation of the length for a message consisting of 120 bits?
3. Add the bit length to this messages and write down the final padded message (after padding from step 2 and step 3).

01010111 01101000 01100001 01110100 00100000 01100001 00100000 01101100
01101001 01100110 01100101 00101110

4. Create your own word/phrase. Use the online MD5 demo to record the output after the first three steps of the MD5 algorithm (converting to ASCII, padding the message and adding the length of the message) and record its final MD5 hash value. The online demo can be found here: <https://sites.google.com/a/falcons.fitchburgstate.edu/cbuell/home>.
5. Perform the following shift operations:
 - a. 01011101 00001111 00101010 11100011 <<< 17
 - b. 00000001 11111111 11101011 <<< 3
6. Recall that $T[i]=2^{32}|\sin(i)|$ (where i is measured in radians). Calculate $T[48]$.

Section 8: Uses of MD5 and Future Hashing Functions

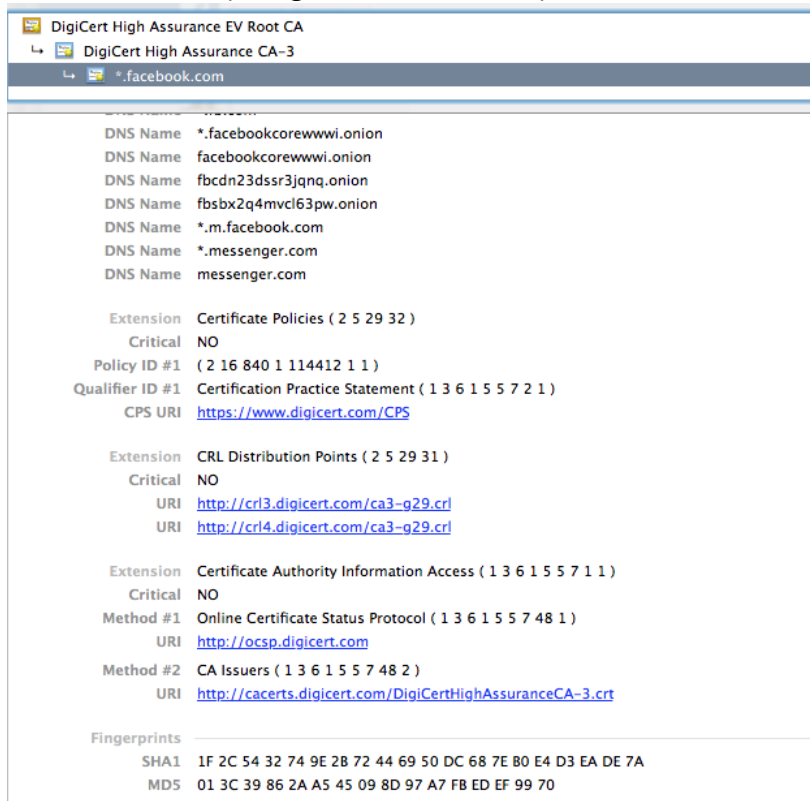
MD5 has been used in a variety of cryptographic applications including data integrity verification. In particular, suppose you want to make sure that a transferred file has arrived intact. A pre-computed MD5 checksum will also be attached to a file stored on a file server, and after downloading, the user can compare the checksum of the downloaded file to the checksum for the stored file to validate the correct file has been downloaded. MD5 is also used to store passwords and to provide a unique identifier for electronically-transmitted documents.

One common use of this algorithm is to compute MD5 hash values for files. The great thing about the MD5 hash is that even a tiny change in a file will result in a drastic change in the MD5 hash value.

Your Turn!

Try this for yourself with a Word file and this free app: <http://www.winmd5.com/>. Begin by calculating the MD5 hash for your file. Then, open the file and make a slight change. See what happens to the MD5 hash after the change. Report the two hash values. If you undo the change, do you get back to the original hash value?

In addition, MD5 hash values are still given on some SSL certificates if you examine them in certain browsers. For example, if you examine Facebook's SSL certificate in Safari, you'll see its MD5 hash value (along with a few others).



DNS Name	*.facebookcorewwwi.onion
DNS Name	facebookcorewwwi.onion
DNS Name	fbcdn23dsr3jqnq.onion
DNS Name	fbsbx2q4mvl63pw.onion
DNS Name	*.m.facebook.com
DNS Name	*.messenger.com
DNS Name	messenger.com
Extension	Certificate Policies (2 5 29 32)
Critical	NO
Policy ID #1	(2 16 840 1 114412 1 1)
Qualifier ID #1	Certification Practice Statement (1 3 6 1 5 5 7 2 1)
CPS URI	https://www.digicert.com/CPS
Extension	CRL Distribution Points (2 5 29 31)
Critical	NO
URI	http://crl3.digicert.com/ca3-g29.crl
URI	http://crl4.digicert.com/ca3-g29.crl
Extension	Certificate Authority Information Access (1 3 6 1 5 5 7 1 1)
Critical	NO
Method #1	Online Certificate Status Protocol (1 3 6 1 5 5 7 48 1)
URI	http://ocsp.digicert.com
Method #2	CA Issuers (1 3 6 1 5 5 7 48 2)
URI	http://cacerts.digicert.com/DigiCertHighAssuranceCA-3.crt
Fingerprints	
SHA1	1F 2C 54 32 74 9E 28 72 44 69 50 DC 68 7E B0 E4 D3 EA DE 7A
MD5	01 3C 39 86 2A A5 45 09 8D 97 A7 FB ED EF 99 70

The Future of Hash Functions

Starting in 1996, weaknesses in MD5 known as collisions were discovered. A collision occurs when two files give the same hash value. A good hash function will make finding two such files very difficult. However, in 2005 Xiaoyun Wang and Hongbo Yu found an algorithm that will produce two files with very small differences that produce the same MD5 hash value.

