Chapter 1.2 Measuring and Coding Information

Measuring information

Information seems such a vague and subjective concept that it may be difficult to imagine how it could be measured. The method is part compromise and part mathematics. The compromise is to define all information as a series of choices between alternatives, and to consider the choices to be the information. To ignore the content of the alternatives may seem too limiting, but we will see that this simplification provides a tool allowing many insights into the performance of communication and information systems.

Specifically, information is defined as a series of events, devices, signs, symbols or tokens that allow a choice between previously specified alternatives. Thus, information is defined in terms of function, what it enables us to do. The function of information is to only make choices between defined alternatives. What the alternatives are, what they mean, and how they are in turn defined has now been separated from the information used to make the choices.

While this abstraction may at first seem forced or artificial, it is perhaps no more abstract than the concept of quantity or number itself. When children are first introduced to numbers the numbers are linked to a specific objects, e.g. one apple, two apples, three apples. However, after a specific number has been used to refer to the same quantity of many different kinds of objects, e.g. two apples, two dogs, two boats, children begin to associate the number with the abstract concept of quantity, and not the objects themselves.

The bit: the unit of information

If information is defined as the ability to make choices, the smallest amount of information would be that which enables you to make the most simple choice: the decision between two alternatives. The information needed to make such a choice is called a bit, a token which has two possible states. You can call the two states + and -, yes and no, 1 and 0, or A and T; whatever pleases you. Of course most decisions are more complex than choosing between two alternatives, but as we will see a decision of arbitrary complexity can be accomplished by the sequential use, or sum, of a series of bits.

Suppose we need to choose among four alternatives, call them A, B, C and D. One way to accomplish this is to first chose between the A-B pair and the C-D pair. This takes one bit of information. Then we need to choose between A and B, or C and D, depending on what the first decision was. This second decision takes one bit. Thus we have picked one out of four choices with two bits. While there are several ways of making the final choice, they all take two bits of information. Extending this method, we can decide among 8 choices with 3 bits, 16 choices with 4 bits, 32 choices with 5

bits, and so on. Each time the total number of choices is doubled, only one more bit is needed to make the final choice.

A series of numbers in which each is a constant multiple of the preceding is an exponentially increasing series. If the first number in the series is the multiplier, the nth member is said to be that number raised to the power n. Specifically, the nth term in the series of the preceding paragraph, 2, 4, 8, ... is 2^n., two raised to the power n. Two to the third power is eight. The process of raising a number to a power, exponentiation, is related to multiplication in the same way that multiplication is related to addition. Multiplication of n by m means that n is added to itself m times. Raising n to the m power means that n is multiplied times itself m times. The inverse of raising a number to a power is to find its logarithm. The logarithm of 8 to the base 2, is 3. Thus, the number of bits needed to make n choices is the logarithm of n to the base 2.

