MITOPENCOURSEWARE

Information and Entropy

by Professor Paul Penfield

Preface

These notes, and the related freshman-level course, are about information. Although you may have a general idea of what information is, you may not realize that the information you deal with can be quantified. That's right, you can often measure the amount of information and use general principles about how information behaves. We will consider applications to computation and to communications, and we will also look at general laws in other fields of science and engineering.

One of these general laws is the Second Law of Thermodynamics. Although thermodynamics, a branch of physics, deals with physical systems, the Second Law is approached here as an example of information processing in natural and engineered systems. The Second Law as traditionally stated in thermodynamics deals with a physical quantity known as "entropy." Everybody has heard of entropy, but few really understand it. Only recently has entropy been widely accepted as a form of information.

The Second Law is surely one of science's most glorious achievements, but as usually taught, through physical systems and models such as ideal gases, it is difficult to appreciate at an elementary level. On the other hand, the forms of the Second Law that apply to computation and communications (where information is processed) are more easily understood, especially today as the information revolution is getting under way.

These notes and the course based on them are intended for first-year university students. Although they were developed at the Massachusetts Institute of Technology, a university that specializes in science and engineering, and one at which a full year of calculus is required, calculus is not used much at all in these notes. Most examples are from discrete systems, not continuous systems, so that sums and differences can be used instead of integrals and derivatives. In other words, we can use algebra instead of calculus. This course should be accessible to university students who are not studying engineering or science, and even to well prepared high-school students. In fact, when combined with a similar one about energy, this course could provide excellent science background for liberal-arts students.

The analogy between information and energy is interesting. In both high-school and first-year college physics courses, students learn that there is a physical quantity known as energy which is conserved (it cannot be created or destroyed), but which can appear in various forms (potential, kinetic, electric, chemical, etc.). Energy can exist in one region of space or another, can flow from one place to another, can be stored for later use, and can be converted from one form to another. In many ways the industrial revolution was all about harnessing energy for useful purposes. But most important, energy is conserved—despite its being moved, stored, and converted, at the end of the day there is still exactly the same total amount of energy.

This conservation of energy principle is sometimes known as the First Law of Thermodynamics. It has proven to be so important and fundamental that whenever a "leak" was found, the theory was rescued by defining a new form of energy. One example of this occurred in 1905 when Albert Einstein recognized that mass is a form of energy, as expressed by his famous formula $E = mc^2$. That understanding later enabled the development of devices (atomic bombs and nuclear power plants) that convert energy from its form as mass to other forms.

But what about information? If entropy is really a form of information, there should be a theory that

covers both and describes how information can be turned into entropy or vice versa. Such a theory is not yet well developed, for several historical reasons. Yet it is exactly what is needed to simplify the teaching and understanding of fundamental concepts.

These notes present such a unified view of information, in which entropy is one kind of information, but in which there are other kinds as well. Like energy, information can reside in one place or another, it can be transmitted through space, and it can be stored for later use. But unlike energy, information is not conserved: the Second Law states that entropy never decreases as time goes on—it generally increases, but may in special cases remain constant. Also, information is inherently subjective, because it deals with what you know and what you don't know (entropy, as one form of information, is also subjective—this point makes some physicists uneasy). These two facts make the theory of information different from theories that deal with conserved quantities such as energy—different and also interesting.

The unified framework presented here has never before been developed specifically for freshmen. It is not entirely consistent with conventional thinking in various disciplines, which were developed separately. In fact, we may not yet have it right. One consequence of teaching at an elementary level is that unanswered questions close at hand are disturbing, and their resolution demands new research into the fundamentals. Trying to explain things rigorously but simply often requires new organizing principles and new approaches. In the present case, the new approach is to start with information and work from there to entropy, and the new organizing principle is the unified theory of information.

This will be an exciting journey. Welcome aboard!

Chapter 1

Bits

Information is measured in bits, just as length is measured in meters and time is measured in seconds. Of course knowing the amount of information, in bits, is not the same as knowing the information itself, what it means, or what it implies. In these notes we will not consider the content or meaning of information, just the quantity.

Different scales of length are needed in different circumstances. Sometimes we want to measure length in kilometers, sometimes in inches, and sometimes in Ångströms. Similarly, other scales for information besides bits are sometimes used; in the context of physical systems information is often measured in Joules per Kelvin.

How is information quantified? Consider a situation or experiment that could have any of several possible outcomes. Examples might be flipping a coin (2 outcomes, heads or tails) or selecting a card from a deck of playing cards (52 possible outcomes). How compactly could one person (by convention often named Alice) tell another person (Bob) the outcome of such an experiment or observation?

First consider the case of the two outcomes of flipping a coin, and let us suppose they are equally likely. If Alice wants to tell Bob the result of the coin toss, she could use several possible techniques, but they are all equivalent, in terms of the amount of information conveyed, to saying either "heads" or "tails" or to saying 0 or 1. We say that the information so conveyed is one bit.

If Alice flipped two coins, she could say which of the four possible outcomes actually happened, by saying 0 or 1 twice. Similarly, the result of an experiment with eight equally likely outcomes could be conveyed with three bits, and more generally 2^n outcomes with n bits. Thus the amount of information is the logarithm (to the base 2) of the number of equally likely outcomes.

Note that conveying information requires two phases. First is the "setup" phase, in which Alice and Bob agree on what they will communicate about, and exactly what each sequence of bits means. This common understanding is called the code. For example, to convey the suit of a card chosen from a deck, their code might be that 00 means clubs, 01 diamonds, 10 hearts, and 11 spades. Agreeing on the code may be done before any observations have been made, so there is not yet any information to be sent. The setup phase can include informing the recipient that there is new information. Then, there is the "outcome" phase, where actual sequences of 0 and 1 representing the outcomes are sent. These sequences are the data. Using the agreed-upon code, Alice draws the card, and tells Bob the suit by sending two bits of data. She could do so repeatedly for multiple experiments, using the same code.

After Bob knows that a card is drawn but before receiving Alice's message, he is uncertain about the suit. His uncertainty, or lack of information, can be expressed in bits. Upon hearing the result, his uncertainty is reduced by the information he receives. Bob's uncertainty rises during the setup phase and then is reduced during the outcome phase.

Note some important things about information, some of which are illustrated in this example:

- Information can be learned through observation, experiment, or measurement
- Information is subjective, or "observer-dependent." What Alice knows is different from what Bob knows (if information were not subjective, there would be no need to communicate it)
- A person's uncertainty can be increased upon learning that there is an observation about which information may be available, and then can be reduced by receiving that information
- Information can be lost, either through loss of the data itself, or through loss of the code
- The physical form of information is localized in space and time. As a consequence,
  - Information can be sent from one place to another
  - Information can be stored and then retrieved later

1.1 The Boolean Bit

As we have seen, information can be communicated by sequences of 0 and 1 values. By using only 0 and 1, we can deal with data from many different types of sources, and not be concerned with what the data means. We are thereby using abstract, not specific, values. This approach lets us ignore many messy details associated with specific information processing and transmission systems.

Bits are simple, having only two possible values. The mathematics used to denote and manipulate single bits is not difficult. It is known as Boolean algebra, after the mathematician George Boole (1815–1864). In some ways Boolean algebra is similar to the algebra of integers or real numbers which is taught in high school, but in other ways it is different.

Algebra is a branch of mathematics that deals with variables that have certain possible values, and with functions which, when presented with one or more variables, return a result which again has certain possible values. In the case of Boolean algebra, the possible values are 0 and 1.

First consider Boolean functions of a single variable that return a single value. There are exactly four of them. One, called the identity, simply returns its argument. Another, called not (or negation, inversion, or complement) changes 0 into 1 and vice versa. The other two simply return either 0 or 1 regardless of the argument. Here is a table showing these four functions:

x	$f(x)$			
Argument	$IDENTITY$	NOT	$ZERO$	ONE
0	0	1	0	1
1	1	0	0	1

Table 1.1: Boolean functions of a single variable

Note that Boolean algebra is simpler than algebra dealing with integers or real numbers, each of which has infinitely many functions of a single variable.

Next, consider Boolean functions with two input variables A and B and one output value C. How many are there? Each of the two arguments can take on either of two values, so there are four possible input patterns (00, 01, 10, and 11). Think of each Boolean function of two variables as a string of Boolean values 0 and 1 of length equal to the number of possible input patterns, i.e., 4. There are exactly 16 (2^4) different ways of composing such strings, and hence exactly 16 different Boolean functions of two variables. Of these 16, two simply ignore the input, four assign the output to be either A or B or their complement, and the other ten depend on both arguments. The most often used are AND, OR, XOR (exclusive or), $NAND$ (not

x	$f(x)$				
Argument	*AND*	*NAND*	*OR*	*NOR*	*XOR*
00	0	1	0	1	0
01	0	1	1	0	1
10	0	1	1	0	1
11	1	0	1	0	0

Table 1.2: Five of the 16 possible Boolean functions of two variables

and), and *NOR* (not or), shown in Table 1.2. (In a similar way, because there are 8 possible input patterns for three-input Boolean functions, there are 2^8 or 256 different Boolean functions of three variables.)

It is tempting to think of the Boolean values 0 and 1 as the integers 0 and 1. Then *AND* would correspond to multiplication and *OR* to addition, sort of. However, familiar results from ordinary algebra simply do not hold for Boolean algebra, so such analogies are dangerous. It is important to distinguish the integers 0 and 1 from the Boolean values 0 and 1; they are not the same.