Currently, despite the collision concerns MD5 is still used in some government applications. However, it has been phased out of online security certificates. It has been replaced by SHA-1, another hash function. In fact, SHA stands for “Secure Hash Algorithm.” However, SHA-1 has also been shown to have collisions and is thus no longer useful in the internet security world; it will be phased out by 2017 and replaced by SHA-2 or other more complicated versions of SHA.

References

Burger, Edward B. "The Heart of Mathematics, An Invitation to Effective Thinking." John Wiley & Sons Inc, 2009.

Fay-Wolfe, Victor. "Digital Signature (Hashing) Introduction." PDF slides presented at RECONNECT 2014.

Rivest, Ronald. "The MD5 Message-Digest Algorithm." <https://www.ietf.org/rfc/rfc1321.txt>, April 1992.

Selinger, Peter. "MD5 Collision Demo." <http://www.mathstat.dal.ca/~selinger/md5collision/>, February 2006.

Appendix: Extended Activities

Extended Activity A:

Consider some reasons that the banks might use 7, 3, and 9 in their calculations when working mod 10 but not use 2,4,5, etc?

Recall the hints/ideas:

1. What would happen if the previous problem came down to solving $8 + 4d \equiv 0 \pmod{10}$ instead of $8 + 7d \equiv 0 \pmod{10}$?
2. The main idea here is the idea of *relatively prime* numbers. In the example, 7, 3 and 9 are all relatively prime to 10.

For Hint/Idea 1, you may have noticed that there are multiple solutions. Also, if the problem came down to $7 + 4d \equiv 0 \pmod{10}$, then there would be no solutions! The reason there are multiple answers or no answer is because 10 and 4 share a common factor whereas 10 and 7, 10 and 3, and 10 and 9 do not share a common factor. When two integers m and n have no common factors (besides 1), we say m and n are *relatively prime*

We can see a pattern in the multiplication tables. The first table is a basic multiplication table and the second table is a multiplication table **mod 10**.

x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

x (mod 10)	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

There are two important aspects to the second table to note:

1. In rows where the value is relatively prime to 10 (these are 1,3,7, and 9) every value between 0 and 9 appears. In rows where the value is not relatively prime to 10 (these are 0,2,4,5,6, and 8), not all values between 0 and 9 will appear. For example, only the values 0,2,4,6, and 8 appear in some rows and only 0 and 5 appear in row 5.
2. In rows where the value is relatively prime to 10 (these are 1,3,7, and 9) each value appears exactly once. In rows where the value is not relatively prime to 10 (these are 0,2,4,5,6, and 8), there are doubles or four copies of particular values. For example, 5 appears four times in row 5 because $5 * 2 = 10 \equiv 0 \pmod{10}$, $5 * 4 = 20 \equiv 0 \pmod{10}$, $5 * 6 = 30 \equiv 0 \pmod{10}$, and $5 * 8 = 40 \equiv 0 \pmod{10}$.

These two facts explain why UPC, bank accounts, and other check digit systems use relatively prime numbers (even more complex systems like RSA use relatively prime numbers in the algorithm). These cases always need an answer and always need a **unique** check digit to work and to check to make sure the product information is correct or the bank account number was entered correctly.

In all our examples we solve an equation of the form: $md + c = 0 \pmod{10}$ or rather $md = -c = 0 \pmod{10}$. Note that $-c \pmod{10}$ can be any value between 0 and 9. This means $md \pmod{10}$ needs to have a solution for every value between 0 and 9 and d is any value. If m is not relatively prime to 10, then either there is no solution to this equation (say c is 7 and m is 2) or there are many solutions (say $c = 5$ and $m = 5$).

These concepts are connected to advanced topics in mathematics like number theory, rings, fields, and abstract algebra. Prime numbers and relatively prime numbers are found throughout mathematics, computer science, and security.

Extended Activity B:

While we didn't discuss multiplication of binary numbers in the module, we can still multiply values in binary. There is a rote method for multiplying in base-ten; however, this rote method is difficult to do in binary. We will consider the grouping method or a modified area model concept.