Figure Exponents

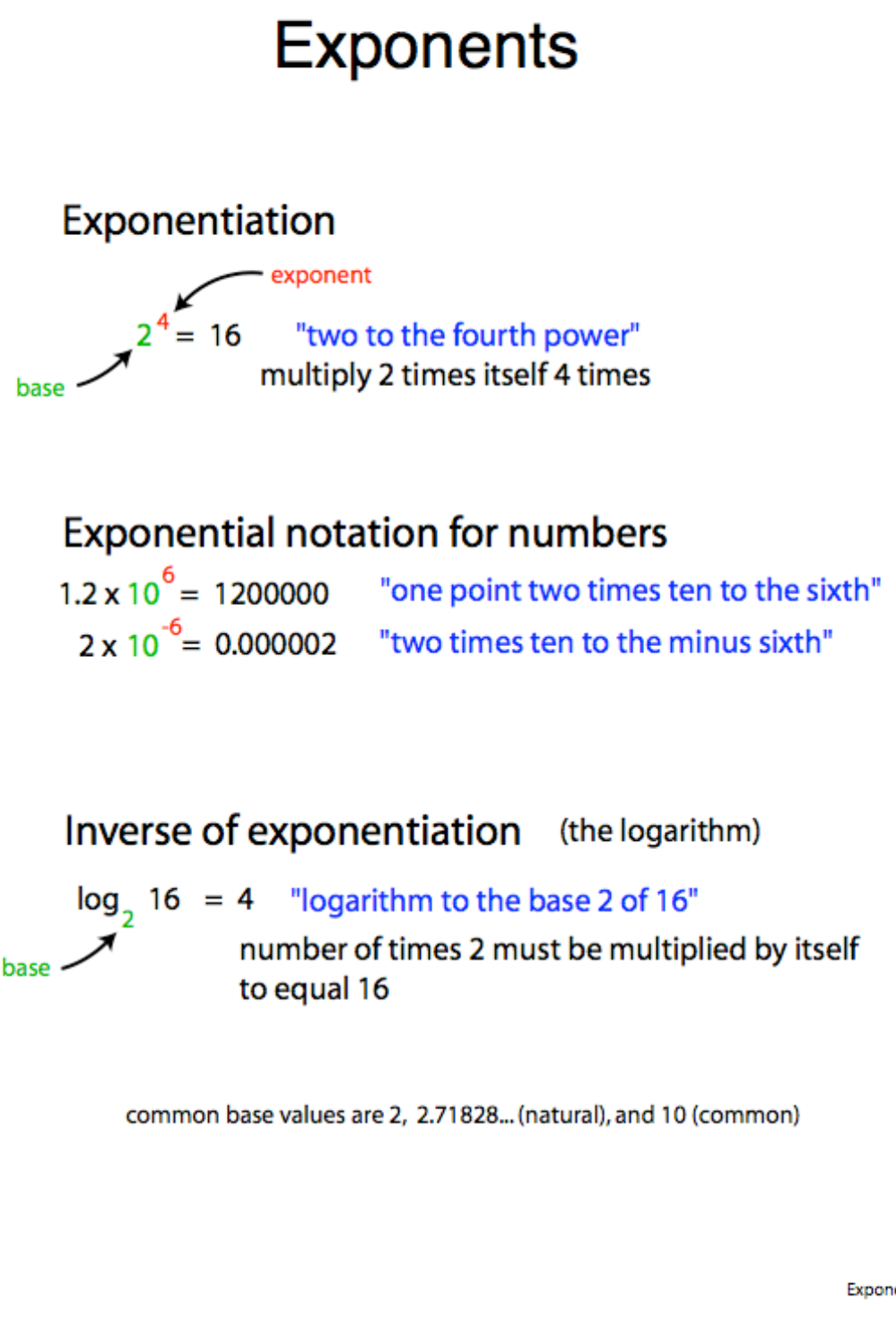


Figure Exponent. An exponent is a shorthand way of indicating repeated multiplication of a number by itself, e.g. 2^4 means 2x2x2x2. A negative exponent indicates the reciprocal , e.g. 2^{-4} is 1/ 2^4. The logarithm of a number is the number of times the base number must be multiplied by itself to equal the number. In information theory the base is 2. Other common bases for logarithms are 2.71828… (natural logarithms) and 10 (common logarithms).

If we are using tokens that have more than two states or values, then each token is worth more than one bit of information. For example, if the token is a numerical digit which can have a value between 0 and 7, it carries 3 bits of information since there are 8 possible states. If the digit can have values between 0 and 9, it represents slightly more information, 3.23 to be more exact. If we use the numerical digits 0 to 9 plus all the letters of the English alphabet for a token there will 36 possible values. The information that can be transmitted by one of these tokens is 5.17 bits. This is one of the motivations for using letters and numbers in automobile license plates, you need a smaller number of symbols to specify a large number.

Are bits worth the trouble?

The skeptic will point out that it not necessary to use bits in order to deduce that fewer symbols are required to uniquely label one million license plates if both numbers and letters are used as opposed to using numbers alone. The simple argument is that it takes six digits (in the range 0 to 10) to count to one million (000000 to 999999), while it takes less than four symbols if each symbol has 36 possible values, since 36 X 36 X 36 X 36 = 1,679,616.

The advantage of thinking in bits is that the information content of a string of symbols is simply the sum of the information in each of the symbols. As comparisons of strings of different symbols and various operations on the transmission and reading of those symbols become more complex, the advantages of using the concept of bits increases. Trust me.

Information in a string of symbols

If you are shown a string of symbols in an unfamiliar language or code, you will of course not be able to determine the content. However, you also can't easily calculate how much information is in the message, even using our constrained definition of information. Why can't you just determine the number of bits in each symbol and multiply that by the number of symbols in the string to get the total information?

The first problem is that you may not know how many possible values the symbols can have. Suppose it is short text in Latin, but you aren't familiar with Latin. It looks like the English alphabet is being used, and you thus assume that each letter can be one of 26 possible types, not knowing that Latin does not use the letter v. If the message is in English, but you don't know the English alphabet, you might not come across the letter z. and thus think there are only 25 possibilities. The longer the string the more likely you can guess the total number of different symbols, but you can never be sure if you don't know the language or code being used.

There is a more important but subtle problem A very basic law in information theory is that the occurrence of symbols in a string must appear to be random if the code used carries the maximum amount of information per symbol. The definition of random, and more specifically methods for proving that a string or anything else is random, are deep and important philosophical and mathematical questions, but we all have an intuitive idea of what random is. A string of symbols is random if we have no way of predicting in advance the symbols in the string, which is a way of saying that there are no patterns in the string that are independent of the message. The string must appear to be random when we don't know the message.

The reason we must not be able to predict the content of the string is actually obvious when you think about it. Any predetermined pattern in the string restricts the choices the sender can use to specify the message. If there are fewer patterns of symbols that can be used by the sender, the string must be longer to contain the same message. Thus, the formula given in the previous section for the number of bits a string of symbols can carry is actually just the maximum possible, not necessarily the amount in a real string.

English as an example

Since there are 26 letters used to write English text, each could represent 4.7 bits. But, in real English text (as opposed to an artificial code written in English characters) each letter is not independent of its neighbors. Each group of letters in English must be a word, but many combinations of letters are not words. There are many rules of grammar, e.g. sentences must contain a verb. Finally, there are strong frequency preferences e.g. "e" is used far more frequently than "z", what ever the message is. These patterns lower the average information density of this or any other English sentence to a value far lower that what the simple formula would predict. Thus, English text can be coded by a string of English letters typically 2 to 3 times shorter than the plain text, and this is sometimes done before English text is stored or transmitted by a computer.