There is a standard notation used in Boolean algebra. (This notation is sometimes confusing, but other notations that are less confusing are awkward in practice.) The *AND* function is represented the same way as multiplication, by writing two Boolean values next to each other or with a dot in between: A *AND* B is written AB or $A \cdot B$. The *OR* function is written using the plus sign: $A + B$ means A *OR* B. Negation, or the *NOT* function, is denoted by a bar over the symbol or expression, so *NOT* A is $\overline{A}$. Finally, the exclusive-or function *XOR* is represented by a circle with a plus sign inside, $A \oplus B$.

NOT	$\overline{A}$
AND	$A \cdot B$
NAND	$\overline{A \cdot B}$
OR	$A + B$
NOR	$\overline{A + B}$
XOR	$A \oplus B$

Table 1.3: Boolean logic symbols

There are other possible notations for Boolean algebra. The one used here is the most common. Sometimes *AND*, *OR*, and *NOT* are represented in the form $AND(A, B)$, $OR(A, B)$, and $NOT(A)$. Sometimes infix notation is used where $A \wedge B$ denotes $A \cdot B$, $A \vee B$ denotes $A + B$, and $\sim A$ denotes $\overline{A}$. Boolean algebra is also useful in mathematical logic, where the notation $A \wedge B$ for $A \cdot B$, $A \vee B$ for $A + B$, and $\neg A$ for $\overline{A}$ is commonly used.

Several general properties of Boolean functions are useful. These can be proven by simply demonstrating that they hold for all possible input values. For example, a function is said to be **reversible** if, knowing the output, the input can be determined. Two of the four functions of a single variable are reversible in this sense (and in fact are self-inverse). Clearly none of the functions of two (or more) inputs can by themselves be reversible, since there are more input variables than output variables. However, some combinations of two or more such functions can be reversible if the resulting combination has the name number of outputs as inputs; for example it is easily demonstrated that the exclusive-or function $A \oplus B$ is reversible when augmented by the function that returns the first argument—that is to say, more precisely, the function of two variables that has two outputs, one $A \oplus B$ and the other A, is reversible.

For functions of two variables, there are many properties to consider. For example, a function of two variables A and B is said to be **commutative** if its value is unchanged when A and B are interchanged, i.e., if $f(A, B) = f(B, A)$. Thus the function *AND* is commutative because $A \cdot B = B \cdot A$. Some of the other 15 functions are also commutative. Some other properties of Boolean functions are illustrated in the identities in Table 1.4.

Idempotent:	$A \cdot A = A$ $A + A = A$	Absorption:	$A \cdot (A + B) = A$ $A + (A \cdot B) = A$
Complementary:	$A \cdot \overline{A} = 0$ $A + \overline{A} = 1$ $A \oplus A = 0$ $A \oplus \overline{A} = 1$	Associative:	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$ $A + (B + C) = (A + B) + C$ $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
Minimum:	$A \cdot 1 = A$ $A \cdot 0 = 0$	Unnamed Theorem:	$A \cdot (\overline{A} + B) = A \cdot B$ $A + (\overline{A} \cdot B) = A + B$
Maximum:	$A + 0 = A$ $A + 1 = 1$	De Morgan:	$\overline{A} \cdot \overline{B} = \overline{A + B}$ $\overline{A} + \overline{B} = \overline{A \cdot B}$
Commutative:	$A \cdot B = B \cdot A$ $A + B = B + A$ $A \oplus B = B \oplus A$ $\overline{A \cdot B} = \overline{B \cdot A}$ $\overline{A + B} = \overline{B + A}$	Distributive:	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ $A + (B \cdot C) = (A + B) \cdot (A + C)$

Table 1.4: Properties of Boolean Algebra
These formulas are valid for all values of A, B, and C.

The Boolean bit has the property that it can be copied (and also that it can be discarded). In Boolean algebra copying is done by assigning a name to the bit and then using that name more than once. Because of this property the Boolean bit is not a good model for quantum-mechanical systems. A different model, the quantum bit, is described below.

1.2 The Circuit Bit

Combinational logic circuits are a way to represent Boolean expressions graphically. Each Boolean function (NOT, AND, XOR, etc.) corresponds to a "combinational gate" with one or two inputs and one output, as shown in Figure 1.1. The different types of gates have different shapes. Lines are used to connect the output of one gate to one or more gate inputs, as illustrated in the circuits of Figure 1.2.

Logic circuits are widely used to model digital electronic circuits, where the gates represent parts of an integrated circuit and the lines represent the signal wiring.

The circuit bit can be copied (by connecting the output of a gate to two or more gate inputs) and

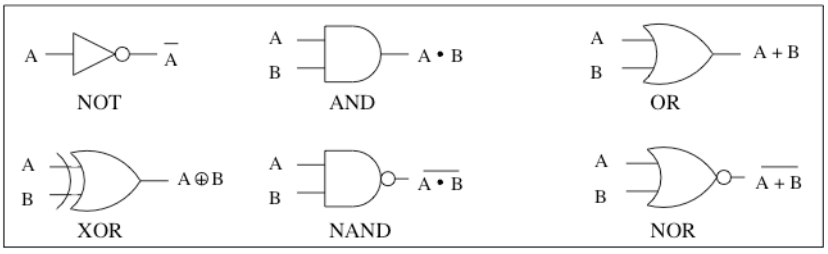


Figure 1.1: Logic gates corresponding to the Boolean functions NOT, AND, OR, XOR, $NAND$, and NOR

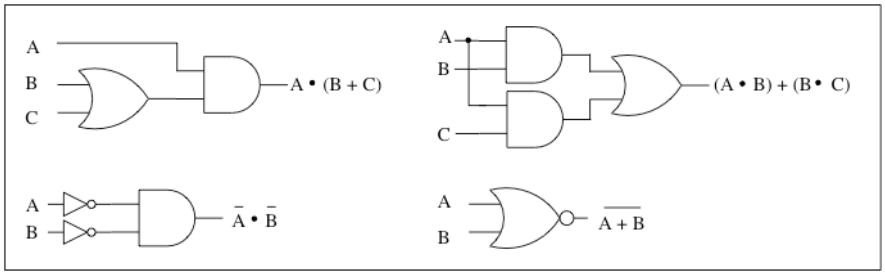


Figure 1.2: Some combinational logic circuits and the corresponding Boolean expressions

discarded (by leaving an output unconnected).

Combinational circuits have the property that the output from a gate is never fed back to the input of any gate whose output eventually feeds the input of the first gate. In other words, there are no loops in the circuit. Circuits with loops are known as **sequential logic**, and Boolean algebra is not sufficient to describe them. For example, consider the simplest circuit in Figure 1.3. The inverter (*NOT* gate) has its output connected to its input. Analysis by Boolean algebra leads to a contradiction. If the input is 1 then the output is 0 and therefore the input is 0. There is no possible state according to the rules of Boolean algebra. On the other hand, consider the circuit in Figure 1.3 with two inverters. This circuit has two possible states. The bottom circuit has two stable states if it has an even number of gates, and no stable state if it has an odd number.

A model more complicated than Boolean algebra is needed to describe the behavior of sequential logic circuits. For example, the gates or the connecting lines (or both) could be modeled with time delays. The circuit at the bottom of Figure 1.3 (for example with 13 or 15 gates) is commonly known as a ring oscillator, and is used in semiconductor process development to test the speed of circuits made using a new process.

1.3 The Control Bit

In computer programs, Boolean expressions are often used to determine the flow of control, i.e., which statements are executed. Suppose, for example, that if one variable x is negative and another y is positive, then a third variable z should be set to zero. In the language Scheme, the following statement would accomplish this: `(if (and (< x 0) (> y 0)) (define z 0))` (other languages have their own ways of expressing the same thing).

The algebra of control bits is like Boolean algebra with an interesting difference: any part of the control expression that does not affect the result may be ignored. In the case above (assuming the arguments of `and` are evaluated left to right), if x is found to be positive then the result of the `and` operation is 0 regardless of the value of y, so there is no need to see if y is positive or even to evaluate y. As a result the program can run faster, and side effects associated with evaluating y do not happen.

1.4 The Physical Bit

If a bit is to be stored or transported, it must have a physical form. Whatever object stores the bit has two distinct states, one of which is interpreted as 0 and the other as 1. A bit is stored by putting the object in one of these states, and when the bit is needed the state of the object is measured. If the object has moved from one location to another without changing its state then communications has occurred. If the object has persisted over some time in its same state then it has served as a memory. If the object has had its state changed in a random way then its original value has been forgotten.

In keeping with the Engineering Imperative (make it smaller, faster, stronger, smarter, safer, cheaper),

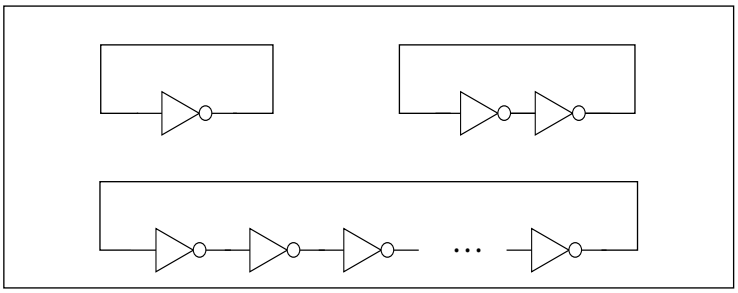


Figure 1.3: Some sequential logic circuits

we are especially interested in physical objects that are small. The limit to how small an object can be and still store a bit of information comes from quantum mechanics. The quantum bit, or qubit, is a model of an object that can store a single bit but is so small that it is subject to the limitations quantum mechanics places on measurements.

1.5 The Quantum Bit

According to quantum mechanics, it is possible for a small object to have two states which can be measured. This sounds perfect for storing bits, and the result is often called a **qubit**, pronounced "cue-bit." The two values are often denoted $|0\rangle$ and $|1\rangle$ rather than 0 and 1, because this notation generalizes to what is needed to represent more than one qubit, and it avoids confusion with the real numbers 0 and 1. There are three features of quantum mechanics, **reversibility**, **superposition**, and **entanglement**, that make qubits or collections of qubits different from Boolean bits.