In Section 3, we multiplied 16 and 31 using a tabular method. We can also write out this multiplication in two ways:

$$16 * 31 = (10 + 6) * (30 + 1) = (300 + 10 + 180 + 6) = 496$$

or

$$16 * 31 = (10 + 6) * (31) = (310 + 186) = 496.$$

Note that the second version is not as intuitive because we do not have 6 times 31 as a known multiplication fact. While a bit strange to see it in a decimal expansion, we can perform the above operations as follows:

$$\begin{aligned} 16 * 31 &= (1 * 10^1 + 6 * 10^0) * (3 * 10^1 + 1 * 10^0) \\ &= 3 * 10^2 + 1 * 10^1 + 18 * 10^1 + 6 * 10^0 \end{aligned}$$

or

$$16 * 31 = (1 * 10^1 + 6 * 10^0) * (31) = 31 * 10^1 + 186 * 10^0$$

Upon regrouping, we have:

$$\begin{aligned} &3 * 10^2 + 1 * 10^1 + 18 * 10^1 + 6 * 10^0 \\ &= 3 * 10^2 + 19 * 10^1 + 6 * 10^0 \\ &= 3 * 10^2 + (10 * 10^1 + 9 * 10^1) + 6 * 10^0 \\ &= 3 * 10^2 + 1 * 10^2 + 9 * 10^1 + 6 * 10^0 \\ &= 4 * 10^2 + 9 * 10^1 + 6 * 10^0 = 496 \end{aligned}$$

or

$$\begin{aligned} &31 * 10^1 + 186 * 10^0 \\ &= 31 * 10^1 + (180 * 10^0 + 6 * 10^0) \\ &= 31 * 10^1 + 18 * 10^1 + 6 * 10^0 \\ &= 49 * 10^1 + 6 * 10^0 \\ &= (40 * 10^1 + 9 * 10^1) + 6 * 10^0 \\ &= 4 * 10^2 + 9 * 10^1 + 6 * 10^0 = 496 \end{aligned}$$

Now, clearly the first method was more effective but the second will illuminate what happens in base-two. Let's write both 16 and 31 in their binary expansions before we perform the multiplication.

$$16 * 31 = (10000)_2 * (11111)_2$$

$$= (1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0) * (1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0)$$

While this expansion looks complicated, we can see that many terms will be zero. That is a perk of binary expansion: every coefficient is either 0 or 1.

$$16 * 31 = (10000)_2 * (11111)_2$$

$$= (1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0) * (1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0)$$

$$= (2^4) * (1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0) = 496$$

Really this last line just says $(2^4) * 31 = 496$, which just says double 31 four times. This matches our tabular conclusion in Section 3. We will see in the next example that we are only using the binary expansion of the first number.

Now, let's take consider more complicated examples, meaning some rows are not crossed out:

Example 1	
19	13
9	26
4	52
2	104
1	208

Example 2	
20	13
10	26
5	52
2	104
1	208

In Example 1, $19 * 13$, the 3rd and 4th rows are crossed out. So we see that

$$19 * 13 = 13 + 26 + 208 = 247.$$

Let's interpret this result in binary:

$$19 * 13 = (2^4 + 2^1 + 2^0) * (13)$$

Looking at the 19, we want to make sense of the binary expansion from the table. We will discuss this for any value m , but demonstrate the pattern with 19. Suppose m was already written in base-two. Asking whether m is even or odd is just asking whether m has a 2^0 term. If so, then keep that row. Otherwise cross it out. Notice that by keeping the row we are telling ourselves that we will have a 2^0 in the binary expansion. This 2^0 will need to be multiplied to the other value, in this case, 13. We add 13 to our total value because $13 = 2^0 * 13 = 1 * 13$.

Then we divide m by two and determine whether the new value is even or odd. In base-two, dividing by two is merely a matter of subtracting 1 from every exponent:

$$\frac{(2^4 + 2^1 + 2^0)}{2} = 2^3 + 2^0 + 2^{-1}.$$

Since we made a rule of rounding down, the value here would be $2^3 + 2^0$. Again, the value is odd, so we should keep this term. The value being odd says that we must have had a 2^1 term in m , because dividing by two and having a 2^0 term means you had a 2^1 in the first place. So, we

will need to multiply 2 times our value, here 13. That is why we have a 26 in our total. Another way to say this is that we needed to double 13 once because we have a 2^1 term.

We continue this process of assessing whether the binary expansion of the number has a 0 term or a 1 term at the 2^n position by assessing even or odd values. If the value exists in the binary expansion in position n , then we know it will be part of our total and that we will need the value doubled n times.

n	2^n term	m	
0	yes	19	$2^0 * 13$
1	yes	9	$2^1 * 13$
2	no	4	$2^2 * 13$
3	no	2	$2^3 * 13$
4	yes	1	$2^4 * 13$

Example 2	
20	13
10	26
5	52
2	104
1	208

In Example 2, $20 * 13$, the 1st, 2nd, and 4th rows are crossed out. Then
 $20 * 13 = 52 + 208 = 260$.

Let's interpret this result in binary:

$$20 * 13 = (2^4 + 2^2) * (13).$$

We can read this as needing 13 doubled 4 times and 13 doubled twice, 208 and 52 respectively. There is a tendency to want to say 4 times 13 and 2 times 13, but that is not correct because we are counting the number of times we double since it is $2 * 2 * 2 * 2 * 13$, etc.