Redundancy and error correction

Error correction is an essential process for both the Internet and life. Most of the machinery we will discuss later makes far too many errors for the systems they power to function without error correction, and typically several layers of error handling are used.

The most primitive error correction method is redundancy, and the most primitive redundancy is repetition. If a message is repeated and the two copies do not agree a request for a new copy can be made, and the two out of three that agree must be the correct message. A more complicated example is English text, which as we have noted earlier, does not send information in the most compact or efficient code. However, the rather verbose nature of English is also an advantage, since a single mistake is usually (but not always) detected, and often the correct text can be deduced so that the message does not have to be sent again. Often error detection and error correction are separate processes.

An example: check sums

The design and implementation of error detection and correction is an entire subfield of communication design. The many different strategies for error detection each have advantages and disadvantages. Thus the design goal is to match the error handling methods to the specific characteristics and requirements of the system. However, there is one simple scheme that is so common that I describe it here as an example.

The goal is error detection and the method consists in computing a check sum which is transmitted along with the message. The check sum is determined by giving each character a numerical value. With English text the value could be the position of the letter in the alphabet, i.e. a=1, b=2 ...At the end of the message, or paragraph, or sentence, the sum of the values of the letters is determined. This is the check sum. The recipient repeats the calculation of the check sum using the transmitted message, and if this computed value equals the transmitted one the message is assumed to be correct. If the message is incorrect, a request is made to send the message again. This method is very efficient, since symbols needed to code the check sum is usually insignificant in comparison to the message itself. Of course it only detects an error, the message needs to be retransmitted if one is found. However, if the error rate is low, retransmission needs to be done infrequently, and thus the strategy can be very efficient.

Codes and formats

In order to use a string of symbols to make choices you need a method for associating the characters with specific choices. The association or mapping of symbols to alternatives is called a code. A related concept is a format: a set of rules that specifies the position, organization or size of symbols. Often a format is part of a code because it indicates how the symbols are to be interpreted. However, there are formats which do not convey information. One example is alphabetical order. If we have a list of words in random order, and then rearrange the words to produce a list in which they are in alphabetical order, we have not added information to the list. That is because the rules for arranging a list in alphabetical order are commonly known, and can be done by anyone to any list with no special knowledge about what the list is intended to code. This does not mean that it may not be useful to arrange a list in alphabetical order, just that the rearrangement does not add information.

Roman and Arabic numbers: examples of formatted codes

Familiar examples of two codes in which a positional format is part of the code are the Roman and Arabic method of representing numbers. A code for numbers, here more specifically integers, is a code that enables you to chose between all possible integers. The number twenty-four is encoded very differently in the two systems.

Twenty four in the Roman code is "XXIV". The symbols are most easily read left to right. In this direction, as long as symbols code for equal or smaller values, the numbers represented by the symbols are just added to give the final number. All

numbers could be encoded in this manner, but in order to make the code more compact you can place a symbol for a smaller number before a symbol for a larger one, in which case the value of the smaller is subtracted from the number. The symbol “X” means ten, “V” is five, and "I" is one. Thus the example Roman number is ten plus ten plus five minus one, or twenty-four. To code for large numbers, e.g. the number of years since the birth of Christ, you need additional symbols, e.g. "C" for one hundred. I don't think Romans wrote really large numbers.

Twenty four in the Arabic code is "24". The symbols are most easy read right to left. In this direction the first symbol, “4” means four. The second symbol, “2” is two, but because it is in the second position, it must be multiplied by ten and then added to the number. Thus this collection of symbols means twenty-four. The absolute position of each symbol is essential for its interpretation, and thus there must be a means of indicating position, whatever the value of the other symbols in the number. This means that there must be a symbol for zero. If there were no zero, how would you indicate twenty? One of the advantages of the Arabic system is that position is efficiently used to transmit information. Thus, only the symbols 0-9 are required to code even very large numbers. Another advantage of the Arabic system is that it is easier to specify algorithms for addition, subtraction, multiplication and division. However, to use the Arabic code you have to understand the process of multiplication, at least implicitly, while the Roman code uses only addition and subtraction to combine the symbols.

Information storage in computers

Information can be stored in several types of devices in a computer. Examples are a register in the CPU (central processing unit), an address in a RAM (random access memory) or a track on a magnetic disk. However, in all commonly used devices, the fundamental units have only two states, and thus store one bit of information. The usual symbols used to represent the two states are 0 and 1. The use of these two symbols is just convention; you could just as well use "A" and "B" or "up" and "down" to represent the two states of the device. Thus, there is no special relation between computers and numbers. Computers have no “knowledge” of numbers, only of states. What you call those states is up to you. There is also no fundamental reason why memory devices can not have more than two states. If the device had more than two states it would take less units to represent a given amount of information. If you can invent a cost effective way to make memory that has more than two states, do it now, and be sure to get a patent. You can then finish this book from the deck of your very large yacht.