Reversibility: It is a property of quantum systems that if one state can lead to another by means of some transition, then the reverse transition is also possible. Thus all functions in the mathematics of qubits are reversible, and the output of a function cannot be discarded, since that would be irreversible. However, there are at least two important sources of irreversibility in quantum systems. First, if a quantum system interacts with its environment and the state of the environment is unknown, then some of the information in the system is lost. Second, the very act of measuring the state of a system is irreversible.

Superposition: Suppose a quantum mechanical object is prepared so that it has a combination, or superposition, of its two states, i.e., a state somewhere between the two states. What is it that would be measured in that case?

In a classical, non-quantum context, a measurement could determine just what that combination is. Furthermore, for greater precision a measurement could be repeated, and multiple results averaged. However, the quantum context is different. In a quantum measurement, the question that is asked is whether the object is or is not in some particular state, and the answer is always either "yes" or "no," never "maybe" and never, for example, "27% yes, 73% no." Furthermore, after the measurement the system ends up in the state corresponding to the answer, so further measurements will not yield additional information. The result of any particular measurement cannot be predicted, but the likelihood of the answer, expressed in terms of probabilities, can. This peculiar nature of quantum mechanics offers both a limitation of how much information can be carried by a single qubit, and an opportunity to design systems which take special advantage of these features.

We will illustrate quantum bits with an example. Let's take as our qubit a photon, which is the elementary particle for electromagnetic radiation, including radio, TV, and light. A photon is a good candidate for carrying information from one place to another. It is small, and travels fast.

A photon has electric and magnetic fields oscillating simultaneously. The direction of the electric field is called the direction of polarization (we will not consider circularly polarized photons here). Thus if a photon

is headed in the z-direction, its electric field can be in the x-direction, in the y-direction, or in fact in any direction in the x-y plane, sometimes called the "horizontal-vertical plane."

The polarization can be used to store a bit of information. Thus Alice could prepare a photon with horizontal polarization if the bit is $|0\rangle$ and vertical polarization if the bit is $|1\rangle$. Then when Bob gets the photon, he can measure its vertical polarization (i.e., ask whether the polarization is vertical). If the answer is "yes," then he infers the bit is $|1\rangle$.

It might be thought that more than a single bit of information could be transmitted by a single photon's polarization. Why couldn't Alice send two bits, using angles of polarization different from horizontal and vertical? Why not use horizontal, vertical, half-way between them tilted right, and half-way between them tilted left? The problem is that Bob has to decide what angle to measure. He cannot, because of quantum-mechanical limitations, ask the question "what is the angle of polarization" but only "is the polarization in the direction I choose to measure." And the result of his measurement can only be "yes" or "no," in other words, a single bit. And then after the measurement the photon ends up either in the plane he measured (if the result was "yes") or perpendicular to it (if the result was "no").

If Bob wants to measure the angle of polarization more accurately, why couldn't he repeat his measurement many times and take an average? This does not work because the very act of doing the first measurement resets the angle of polarization either to the angle he measured or to the angle perpendicular to it. Thus subsequent measurements will all be the same.

Or Bob might decide to make multiple copies of the photon, and then measure each of them. This approach does not work either. The only way he can make a copy of the photon is by measuring its properties and then creating a new photon with exactly those properties. All the photons he creates will be the same.

What does Bob measure if Alice had prepared the photon with an arbitrary angle? Or if the photon had its angle of polarization changed because of random interactions along the way? Or if the photon had been measured by an evil eavesdropper (typically named Eve) at some other angle and therefore been reset to that angle? In these cases, Bob always gets an answer "yes" or "no," for whatever direction of polarization he chooses to measure, and the closer the actual polarization is to that direction the more likely the answer is yes. To be specific, the probability of the answer yes is the square of the cosine of the angle between Bob's angle of measurement and Alice's angle of preparation. It is not possible to predict the result of any one of Bob's measurements. This inherent randomness is an unavoidable aspect of quantum mechanics.

Entanglement: Two or more qubits can be prepared together in particular ways. One property, which we will not discuss further now, is known as "entanglement." Two photons, for example, might have identical polarizations (either both horizontal or both vertical). Then they might travel to different places but retain their entangled polarizations. They then would be separate in their physical locations but not separate in their polarizations. If you think of them as two separate photons you might wonder why measurement of the polarization of one would affect a subsequent measurement of the polarization of the other, located far away.

Note that quantum systems don't *always* exhibit the peculiarities associated with superposition and entanglement. For example, photons can be prepared independently (so there is no entanglement) and the angles of polarization can be constrained to be horizontal and vertical (no superposition). In this case qubits behave like Boolean bits.

1.5.1 An Advantage of Qubits

There are things that can be done in a quantum context but not classically. Some are advantageous. Here is one example:

Consider again Alice trying to send information to Bob using polarized photons. She prepares photons that have either horizontal or vertical polarization, and tells that to Bob, during the setup phase. Now let us suppose that a saboteur Sam wants to spoil this communication by processing the photons at some point in the path between Alice and Bob. He uses a machine that simply measures the polarization at an angle he selects. If he selects 45°, every photon ends up with a polarization at that angle or perpendicular to it,

regardless of its original polarization. Then Bob, making a vertical measurement, will measure 0 half the time and 1 half the time, regardless of what Alice sent.

Alice learns about Sam's scheme and wants to reestablish reliable communication with Bob. What can she do?

She tells Bob (using a path that Sam does not overhear) she will send photons at 45° and 135°, so he should measure at one of those angles. Sam's machine then does not alter the photons. Of course if Sam discovers what Alice is doing, he can rotate his machine back to vertical. Or there are other measures and counter-measures that could be put into action.

This scenario relies on the quantum nature of the photons, and the fact that single photons cannot be measured by Sam except along particular angles of polarization. Thus Alice's technique for thwarting Sam is not possible with classical bits.

1.6 The Classical Bit

Because quantum measurement generally alters the object being measured, a quantum bit cannot be measured a second time. On the other hand, if a bit is represented by many objects with the same properties, then after a measurement enough objects can be left unchanged so that the same bit can be measured again.

In today's electronic systems, a bit of information is carried by many objects, all prepared in the same way (or at least that is a convenient way to look at it). Thus in a semiconductor memory a single bit is represented by the presence or absence of perhaps 60,000 electrons (stored on a 10 fF capacitor charged to $1V$). Similarly, a large number of photons are used in radio communication.

Because many objects are involved, measurements on them are not restricted to a simple yes or no, but instead can range over a continuum of values. Thus the voltage on a semiconductor logic element might be anywhere in a range from, say, $0V$ to $1V$. The voltage might be interpreted to allow a margin of error, so that voltages between $0V$ and $0.2V$ would represent logical 0, and voltages between $0.8V$ and $1V$ a logical 1. The circuitry would not guarantee to interpret voltages between $0.2V$ and $0.8V$ properly. If the noise in a circuit is always smaller than $0.2V$, and the output of every circuit gate is either $0V$ or $1V$, then the voltages can always be interpreted as bits without error.

Circuits of this sort display what is known as "restoring logic" since small deviations in voltage from the ideal values of $0V$ and $1V$ are eliminated as the information is processed. The robustness of modern computers depends on the use of restoring logic.

A classical bit is an abstraction in which the bit can be measured without perturbing it. As a result copies of a classical bit can be made. This model works well for circuits using restoring logic.

Because all physical systems ultimately obey quantum mechanics, the classical bit is always an approximation to reality. However, even with the most modern, smallest devices available, it is an excellent one. An interesting question is whether the classical bit approximation will continue to be useful as advances in semiconductor technology allow the size of components to be reduced. Ultimately, as we try to represent or control bits with a small number of atoms or photons, the limiting role of quantum mechanics will become important. It is difficult to predict exactly when this will happen, but some people believe it will be before the year 2015.

1.7 Summary

There are several models of a bit, useful in different contexts. These models are not all the same. In the rest of these notes, the Boolean bit will be used most often, but sometimes the quantum bit will be needed.

Chapter 2

Codes

In the previous chapter we examined the fundamental unit of information, the bit, and its various abstract representations: the Boolean bit (with its associated Boolean algebra and realization in combinational logic circuits), the control bit, the quantum bit, and the classical bit.

A single bit is useful if exactly two answers to a question are possible. Examples include the result of a coin toss (heads or tails), the gender of a person (male or female), the verdict of a jury (guilty or not guilty), and the truth of an assertion (true or false). Most situations in life are more complicated. This chapter concerns ways in which complex objects can be represented not by a single bit, but by arrays of bits.

It is convenient to focus on a very simple model of a system, shown in Figure 2.1, in which the input is one of a predetermined set of objects, or "symbols," the identity of the particular symbol chosen is encoded in an array of bits, these bits are transmitted through space or time, and then are decoded at a later time or in a different place to determine which symbol was originally chosen. In later chapters we will augment this model to deal with issues of robustness and efficiency.

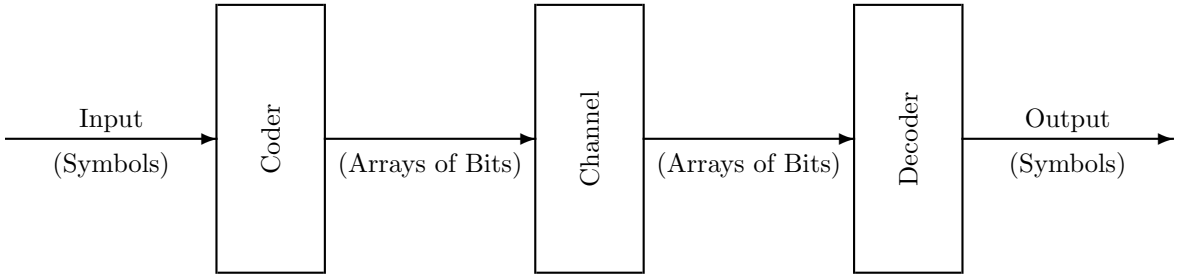


Figure 2.1: Simple model of a communication system

In this chapter we will look into several aspects of the design of codes, and show some examples in which these aspects were either done well or not so well. Individual sections will describe codes that illustrate the important points. Some objects for which codes may be needed include:

- Letters: BCD, EBCDIC, ASCII, Unicode, Morse Code
- Integers: Binary, Gray, 2's complement

- Numbers: Floating-Point
- Proteins: Genetic Code
- Telephones: NANP, International codes
- Hosts: Ethernet, IP Addresses, Domain names
- Images: TIFF, GIF, and JPEG
- Audio: MP3
- Video: MPEG

2.1 Symbol Space Size

The first question to address is the number of symbols that need to be encoded. This is called the **symbol space size**. We will consider symbol spaces of different sizes:

- 1
- 2
- Integral power of 2
- Finite
- Infinite, Countable
- Infinite, Uncountable

If the number of symbols is 2, then the selection can be encoded in a single bit. If the number of possible symbols is 4, 8, 16, 32, 64, or another integral power of 2, then the selection may be coded in the number of bits equal to the logarithm, base 2, of the symbol space size. Thus 2 bits can designate the suit (clubs, diamonds, hearts, or spades) of a playing card, and 5 bits can encode the selection of one student in a class of 32. As a special case, if there is only one symbol, no bits are required to specify it. A dreidel is a four-sided toy marked with Hebrew letters, and spun like a top in a children's game, especially at Hanukkah. The result of each spin could be encoded in 2 bits.

If the number of symbols is finite but not an integral power of 2, then the number of bits that would work for the next higher integral power of 2 can be used to encode the selection, but there will be some unused bit patterns. Examples include the 10 digits, the six faces of a cubic die, the 13 denominations of a playing card, and the 26 letters of the English alphabet. In each case, there is spare capacity (6 unused patterns in the 4-bit representation of digits, 2 unused patterns in the 3-bit representation of a die, etc.) What to do with this spare capacity is an important design issue that will be discussed in the next section.

If the number of symbols is infinite but countable (able to be put into a one-to-one relation with the integers) then a bit string of a given length can only denote a finite number of items from this infinite set. Thus, a 4-bit code for non-negative integers might designate integers from 0 through 15, but would not be able to handle integers outside this range. If, as a result of some computation, it were necessary to represent larger numbers, then this "overflow" condition would have to be handled in some way.

If the number of symbols is infinite and uncountable (such as the value of a physical quantity like voltage or acoustic pressure) then some technique of "discretization" must be used to replace possible values by a finite number of selected values that are approximately the same. For example, if the numbers between 0 and 1 were the symbols and if 2 bits were available for the coded representation, one approach might be to approximate all numbers between 0 and 0.25 by the number 0.125, all numbers between 0.25 and 0.5 by 0.375, and so on. Whether such an approximation is adequate depends on how the decoded data is used.

The approximation is not reversible, in that there is no decoder which will recover the original symbol given just the code for the approximate value. However, if the number of bits available is large enough, then for many purposes a decoder could provide a number that is close enough. Floating-point representation of real numbers in computers is based on this philosophy.

2.2 Use of Spare Capacity

In many situations there are some unused code patterns, because the number of symbols is not an integral power of 2. There are many strategies to deal with this. Here are some:

- Ignore
- Map to other values
- Reserve for future expansion
- Use for control codes
- Use for common abbreviations

These approaches will be illustrated with examples of common codes.

2.2.1 Binary Coded Decimal (BCD)

A common way to represent the digits 0 - 9 is by the ten four-bit patterns shown in Table 2.1. There are six bit patterns (for example 1010) that are not used, and the question is what to do with them. Here are a few ideas that come to mind.

First, the unused bit patterns might simply be ignored. If a decoder encounters one, perhaps as a result of an error in transmission or an error in encoding, it might return nothing, or might signal an output error. Second, the unused patterns might be mapped into legal values. For example, the unused patterns might all be converted to 9, under the theory that they represent 10, 11, 12, 13, 14, or 15, and the closest digit is 9. Or they might be decoded as 2, 3, 4, 5, 6, or 7, by setting the initial bit to 0, under the theory that the first bit might have gotten corrupted. Neither of these theories is particularly appealing, but in the design of a system using BCD, some such action must be provided.

Digit	Code
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Table 2.1: Binary Coded Decimal

2.2.2 Genetic Code

Another example of mapping unused patterns into legal values is provided by the Genetic Code, described in Section 2.7. A protein consists of a long sequence of amino acids, of 20 different types, each with between 10 and 27 atoms. Living organisms have millions of different proteins, and it is believed that all cell activity involves proteins. Proteins have to be made as part of the life process, yet it would be difficult to imagine millions of special-purpose chemical manufacturing units, one for each type of protein. Instead, a general-purpose mechanism assembles the proteins, guided by a description (think of it as a blueprint) that is contained in DNA (deoxyribonucleic acid) and RNA (ribonucleic acid) molecules. Both DNA and RNA are linear chains of small "nucleotides;" a DNA molecule might consist of more than a hundred million such nucleotides. In DNA there are four types of nucleotides, each consisting of some common structure and one of four different bases, named Adenine, Cytosine, Guanine, and Thymine. In RNA the structure is similar except that Thymine is replaced by Uracil.

The Genetic Code is a description of how a sequence of nucleotides specifies an amino acid. Given that relationship, an entire protein can be specified by a linear sequence of nucleotides. Note that the coded description of a protein is not by itself any smaller or simpler than the protein itself; in fact, the number of atoms needed to specify a protein is larger than the number of atoms in the protein itself. The value of the standardized representation is that it allows the same assembly apparatus to fabricate different proteins at different times.

Since there are four different nucleotides, one of them can specify at most four different amino acids. A sequence of two can specify 16 different amino acids. But this is not enough – there are 20 different amino acids used in proteins – so a sequence of three is needed. Such a sequence is called a codon. There are 64 different codons, more than enough to specify 20 amino acids. The spare capacity is used to provide more than one combination for most amino acids, thereby providing a degree of robustness. For example, the amino acid Alanine has 4 codes including all that start with GC; thus the third nucleotide can be ignored, so a mutation which changed it would not impair any biological functions. In fact, eight of the 20 amino acids have this same property that the third nucleotide is a "don't care." (It happens that the third nucleotide is more likely to be corrupted during transcription than the other two, due to an effect that has been called "wobble.")

An examination of the Genetic Code reveals that three codons (UAA, UAG, and UGA) do not specify any amino acid. These three signify the end of the protein. Such a "stop code" is necessary because different proteins are of different length. The codon AUG specifies the amino acid Methionine and also signifies the beginning of a protein; all protein chains begin with Methionine. Many man-made codes have this property, that some bit sequences designate data but a few are reserved for control information.

2.2.3 Telephone Area Codes

The third way in which spare capacity can be used is by reserving it for future expansion. When AT&T started using telephone Area Codes for the United States and Canada in 1947 (they were made available for public use in 1951), the codes contained three digits, with three restrictions.

- The first digit could not be 0 or 1, to avoid conflicts with 0 connecting to the operator, and 1 being an unintended effect of a faulty sticky rotary dial or a temporary circuit break of unknown cause (or today a signal that the person dialing acknowledges that the call may be a toll call)
- The middle digit could only be a 0 or 1 (0 for states and provinces with only one Area Code, and 1 for states and provinces with more than one). This restriction allowed an Area Code to be distinguished from an exchange (an exchange, the equipment that switched up to 10,000 telephone numbers, was denoted at that time by the first two letters of a word and one number; today exchanges are denoted by three digits).
- The last two digits could not be the same (numbers of the form *abb* are more easily remembered and therefore more valuable)—thus $x11$ dialing sequences such as 911 (emergency), 411 (directory

assistance), and 611 (repair service) for local services were protected. This also permitted the later adoption of 500 (follow-me), 600 (Canadian wireless), 700 (interconnect services), 800 (toll-free calls), and 900 (added-value information services).

As a result only 144 Area Codes were possible. Initially 86 were used and were assigned so that numbers more rapidly dialed on rotary dials went to districts with larger incoming traffic (e.g., 212 for Manhattan). The remaining 58 codes were reserved for later assignment.

This pool of 58 new Area Codes was sufficient for more than four decades. Finally, when more than 144 Area Codes were needed, new Area Codes were created by relaxing the restriction that the middle digit be only 0 or 1. On January 15, 1995, the first Area Code with a middle digit other than 0 or 1 was put into service, in Alabama. The present restrictions on area codes are that the first digit cannot be 0 or 1, the middle digit cannot be 9, and the last two digits cannot be the same. As of the beginning of 2000, 108 new Area Codes had been started, this great demand due in part to expanded use of the telephone networks for other services such as fax and cell phones, in part to political pressure from jurisdictions such as the Caribbean islands that wanted their own area codes, and in part by the large number of new telephone companies offering service and therefore needing at least one entire exchange in every rate billing district. Some people believe that the North American Numbering Plan (NANP) will run out of area codes before 2025, and there are various proposals for how to deal with that.