Addresses, bytes, and words

Information is stored and retrieved in two ways in electronic devices, sequential and addressable. Data is stored in a sequential device as one long string of bits. The data may be divided into groups, with perhaps a predefined number of bits in each group, or there can be coded tags which indicate the start and end of the groups. In any case, to find data the string of bits must be read from the beginning until the

desired data is found; there is no other way of getting it. Writing and reading data on magnetic tape is an example sequential process.

In contrast, data can be obtained from addressable memory by sending the address for that data, no searching is required. If the typical unit of data handled by the computer is small it is best for each addressable unit to be small, while if the typical unit of data is large, it would be best to have large units at each address. The usual compromise for the addressable unit size is 8 bits, a byte. Since one hexadecimal symbol represents four bits, a byte can be represented by two hexadecimal symbol.

However, more than one byte of information is often required to accomplish even a basic task. As an example, the address for one byte in a typical personal computer memory is made up of four bytes[1]. Consistent with the need to work with large units of data, each register in a CPU typically holds four bytes, and some handle eight. The size of the CPU registers is often called a word. To speed information transfer from memory to CPU and back one or more words is moved at each cycle, even if all the information is not used.

Codes for numbers

Since the basic information processing devices in computers have only two states, it is natural to represent information using a binary format, e.g. 00110111. This string means that there is a group of 8 binary devices which have the indicated states. The device on the left is defined as the most significant, which is another way of saying that the symbols are to be read from left to right. There is no way to know from this representation what this string means. It could be a number (as we will see there are many number codes), or a string of letters, or part of an image, or a brief segment of audio. Eight bits is a very common unit of information for computers; this unit is called a byte.

It is tedious to write any significant amount of information in a binary representation; it just takes up too much space. It is thus usually more convenient to express information using symbols that represent more than one bit. Of course, the decimal code that we use in everyday life has this property, since there are ten symbols. Ten is a convenient base for a number system if you are using the fingers of both hands as counting and memory devices. However, if you are representing a number which is stored in computer memory you want the number of symbols to be a multiple of two. In that way the information represented by one symbol can be stored in an integer number of bits, or binary devices; 2, 4, 8, or 16 symbols require 1, 2, 3, or 4 devices. The most common code is hexadecimal, meaning it has sixteen different symbols and thus represents 4 bits of information. The symbols used are 0-9 plus A-F. Thus twelve would be "C" in hexadecimal format and twenty four would be 18.

[1] This may seem strange; it would be like having the address on a letter be four times as large as the contents of the letter itself. One reason the size of the addressable unit is as small as a byte is historical. When the IBM PC appeared, which was several years later than the start of personal computers, 0.1 MB of RAM cost several hundred dollars. Today you can buy 1000 MB of RAM for about the same price.

Remember, in hexadecimal, the "1" in the left position means sixteen (not ten), and when added to the 8 in the next position the entire number equals twenty four.

Number			
base 10	base 2	base 4	base 16
0	00000000	0000	00
1	00000001	0001	01
2	00000010	0002	02
3	00000011	0003	03
4	00000100	0010	04
5	00000101	0011	05
6	00000110	0012	06
7	00000111	0013	07
8	00001000	0020	08
9	00001001	0021	09
10	00001010	0022	0A
11	00001011	0023	0B
12	00001100	0030	0C
13	00001101	0031	0D
14	00001110	0032	0E
15	00001111	0033	0F
16	00010000	0100	10
17	00010001	0101	11
127	01111111	1333	7F
255	11111111	3333	FF

	2s compl
-1	11111111
-2	11111110
-3	11111101

Figure Counting. The left column lists consecutive numbers in base 10, the common way we humans write numbers. In the next three columns are the same numbers are represented in base 2, 4, and 16 formats. The base 16, or hexadecimal format uses

the first six letters of the alphabet to represent values greater than nine. Base 2 and 16 formats are commonly used by computer engineers while the base 4 format is included merely to illustrate that it could be used (and we will see later that living organisms use base 4 hardware). When you count you increase the value of a symbol until you run out of values, then you "carry" a one over to the next digit. In a base 2 system you only have 0 and 1 to count with, so you must "carry" for every other number as you count. There are several ways negative numbers could be represented, but the most common is the two's complement method. In this format the sum of a positive and negative number is obtained by simply adding the binary representations of the two numbers.

Since a series of bits can be interpreted as number, it might seem that codes would not be needed to represent numbers in computer memory. That is wrong. An example of a code to represent negative numbers has been already been presented in Figure Counting. The code is called "two's complement"; it is the most common code for negative numbers because the same algorithm used to add two positive numbers can be used to add a positive and negative or two negative numbers, i.e. no special rule is needed for subtraction.

To add two binary numbers you use the same algorithm learned in the First grade (or Second or where ever) for decimal numbers. Start at the right end of each number and add the digits. If the sum of the digits can be written as one symbol it is written in that position of the result, and you proceed to the next position of each number. However, if the sum of the digits produces a number with two symbols, the left symbol of that number is written as the result for that position, and the 1 is "carried" over to become part of the sum for the next position. In binary code the addition table is very simple: 0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 10.