The transition in 1995 went remarkably smoothly, considering that every telephone exchange in North America required upgrading, both in revised software and, in some cases, new hardware. By and large the public was not aware of the significance of the change. This was a result of the generally high quality of North American telephone service, and the fact that the industry was tightly coordinated. The only glitches seem to have been that a few PBX (Private Branch eXchanges) designed by independent suppliers were not upgraded in time. Since 1995 the telecommunications industry in North America has changed greatly: it now has less central control, much more competition, and a much wider variety of services offered. Future changes in the numbering plan will surely result in much greater turmoil and inconvenience to the public.

2.2.4 IP Addresses

Another example of the need to reserve capacity for future use is afforded by IP (Internet Protocol) addresses, which is described in Section 2.8. These are (in version 4) of the form `x.x.x.x` where each `x` is a number between 0 and 255, inclusive. Thus each Internet address can be coded in a total of 32 bits. IP addresses are assigned by the Internet Assigned Numbers Authority, http://www.iana.org/, (IANA).

The explosion of interest in the Internet has created a large demand for IP addresses, and the organizations that participated in the development of the Internet, who had been assigned large blocks of numbers, began to feel as though they were hoarding a valuable resource. Among these organizations are AT&T, BBN, IBM, Xerox, HP, DEC, Apple, MIT, Ford, Stanford, BNR, Prudential, duPont, Merck, the U.S. Postal Service, and several U.S. DoD agencies (see Section 2.8). The U.S. electric power industry, in the form of EPRI (Electric Power Research Institute), requested a large number of Internet addresses, for every billable household or office suite, for eventual use by remote meter reading equipment. The Internet Engineering Task Force, http://www.ietf.org/, (IETF) came to realize that Internet addresses were needed on a much more pervasive and finer scale than had been originally envisioned—for example, there will be a need for addresses for appliances such as refrigerators, ovens, telephones, and furnaces when these are Internet-enabled, and there will be several needed within every automobile and truck, perhaps one for each microprocessor and sensor on the vehicle. The result has been the development of version 6, IPv6, in which each address is still of the form `x.x.x.x`, but each `x` is now a 32-bit number between 0 and 4,294,967,295 inclusive. Thus new Internet addresses will require 128 bits. Existing addresses will not have to change, but all the network equipment will have to change to accommodate the longer addresses. The new allocations include large blocks which are reserved for future expansion, and it is said (humorously) that there are blocks of addresses set aside for use by the other planets. The size of the address space is large enough to accommodate a unique hardware identifier for each personal computer, and some privacy advocates have pointed out that IPv6 may make anonymous Web surfing impossible.

2.2.5 ASCII

A fourth use for spare capacity in codes is to use some of it for denoting formatting or control operations. Many codes incorporate code patterns that are not data but control codes. For example, the Genetic Code includes three patterns of the 64 as stop codes to terminate the production of the protein.

The most commonly used code for text characters, ASCII (American Standard Code for Information Interchange, described in Section 2.5) reserves 33 of its 128 codes explicitly for control, and only 95 for characters. These 95 include the 26 upper-case and 26 lower-case letters of the English alphabet, the 10 digits, space, and 32 punctuation marks.

2.3 Extension of Codes

Many codes are designed by humans. Sometimes codes are amazingly robust, simple, easy to work with, and extendable. Sometimes they are fragile, arcane, complex, and defy even the simplest generalization. Often a simple, practical code is developed for representing a small number of items, and its success draws attention and people start to use it outside its original context, to represent a larger class of objects, for purposes not originally envisioned.

Codes that are generalized often carry with them unintended biases from their original context. Sometimes the results are merely amusing, but in other cases such biases make the codes difficult to work with.

An example of a reasonably benign bias is the fact that ASCII has two characters that were originally intended to be ignored. ASCII started as the 7-bit pattern of holes on paper tape, used to transfer information to and from teletype machines. The tape originally had no holes (except a series of small holes, always present, to align and feed the tape), and travelled through a punch. The tape could be punched either from a received transmission, or by a human typing on a keyboard. The debris from this punching operation was known as "chad." The leader (the first part of the tape) was unpunched, and therefore represented, in effect, a series of the character 0000000 of undetermined length (0 is represented as no hole). Of course when the tape was read the leader should be ignored, so by convention the character 0000000 was called NUL and was ignored. Later, when ASCII was used in computers, different systems treated NULs differently. Unix treats NUL as the end of a word in some circumstances, and this use interferes with applications in which characters are given a numerical interpretation. The other ASCII code which was originally intended to be ignored is DEL, 1111111. This convention was helpful to typists who could "erase" an error by backing up the tape and punching out every hole. In modern contexts DEL is often treated as a destructive backspace, but some text editors in the past have used DEL as a forward delete character, and sometimes it is simply ignored.

A much more serious bias carried by ASCII is the use of two characters, CR (carriage return) and LF (line feed), to move to a new printing line. The physical mechanism in teletype machines had separate hardware to move the paper (on a continuous roll) up, and reposition the printing element to the left margin. The engineers who designed the code that evolved into ASCII surely felt they were doing a good thing by permitting these operations to be called for separately. They could not have imagined the grief they have given to later generations as ASCII was adapted to situations with different hardware and no need to move the point of printing as called for by CR or LF separately. Different computing systems do things differently—Unix uses LF for a new line and ignores CR, Macintoshes (at least prior to OS X) use CR and ignore LF, and DOS/Windows requires both. This incompatibility is a continuing, serious source of frustration and errors. For example, in the transfer of files using FTP (File Transfer Protocol) CR and LF should be converted to suit the target platform for text files, but not for binary files. Some FTP programs infer the file type (text or binary) from the file extension (the part of the file name following the last period). Others look inside the file and count the number of "funny characters." Others rely on human input. These techniques usually work but not always. File extension conventions are not universally followed. Humans make errors. What if part of a file is text and part binary?

2.4 Fixed-Length and Variable-Length Codes

A decision that must be made very early in the design of a code is whether to represent all symbols with codes of the same number of bits (fixed length) or to let some symbols use shorter codes than others (variable length). There are advantages to both schemes.

Fixed-length codes are usually easier to deal with because both the coder and decoder know in advance how many bits are involved, and it is only a matter of setting or reading the values. With variable-length codes, the decoder needs a way to determine when the code for one symbol ends and the next one begins.

Fixed-length codes can be supported by parallel transmission, in which the bits are communicated from the coder to the decoder simultaneously, for example by using multiple wires to carry the voltages. This approach should be contrasted with serial transport of the coded information, in which a single wire sends a stream of bits and the decoder must decide when the bits for one symbol end and those for the next symbol start. If a decoder gets mixed up, or looks at a stream of bits after it has started, it might not know. This is referred to as a "framing error." To eliminate framing errors, stop bits are often sent between symbols; typically ASCII sent over serial lines has 1 or 2 stop bits, normally given the value 0. Thus if a decoder is out of step, it will eventually find a 1 in what it assumed should be a stop bit, and it can try to resynchronize. Although in theory framing errors could persist for long periods, in practice use of stop bits works well.

2.4.1 Morse Code

An example of a variable-length code is Morse Code, developed for the telegraph. The codes for letters, digits, and punctuation are sequences of dots and dashes with short-length intervals between them. See Section 2.9.

The decoder tells the end of the code for a single character by noting the length of time before the next dot or dash. The intra-character separation is the length of a dot, and the inter-character separation is longer, the length of a dash. The inter-word separation is even longer.

2.5 Detail: ASCII

ASCII, which stands for "The American Standard Code for Information Interchange," was introduced by the American National Standards Institute (ANSI) in 1963. It is the most commonly used character code.

ASCII is a seven-bit code, representing the 33 control characters and 95 printing characters (including space) in Table 2.2. The control characters are used to signal special conditions, as described in Table 2.3.

Control Characters

HEX	DEC	CHR	Ctrl
00	0	NUL	^@
01	1	SOH	^A
02	2	STX	^B
03	3	ETX	^C
04	4	EOT	^D
05	5	ENQ	^E
06	6	ACK	^F
07	7	BEL	^G
08	8	BS	^H
09	9	HT	^I
0A	10	LF	^J
0B	11	VT	^K
0C	12	FF	^L
0D	13	CR	^M
0E	14	SO	^N
0F	15	SI	^O
10	16	DLE	^P
11	17	DC1	^Q
12	18	DC2	^R
13	19	DC3	^S
14	20	DC4	^T
15	21	NAK	^U
16	22	SYN	^V
17	23	ETB	^W
18	24	CAN	^X
19	25	EM	^Y
1A	26	SUB	^Z
1B	27	ESC	^[
1C	28	FS	^\
1D	29	GS	^]
1E	30	RS	^^
1F	31	US	^_

Digits

HEX	DEC	CHR
20	32	SP
21	33	!
22	34	"
23	35	#
24	36	$
25	37	%
26	38	&
27	39	’
28	40	(
29	41	)
2A	42	*
2B	43	+
2C	44	,
2D	45	-
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	¡
3D	61	=
3E	62	>
3F	63	?