Use this procedure to add the bytes 00000001 and 11111111. The add for position one produces 0 and carry the 1. All the remaining adds give the same result and the 1 to be carried from the last add is lost because the byte is stored as eight bits; any higher bit "overflows", i.e. it is lost. Thus the result is 00000000. This is exactly the result of adding 1 to –1. Thus, as seen in Figure Counting, the two's complement representation of –1 is 11111111. This certainly isn't the only possible code for –1, it's just the most efficient for most computers.

As the second example of a code, suppose you want to store a number in memory that requires than 8 bits, more than one byte. When you retrieve several consecutive bytes from a memory device, in which order do you assemble the bytes? Unfortunately, there are two systems. In the Big-Endian representation the byte with the lowest memory address (the one received first) is the most significant, i.e. it goes at the left end. In the Little-Endian representation the byte with the lowest address value is the least significant byte, i.e. it goes at the right end. The Big-Endian format is followed in IBM mainframes, computers running UNIX, and Apple Macintosh computers. The Little-Endian format is followed in Intel-Microsoft PCs. The PowerPC chips used in many IBM and Apple computers can be switched to either system. Thus, even concatenating memory locations requires a code.

Up to now we have been talking about integers. A more complex code is needed to represent fractions (including decimal fractions), large, and small numbers. These numbers are typically stored and manipulated in exponential (or floating point) format, containing a mantissa and an exponent term (see Figure Exponent). Consistent with other numerical codes used by computers, both the mantissa and the exponent are in binary format, and the base for the exponent is two.

A common size for a floating point number is 4 bytes or 32 bits, sometimes called a "single precision" representation. As this term suggests, it does not promise to represent all original numbers exactly, but just to a defined precision, e.g. there is no exact representation of the fraction 1/3 in this format. In the most commonly used format, described in Figure FloatFormat, the first bit is used to indicate the sign, the next 8 bits are the exponent, and the remaining 23 bits represent the fractional part of number.

Figure FloatFormat

Single precision float format
(IEEE 754-1990)

Example: convert "7.25" to a binary 32 bit floating point number

1. Write the number in binary point notation (bits to the right of the point represent increasing powers of 1/2 = 1/2, 1/4, 1/8...):

111.01

2. Move the point repeatedly to the left and stop before last 1, the number of moves required is the exponent (to the base 2), bits to the right of the point become the fractional part of the float (the mantissa):

1.1101

3. Convert the exponent, 2 for this example, into "excess 127" notation by adding 127 (or 1111111 in binary):

00000010 + 01111111 = 10000001

4. Put it all together; the first bit is 0 if the number is positive, 1 if negative, the exponent part takes 8 bits and remaining 23 bits are the mantissa:

sign	exponent	mantissa
0	10000001	11010000000000000000000

FloatFormat

Figure FloatFormat. A positive or negative fractional number can be represented in this format. The number is first written in binary point notation where the positions to the left of the point represent increasing powers of 2 while the positions on the right represent increasing powers of 1/2. This binary number is then “normalized” by moving the point to the left until there is only one “1” to the left. The number of moves is the exponent of the final number, and the binary array to the right of the point is the fractional, or mantissa of the final number. The exponent is represented in “excess 127” notation, which is just a convenient way to represent negative numbers, similar to the “twos complement” format.

There are many formats that are use in computer systems to represent different kinds of numbers. Most of the formats are defined after many meetings of large committees of computer programmers and scientists from both corporate and academic environments. The Institute of Electrical and Electronics Engineers organizes many of these standards committees, and thus the format names often start with IEEE.

The computer programmer that is writing code makes the decision of which type of format to use when storing each data item. If the data is the amount deposited in a bank by a customer the programmer will use one type. If the data is the speed of a rocket the programmer will use another type. At least one serious failure of a rocket was due to the assumption by a programmer that the velocity would never exceed a certain value, and thus used a format that could only store numbers up to that value. When the velocity exceed that limit, the bits in the memory location had no relation to the velocity, and the rocket was lost. As we see below, there are many types of data other that numerical values.

Computer codes for text

It is common to want to transmit alphanumeric characters on the Internet or manipulate and store them using a computer. Since I am writing this paragraph using a computer and a word processing program, I am doing exactly that now using the ASCII (American Standard Code for Information Interchange; pronounced “ass key”) code. There are old 6 bit and 7 bit versions of this code, but now 8 bits, or one byte, are reserved for each character. Since one byte can code for 256 alternatives, there is more than enough room for lower and upper case versions of the letters of the English alphabet, the numerical digits, punctuation and formatting characters, and many special symbols. In addition, about a dozen of the first characters (those that come early in the code) are used as control instructions for hardware devices that handle the data stream. The code for non-common characters is not as standardized, and thus can only be used between compatible devices. As an example, Apple Macintosh computers use much of the non-standard portion of the code for mathematical symbols and the special characters used in French, Spanish, German and Greek languages, with variations between fonts. A portion of the standard ASCII code is given in Figure ASCII.

BINARY	HEX	DECIMAL	NAME	SYMBOL
00101101	2D	45	hyphen	-
00101110	2E	46	period	.
00101111	2F	47	slash	/
00110000	30	48	zero	0
00110001	31	49	one	1
00110010	32	50	two	2
00110011	33	51	three	3
00110100	34	52	four	4
00110101	35	53	five	5
00110110	36	54	six	6
00110111	37	55	seven	7
00111000	38	56	eight	8
00111001	39	57	nine	9
00111010	3A	58	colon	:
00111011	3B	59	semicolon	;
00111100	3C	60	less than	<
00111101	3D	61	equal	=
00111110	3E	62	greater than	>
00111111	3F	63	question	?
01000000	40	64	at	@
01000001	41	65	capital A	A
01000010	42	66	capital B	B

Figure ASCII. The “American Standard Code for Information Interchange” or ASCII, is perhaps the most common, but certainly not the only code used for English characters. It also contains “control” characters which can be used to control simple printers and monitors.

There are other digital codes for alphanumeric symbols. In a time long ago, before personal computers were invented, when computers were room-filling machines made by IBM, alphanumeric data was often encoded by the Extended Binary-Coded Decimal Interchange Code (EBCDIC; pronounced “ebb see dick”). This is the code you would need to understand a stack of punched cards that you found in your father’s attic.

Since the entire world is becoming dependent on computers for every mode of communication, any modern symbol code must support languages that use large

numbers of non-Roman characters, e.g. Chinese. The Unicode character code was proposed by a number of companies in 1991 to accomplish this end, and is now the “established” modern character code. This code uses 16 bits (2 bytes) for each character, so that 65,536 characters can be represented. The lower byte of Unicode still represents the ASCII character code, so it is relatively easy to integrate Unicode in older applications. The goal for the developers of Unicode is to assign unique codes to characters and ideographs of the most common languages of this world, so that one coding can be used for all. Over 20,000 ideograph characters have been assigned by standards bodies in China, Japan, Korea, and Taiwan.