Uppercase

HEX	DEC	CHR
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R
53	83	S
54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z
5B	91	[
5C	92	\
5D	93	]
5E	94	^
5F	95	_

Lowercase

HEX	DEC	CHR
60	96	‘
61	97	a
62	98	b
63	99	c
64	100	d
65	101	e
66	102	f
67	103	g
68	104	h
69	105	i
6A	106	j
6B	107	k
6C	108	l
6D	109	m
6E	110	n
6F	111	o
70	112	p
71	113	q
72	114	r
73	115	s
74	116	t
75	117	u
76	118	v
77	119	w
78	120	x
79	121	y
7A	122	z
7B	123	{
7C	124	—
7D	125	}
7E	126	~
7F	127	DEL

Table 2.2: ASCII Character Set

On to 8 Bits

In an 8-bit context, ASCII characters follow a leading 0, and thus may be thought of as the "bottom half" of a larger code. The 128 characters represented by codes between HEX 80 and HEX FF (sometimes incorrectly called "high ASCII" of "extended ASCII") have been defined differently in different contexts. On many operating systems they included the accented Western European letters and various additional

HEX	DEC	CHR	Ctrl	Meaning
00	0	NUL	^@	NULl blank leader on paper tape; generally ignored
01	1	SOH	^A	Start Of Heading
02	2	STX	^B	Start of TeXt
03	3	ETX	^C	End of TeXt; matches STX
04	4	EOT	^D	End Of Transmission
05	5	ENQ	^E	ENQuiry
06	6	ACK	^F	ACKnowledge; affirmative response to ENQ
07	7	BEL	^G	BELl; audible signal, a bell on early machines
08	8	BS	^H	BackSpace; nondestructive, ignored at left margin
09	9	HT	^I	Horizontal Tab
0A	10	LF	^J	Line Feed; paper up or print head down; new line on Unix
0B	11	VT	^K	Vertical Tab
0C	12	FF	^L	Form Feed; start new page
0D	13	CR	^M	Carriage Return; print head to left margin; new line on Macs
0E	14	SO	^N	Shift Out; start use of alternate character set
0F	15	SI	^O	Shift In; resume use of default character set
10	16	DLE	^P	Data Link Escape; changes meaning of next character
11	17	DC1	^Q	Device Control 1; if flow control used, XON, OK to send
12	18	DC2	^R	Device Control 2
13	19	DC3	^S	Device Control 3; if flow control used, XOFF, stop sending
14	20	DC4	^T	Device Control 4
15	21	NAK	^U	Negative AcKnowledge; response to ENQ
16	22	SYN	^V	SYNchronous idle
17	23	ETB	^W	End of Transmission Block
18	24	CAN	^X	CANcel; disregard previous block
19	25	EM	^Y	End of Medium
1A	26	SUB	^Z	SUBstitute
1B	27	ESC	^[	ESCape; changes meaning of next character
1C	28	FS	^\	File Separator; coarsest scale
1D	29	GS	^]	Group Separator; coarse scale
1E	30	RS	^^	Record Separator; fine scale
1F	31	US	^_	Unit Separator; finest scale
20	32	SP		SPace; usuallly not considered a control character
7F	127	DEL		DELete; orginally ignored; sometimes destructive backspace

Table 2.3: ASCII control characters

punctuation marks. On IBM PCs they included line-drawing characters. Macs used (and still use) a different encoding.

Fortunately, people now appreciate the need for interoperability of computer platforms, so more universal standards are coming into favor. The most common code in use for Web pages is ISO-8859-1 (ISO-Latin) which uses the 96 codes between HEX A0 and HEX FF for various accented letters and punctuation of Western European languages, and a few other symbols. The 32 characters between HEX 80 and HEX 9F are reserved as control characters in ISO-8859-1.

Nature abhors a vacuum. Most people don't want 32 more control characters (indeed, of the 33 control characters in 7-bit ASCII, only about ten are regularly used in text). Consequently there has been no end of ideas for using HEX 80 to HEX 9F. The most widely used convention is Microsoft's Windows Code Page 1252 (Latin I) which is the same as ISO-8859-1 (ISO-Latin) except that 27 of the 32 control codes are assigned to printed characters, one of which is HEX 80, the Euro currency character. Not all platforms and operating systems recognize CP-1252, so documents, and in particular Web pages, require special attention.

Beyond 8 Bits

To represent Asian languages, many more characters are needed. There is currently active development of appropriate standards, and it is generally felt that the total number of characters that need to be represented is less that 65,536. This is fortunate because that many different characters could be represented in 16 bits, or 2 bytes. In order to stay within this number, the written versions of some of the Chinese dialects must share symbols that look alike.

The strongest candidate for a 2-byte standard character code today is known as Unicode.

References

There are many Web pages that give the ASCII chart, with extensions to all the world's languages. Among the more useful:

- Jim Price, with PC and Windows 8-bit charts, and several further links
  http://www.jimprice.com/jim-asc.shtml
- A Brief History of Character Codes, with a discussion of extension to Asian languages
  http://tronweb.super-nova.co.jp/characcodehist.html
- Unicode home page
  http://www.unicode.org/
- Windows CP-1252 standard, definitive
  http://www.microsoft.com/globaldev/reference/sbcs/1252.htm
- CP-1252 compared to:
  - Unicode
    http://ftp.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WINDOWS/CP1252.TXT
  - Unicode/HTML
    http://www.alanwood.net/demos/ansi.html
  - ISO-8859-1/Mac OS
    http://www.jwz.org/doc/charsets.html

2.6 Detail: Integer Codes

There are many ways to represent integers as bit patterns. All suffer from an inability to represent arbitrarily large integers in a fixed number of bits. A computation which produces an out-of-range result is said to overflow.

The most commonly used representations are binary code for unsigned integers (e.g., memory addresses), 2's complement for signed integers (e.g., ordinary arithmetic), and binary gray code for instruments measuring changing quantities.

The following table gives five examples of 4-bit integer codes. The **MSB** (most significant bit) is on the left and the **LSB** (least significant bit) on the right.

	Unsigned Integers		Signed Integers		
	Binary Code	Binary Gray Code	2's Complement	Sign/Magnitude	1's Complement
Range →	[0, 15]	[0, 15]	[-8, 7]	[-7,7]	[-7,7]
-8			1 0 0 0		
-7			1 0 0 1	1 1 1 1	1 0 0 0
-6			1 0 1 0	1 1 1 0	1 0 0 1
-5			1 0 1 1	1 1 0 1	1 0 1 0
-4			1 1 0 0	1 1 0 0	1 0 1 1
-3			1 1 0 1	1 0 1 1	1 1 0 0
-2			1 1 1 0	1 0 1 0	1 1 0 1
-1			1 1 1 1	1 0 0 1	1 1 1 0
0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1	0 0 1 0	0 0 1 0	0 0 1 0
3	0 0 1 1	0 0 1 0	0 0 1 1	0 0 1 1	0 0 1 1
4	0 1 0 0	0 1 1 0	0 1 0 0	0 1 0 0	0 1 0 0
5	0 1 0 1	0 1 1 1	0 1 0 1	0 1 0 1	0 1 0 1
6	0 1 1 0	0 1 0 1	0 1 1 0	0 1 1 0	0 1 1 0
7	0 1 1 1	0 1 0 0	0 1 1 1	0 1 1 1	0 1 1 1
8	1 0 0 0	1 1 0 0			
9	1 0 0 1	1 1 0 1			
10	1 0 1 0	1 1 1 1			
11	1 0 1 1	1 1 1 0			
12	1 1 0 0	1 0 1 0			
13	1 1 0 1	1 0 1 1			
14	1 1 1 0	1 0 0 1			
15	1 1 1 1	1 0 0 0			

Table 2.4: Four-bit integer codes

Binary Code

This code is for nonnegative integers. For code of length n, the 2^n patterns represent integers 0 through $2^n - 1$. The LSB (least significant bit) is 0 for even and 1 for odd integers.

Binary Gray Code

This code is for nonnegative integers. For code of length n, the 2^n patterns represent integers 0 through $2^n - 1$. The two bit patterns of adjacent integers differ in exactly one bit. This property makes the code useful for sensors where the integer being encoded might change while a measurement is in progress. The following anonymous tribute appeared in Martin Gardner's column "Mathematical Games" in Scientific American, August, 1972, but actually was known much earlier.

The Binary Gray Code is fun,
for with it STRANGE THINGS can be done...
Fifteen, as you know,
is one oh oh oh,
while ten is one one one one.

2's Complement

This code is for integers, both positive and negative. For a code of length n, the 2^n patterns represent integers -2^{n-1} through $2^{n-1} - 1$. The LSB (least significant bit) is 0 for even and 1 for odd integers. Where they overlap, this code is the same as binary code. This code is widely used.

Sign/Magnitude

This code is for integers, both positive and negative. For code of length n, the 2^n patterns represent integers $-(2^{n-1} - 1)$ through $2^{n-1} - 1$. The MSB (most significant bit) is 0 for positive and 1 for negative integers; the other bits carry the magnitude. Where they overlap, this code is the same as binary code. While conceptually simple, this code is awkward in practice. Its separate representations for +0 and -0 are not generally useful.

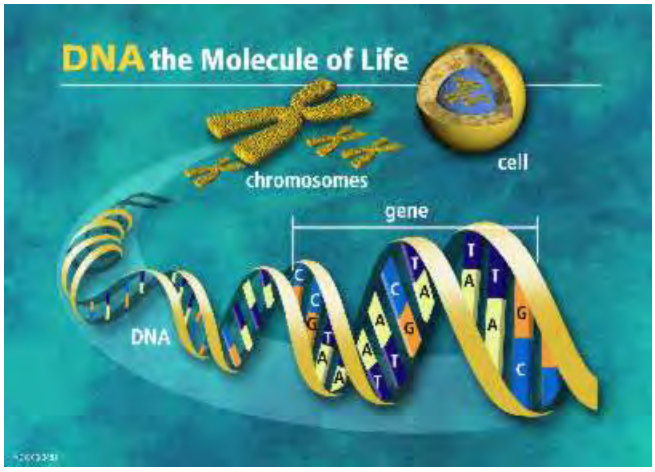
1's Complement

This code is for integers, both positive and negative. For code of length n, the 2^n patterns represent integers $-(2^{n-1} - 1)$ through $2^{n-1} - 1$. The MSB is 0 for positive integers; negative integers are formed by complementing each bit of the corresponding positive integer. Where they overlap, this code is the same as binary code. This code is awkward and rarely used today. Its separate representations for +0 and -0 are not generally useful.