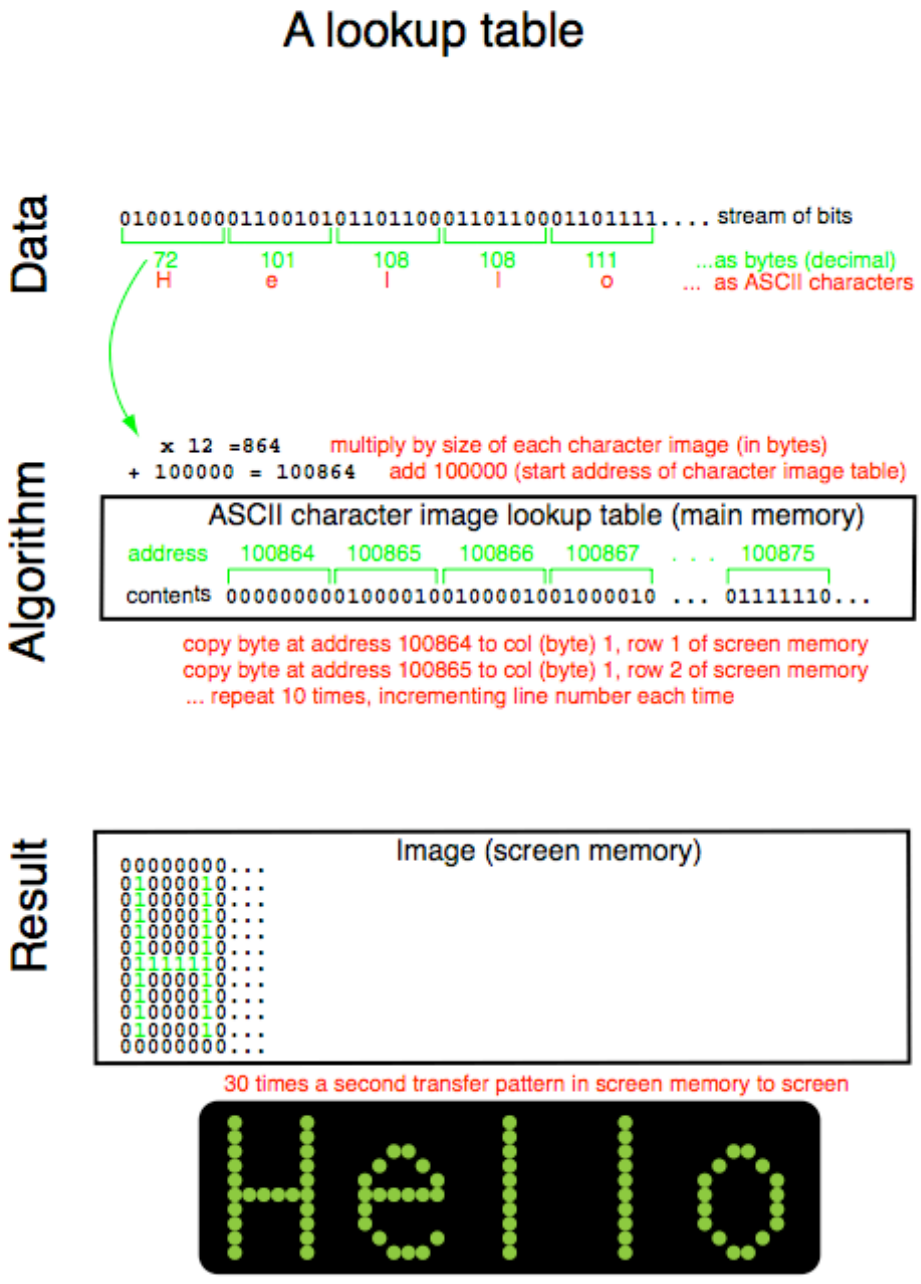
Codes implemented using a lookup table

There are often cases when decoding a string of bits can not be defined as a simple arithmetic process or logical rule. A powerful and general method is to translate the string into a number and use that number as an address to retrieve the decoded information from a table stored in computer memory.

As an example, a lookup table could contain digital images of text in sequential locations in computer memory. To obtain the image of one character, take the code for the character (interpreting it as a positive integer), multiply by the size (number of bytes) needed to store one image, and add the number which is the starting address in memory for the lookup table. Then read the series of bits stored in memory at that location, and you have the image. Of course, you have to have a rule for translating the stored bit stream into an image. The usual method is to sequentially scan a rectangle of n x m pixels (locations) on the screen and make pixels bright if the bit is 1, dark otherwise.

If it is only necessary to display text with one size and font, the table lookup can be implemented with a special chip, a character generator. This solution is fast and cheap and frees up the main CPU to do other tasks. Computer monitors and personal computers up to 1984, used character generators. The Apple Macintosh, introduced that year, had a Graphical Users Interface (GUI) and was always in "graphics mode", i.e. all characters were "drawn" on the screen using a software look-up table stored in main memory. Thus, look-up tables could be easily generated to represent many fonts and styles of text. If you were using a word processing program, you could now see the text close to the way it would appear after being printed. Of course for this to be really useful it was necessary to develop the laser printer, which also "drew" the characters on the paper. Apple later introduced laser printers which were compatible with its computers, creating the “desk top publishing” industry.

Figure LookUpTable



LookUpTable

Figure LookUpTable. In this example the look-up-table contains the graphical representation of a series of characters. A “0” means black while a “1” means white. The data representing individual characters can be easily accessed and used to generate a picture of the character on a screen.

A look-up table can be useful for storing many kinds of data, not just images. You might be building a digital scale which generates the weight in grams, but you want to sell it in the USA, which also uses ounces. One method to convert the number of grams to ounces would be to use a chip that could multiply the number of grams by a constant. However, another would be to use the number of grams as an address to a memory chip containing the weight in ounces (and tenths) in each memory location.

Codes for images

The previous scheme for painting images of the ASCII characters on a screen is a code for producing specific images. However, simple codes for representing general images are constructed in a similar manor. The image is superimposed on a grid. The grid is then scanned left to right, usually starting from the top left hand corner. A white pixel is represented by a 1, and black by 0. The resulting bit stream codes for the image. A real code needs a few more features, e.g. the width and height of the image in pixels should be encoded in the beginning of the image so the display device can present the data.