2.7 Detail: The Genetic Code*

The basic building block of your body is a cell. Two or more groups of cells form tissues, such as bone or muscle; tissues organize to form organs, such as the heart or brain; organs form organ systems, such as the circulatory system or nervous system; the organ systems together form you, the organism. Cells can be classified as either eukaryote or prokaryote cells – with or without a nucleus, respectively. The cells that make up your body and those of all animals, plants, and fungi are eukaryotic. Prokaryotes are bacteria and cyanobacteria.

The nucleus forms a separate compartment from the rest of the cell body; this compartment serves as the central storage center for all the hereditary information of the eukaryote cells. All of the genetic information that forms the book of life is stored on individual chromosomes found within the nucleus. In healthy humans there are 23 pairs of chromosomes (46 total). Each one of the chromosomes contains one threadlike deoxyribonucleic acid (DNA) molecule. Genes are the functional regions along these DNA strands, and are the fundamental physical units that carry hereditary information from one generation to the next. In the prokaryotes the chromosomes are free floating in the cell body since there is no nucleus.



Courtesy of the Genomics Management Information System,
U.S. Department of Energy Genome Programs, http://genomics.energy.gov.

Figure 2.2: Location of DNA inside of a Cell

The DNA molecules are composed of two interconnected chains of nucleotides that form one DNA strand. Each nucleotide is composed of a sugar, phosphate, and one of four bases. The bases are adenine, guanine, cytosine, and thymine. For convenience each nucleotide is referenced by its base; instead of saying deoxyguanosine monophosphate we would simply say guanine (or G) when referring to the individual nucleotide. Thus we could write CCACCA to indicate a chain of interconnected cytosine-cytosine-adenine-cytosine-cytosine-adenine nucleotides.

The individual nucleotide chains are interconnected through the pairing of their nucleotide bases into a single double helix structure. The rules for pairing are that cytosine always pairs with guanine and thymine always pairs with adenine. These DNA chains are replicated during somatic cell division (that is, division of all cells except those destined to be sex cells) and the complete genetic information is passed on to the resulting cells.

Genes are part of the chromosomes and coded for on the DNA strands. Individual functional sections of the threadlike DNA are called genes. The information encoded in genes directs the maintenance and development of the cell and organism. This information travels a path from the input to the output: DNA (genes) $\Rightarrow$ mRNA (messenger ribonucleic acid) $\Rightarrow$ ribosome/tRNA $\Rightarrow$ Protein. In essence the protein is the final output that is generated from the genes, which serve as blueprints for the individual proteins.

*This section is based on notes written by Tim Wagner

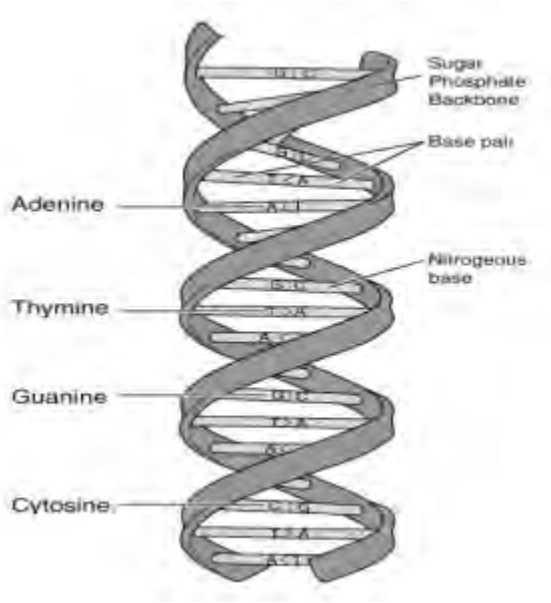


Figure 2.3: A schematic of DNA showing its helical structure

The proteins themselves can be structural components of your body (such as muscle fibers) or functional components (enzymes that help regulate thousands of biochemical processes in your body). Proteins are built from polypeptide chains, which are just strings of amino acids (a single polypeptide chain constitutes a protein, but often functional proteins are composed of multiple polypeptide chains).

The genetic message is communicated from the cell nucleus's DNA to ribosomes outside the nucleus via messenger RNA (ribosomes are cell components that help in the eventual construction of the final protein). Transcription is the process in which messenger RNA is generated from the DNA. The messenger RNA is a copy of a section of a single nucleotide chain. It is a single strand, exactly like DNA except for differences in the nucleotide sugar and that the base thymine is replaced by uracil. Messenger RNA forms by the same base pairing rule as DNA except T is replaced by U (C to G, U to A).

This messenger RNA is translated in the cell body, with the help of ribosomes and tRNA, into a string of amino acids (a protein). The ribosome holds the messenger RNA in place and the transfer RNA places the appropriate amino acid into the forming protein, illustrated schematically in Figure 2.4.

The messenger RNA is translated into a protein by first docking with a ribosome. An initiator tRNA binds to the ribosome at a point corresponding to a start codon on the mRNA strand – in humans this corresponds to the AUG codon. This tRNA molecule carries the appropriate amino acid called for by the codon and matches up at with the mRNA chain at another location along its nucleotide chain called an anticodon. The bonds form via the same base pairing rule for mRNA and DNA (there are some pairing exceptions that will be ignored for simplicity). Then a second tRNA molecule will dock on the ribosome of the neighboring location indicated by the next codon. It will also be carrying the corresponding amino acid that the codon calls for. Once both tRNA molecules are docked on the ribosome the amino acids that they are carrying bond together. The initial tRNA molecule will detach leaving behind its amino acid on a now growing chain of amino acids. Then the ribosome will shift over one location on the mRNA strand to make room for another tRNA molecule to dock with another amino acid. This process will continue until a stop codon is read on the mRNA; in humans the termination factors are UAG, UAA, and UGA. When the stop codon is read the chain of amino acids (protein) will be released om the ribosome structure.

What are amino acids? They are organic compounds with a central carbon atom, to which is attached by covalent bonds

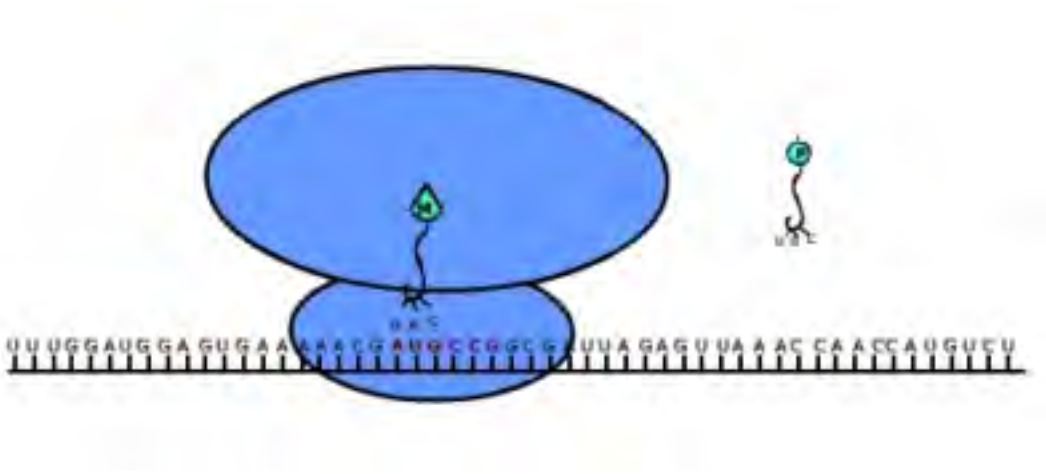


Figure 2.4: RNA to Protein transcription (click on figure for the online animation at http://www.mtl.mit.edu/Courses/6.050/2008/notes/rna-to-proteins.html)

- a single hydrogen atom H
- an amino group NH_2
- a carboxyl group COOH
- a side chain, different for each amino acid

The side chains range in complexity from a single hydrogen atom (for the amino acid glycine), to structures incorporating as many as 18 atoms (arginine). Thus each amino acid contains between 10 and 27 atoms. Exactly twenty different amino acids (sometimes called the "common amino acids") are used in the production of proteins as described above. Ten of these are considered "essential" because they are not manufactured in the human body and therefore must be acquired through eating (arginine is essential for infants and growing children). Nine amino acids are hydrophilic (water-soluble) and eight are hydrophobic (the other three are called "special"). Of the hydrophilic amino acids, two have net negative charge in their side chains and are therefore acidic, three have a net positive charge and are therefore basic; and four have uncharged side chains. Usually the side chains consist entirely of hydrogen, nitrogen, carbon, and oxygen atoms, although two (cysteine and methionine) have sulfur as well.

There are twenty different common amino acids that need to be coded and only four different bases. How is this done? As single entities the nucleotides (A,C,T, or G) could only code for four amino acids, obviously not enough. As pairs they could code for 16 (4^2) amino acids, again not enough. With triplets we could code for 64 (4^3) possible amino acids – this is the way it is actually done in the body, and the string of three nucleotides together is called a codon. Why is this done? How has evolution developed such an inefficient code with so much redundancy? There are multiple codons for a single amino acid for two main biological reasons: multiple tRNA species exist with different anticodons to bring certain amino acids to the ribosome, and errors/sloppy pairing can occur during translation (this is called wobble).

Codons, strings of three nucleotides, thus code for amino acids. In the tables below are the genetic code, from the messenger RNA codon to amino acid, and various properties of the amino acids[1] In the tables below * stands for (U, C, A, or G); thus CU* could be either CUU, CUC, CUA, or CUG.

[1]shown are the one-letter abbreviation for each, its molecular weight, and some of its properties, taken from H. Lodish, D. Baltimore, A. Berk, S. L. Zipursky, P. Matsudaira, and J. Darnell, "Molecular Cell Biology," third edition, W. H. Freeman and Company, New York, NY; 1995.