Figure ImageFormat

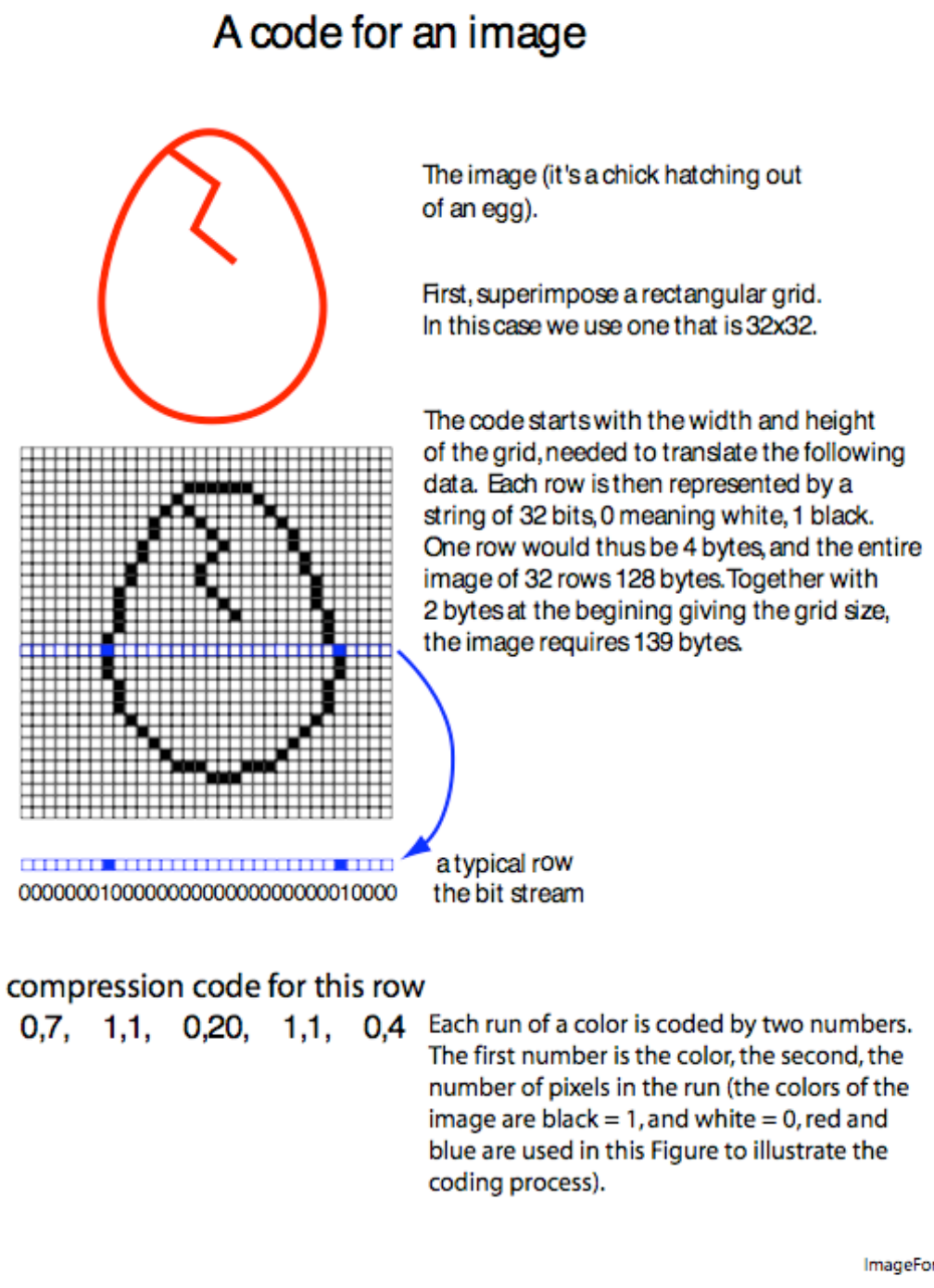


Figure ImageFormat. This format can be considered a version of a look-up table, except it is intended to be read sequentially once to produce an image.

The image of the cracked egg in Figure ImageFormat has long sequences of white, and it is inefficient to represent these sequences with a "1" for each pixel. A typical compression code consists of pairs of numbers, with the first number specifying the color (here black or white) and the second giving the number of sequential pixels having this color. Fax transmissions and the GIF (Graphics Interchange Format) used by Web browsers use a sophisticated version of this code. Images which are not primarily composed of regions with a constant color, e.g. photographs, are not efficiently coded with this scheme. Other codes, e.g. JPEG (Joint Photographic Experts Group), are used for these types of images. A readable summary of graphics formats is "Encyclopedia of Graphics File Formats", by Murray and vanRyper, O'Reillly & Associates, 1996, which at more than 1000 pages, suggests the complexity of image formats.

Biological information

In living systems, information is encoded in molecules, not patterns of electrical charge or magnetic domains. Extremely long molecules, deoxyribonucleic acids (DNA), carry the genetic information as a chain of four kinds of nucleotides. Each nucleotide is a symbol, or token, which carries two bits of information. The information in DNA is translated into proteins, linear polymers of twenty different amino acids. The proteins fold up into specific shapes to form the structural units of cells and to form catalytic machines (enzymes) that promote chemical reactions. The chemical reactions break down food to generate energy, create molecular building units, and form more organism.

Purpose of life

On the Internet there are two distinct classes of information, information that is being transmitted, and information that is used to accomplish the transmission: the cargo and the machine carrying the cargo. It is possible to make a sharp differentiation between these two kinds of information precisely because there is a defined goal: to deliver a defined message.

The purpose of living organisms is to produce progeny (the organisms must survive long enough to produce progeny, but survival itself is not sufficient). This is a tautology, for the progeny of organisms are the only organisms we see on earth. To produce progeny organisms must store and process information. However, most of the information needed to produce (specify) the progeny also is used to maintain the parental organism. Thus, the machine and the cargo are hopelessly intertwined.

Perhaps a more important distinction between information used to run the internet and information carried by living organisms is the fact that there is no manual that tells us what information an organism contains or what that information does. Understanding the Internet may be a prodigious effort, but at least there are manuals.

The study of life is an experimental, inductive process, and you never know when you are finished.

The vocabulary of biochemistry

It is difficult to discuss the details of information transfer and use in a biological system without some familiarity with biochemistry. However, under the details of the chemistry, information theory provides a framework for understanding the function and limitations of the system.

Summary of the chapter

Information is defined and measured as the ability to make choices. The smallest unit of information, the bit, allows a choice between two alternatives.

Humans use many formats to represent numbers as text in written documents. The Romans used addition to represent large numbers while the Arabs used the more powerful process of multiplication. Exponents, an extension of multiplication, are used in the scientific format for numbers to represent large and small values.

Codes and formats are required to convert a sequence of bits into usable data. Numbers, text, images or audio all require codes in order to be represented by the bits stored in computer memory.

Information is stored and transmitted by living organism as a series of nucleotides linked to form large molecules of DNA. The information is translated in the cell into proteins with specific shapes and functions.