First Nucleotide Base of mRNA Codon	Second Nucleotide Base of mRNA Codon: U	C	A	G
U	UUU = Phe UUC = Phe UUA = Leu UUG = Leu	UC* = Ser	UAU = Tyr UAC = Tyr UAA = stop UAG = stop	UGU = Cys UGC = Cys UGA = stop UGG = Trp
C	CU* = Leu	CC* = Pro	CAU = His CAC = His CAA = Gln CAG = Gln	CG* = Arg
A	AUU = Ile AUC = Ile AUA = Ile AUG = Met (start)	AC* = Thr	AAU = Asn AAC = Asn AAA = Lys AAG = Lys	AGU = Ser AGC = Ser AGA = Arg AGG = Arg
G	GU* = Val	GC* = Ala	GAU = Asp GAC = Asp GAA = Glu GAG = Glu	GG* = Gly

Table 2.5: Condensed chart of Amino Acids

Symbols		Amino Acid	M Wt	Properties		Codon(s)
Ala	A	Alanine	89.09	Non-essential	Hydrophobic	GC*
Arg	R	Arginine	174.20	Essential	Hydrophilic, basic	CG* AGA AGG
Asn	N	Asparagine	132.12	Non-essential	Hydrophilic, uncharged	AAU AAC
Asp	D	Aspartic Acid	133.10	Non-essential	Hydrophilic, acidic	GAU GAC
Cys	C	Cysteine	121.15	Non-essential	Special	UGU UGC
Gln	Q	Glutamine	146.15	Non-essential	Hydrophilic, uncharged	CAA CAG
Glu	E	Glutamic Acid	147.13	Non-essential	Hydrophilic, acidic	GAA GAG
Gly	G	Glycine	75.07	Non-essential	Special	GG*
His	H	Histidine	155.16	Essential	Hydrophilic, basic	CAU CAC
Ile	I	Isoleucine	131.17	Essential	Hydrophobic	AUU AUC AUA
Leu	L	Leucine	131.17	Essential	Hydrophobic	UUA UUG CU*
Lys	K	Lysine	146.19	Essential	Hydrophilic, basic	AAA AAG
Met	M	Methionine	149.21	Essential	Hydrophobic	AUG
Phe	F	Phenylalanine	165.19	Essential	Hydrophobic	UUU UUC
Pro	P	Proline	115.13	Non-essential	Special	CC*
Ser	S	Serine	105.09	Non-essential	Hydrophilic, uncharged	UC* AGU AGC
Thr	T	Threonine	119.12	Essential	Hydrophilic, uncharged	AC*
Trp	W	Tryptophan	204.23	Essential	Hydrophobic	UGG
Tyr	Y	Tyrosine	181.19	Non-essential	Hydrophobic	UAU UAC
Val	V	Valine	117.15	Essential	Hydrophobic	GU*
start		Methionine				AUG
stop						UAA UAG UGA

Table 2.6: The Amino Acids and some properties

2.8 Detail: IP Addresses

Table 2.7 is an excerpt from IPv4, http://www.iana.org/assignments/ipv4-address-space (version 4, which is in the process of being phased out in favor of version 6). IP addresses are assigned by the Internet Assigned Numbers Authority (IANA), http://www.iana.org/.

IANA is in charge of all "unique parameters" on the Internet, including IP (Internet Protocol) addresses. Each domain name is associated with a unique IP address, a numerical name consisiting of four blocks of up to three digits each, e.g. 204.146.46.8, which systems use to direct information through the network.

Internet Protocol Address Space

The allocation of Internet Protocol version 4 (IPv4) address space to various registries is listed here. Originally, all the IPv4 address spaces was managed directly by the IANA. Later, parts of the address space were allocated to various other registries to manage for particular purposes or regions of the world. RFC 1466 documents most of these allocations.

Address Block	Registry - Purpose	Date
000/8	IANA - Reserved	Sep 81
001/8	IANA - Reserved	Sep 81
002/8	IANA - Reserved	Sep 81
003/8	General Electric Company	May 94
004/8	Bolt Beranek and Newman Inc.	Dec 92
005/8	IANA - Reserved	Jul 95
006/8	Army Information Systems Center	Feb 94
007/8	IANA - Reserved	Apr 95
008/8	Bolt Beranek and Newman Inc.	Dec 92
009/8	IBM	Aug 92
010/8	IANA - Private Use	Jun 95
011/8	DoD Intel Information Systems	May 93
012/8	AT & T Bell Laboratories	Jun 95
013/8	Xerox Corporation	Sep 91
014/8	IANA - Public Data Network	Jun 91
015/8	Hewlett-Packard Company	Jul 94
016/8	Digital Equipment Corporation	Nov 94
017/8	Apple Computer Inc.	Jul 92
018/8	MIT	Jan 94
019/8	Ford Motor Company	May 95
020/8	Computer Sciences Corporation	Oct 94
021/8	DDN-RVN	Jul 91
022/8	Defense Information Systems Agency	May 93
023/8	IANA - Reserved	Jul 95
024/8	IANA - Cable Block	Jul 95
025/8	Royal Signals and Radar Establishment	Jan 95
⋮		

Table 2.7: IP Address Assignments - partial list

2.9 Detail: Morse Code

Samuel F. B. Morse (1791–1872) was a landscape and portrait painter from Charleston, MA. He frequently travelled from his studio in New York City to work with clients across the nation. He was in Washington, DC in 1825 when his wife Lucretia died suddenly of heart failure. Morse learned of this event as rapidly as was possible at the time, through a letter sent from New York to Washington, but it was too late for him to return in time for her funeral.

As a painter Morse met with only moderate success. Although his paintings can be found today in major museums—the Museum of Fine Arts, Boston, has seven—he never had an important impact on contemporary art. It was as an inventor that he is best known. (He combined his interest in technology and his passion for art in an interesting way: in 1839 he learned the French technique of making daguerreotypes and for a few years supported himself by teaching it to others.)

Returning from Europe in 1832, he happened to meet a fellow passenger who had visited the great European physics laboratories. He learned about the experiments of Ampère, Franklin, and others wherein electricity passed instantaneously over any known length of wire. Morse realized this meant that intelligence could be transmitted instantaneously by electricity. He understood from the circumstances of his wife's death the need for rapid communication. Before his ship even arrived in New York he invented the first version of what is today called Morse Code. His later inventions included the hand key and some receiving devices. It was in 1844 that he sent his famous message WHAT HATH GOD WROUGHT from Washington to Baltimore. That event caught the public fancy, and produced national excitement not unlike the Internet euphoria 150 years later.

Morse Code consists of a sequence of short and long pulses or tones (dots and dashes) separated by short periods of silence. A person generates Morse Code by making and breaking an electrical connection on a hand key, and the person on the other end of the line listens to the sequence of dots and dashes and converts them to letters, spaces, and punctuation. The modern form of Morse Code is shown in Table 2.8. The at sign was added in 2004 to accommodate email addresses. Two of the dozen or so control codes are shown. Non-English letters and some of the less used punctuation marks are omitted.

A	·–	K	–·–	U	··–	0	–––––	Question mark	··––··
B	–···	L	·–··	V	···–	1	·––––	Apostrophe	·––––·
C	–·–·	M	––	W	·––	2	··–––	Parenthesis	–·––·–
D	–··	N	–·	X	–··–	3	···––	Quotation mark	·–··–·
E	·	O	–––	Y	–·––	4	····–	Fraction bar	–··–·
F	··–·	P	·––·	Z	––··	5	·····	Equals	–···–
G	––·	Q	––·–	Period	·–·–·–	6	–····	Slash	–··–·
H	····	R	·–·	Comma	––··––	7	––···	At sign	·––·–·
I	··	S	···	Hyphen	–····–	8	–––··	Delete prior word	·····
J	·–––	T	–	Colon	–––···	9	––––·	End of Transmission	·–·–·

Table 2.8: Morse Code

If the duration of a dot is taken to be one unit of time then that of a dash is three units. The space between the dots and dashes within one character is one unit, that between characters is three units, and that between words seven units. Space is not considered a character, as it is in ASCII.

Unlike ASCII, Morse Code is a variable-length code. Morse realized that some letters in the English alphabet are more frequently used than others, and gave them shorter codes. Thus messages could be transmitted faster on average, than if all letters were equally long. Table 2.9 shows the frequency of the letters in written English (the number of times each letter is, on average, found per 1000 letters).

Morse Code was well designed for use on telegraphs, and it later saw use in radio communications before AM radios could carry voice. Until 1999 it was a required mode of communication for ocean vessels, even though it was rarely used (the theory apparently was that some older craft might not have converted to more modern communications gear). Ability to send and receive Morse Code is still a requirement for U.S.

132	E	61	S	24	U
104	T	53	H	20	G, P, Y
82	A	38	D	19	W
80	O	34	L	14	B
71	N	29	F	9	V
68	R	27	C	4	K
63	I	25	M	1	X, J, Q, Z

Table 2.9: Relative frequency of letters in written English

citizens who want some types of amateur radio license.

Since Morse Code is designed to be heard, not seen, Table 2.8 is only marginally useful. You cannot learn Morse Code from looking at the dots and dashes on paper; you have to hear them. If you want to listen to it on text of your choice, try a synthesizer on the Internet, such as

- http://morsecode.scphillips.com/jtranslator.html

A comparison of Tables 2.8 and 2.9 reveals that Morse did a fairly good job of assigning short sequences to the more common letters. It is reported that he did this not by counting letters in books and newspapers, but by visiting a print shop. The printing presses at the time used movable type, with separate letters assembled by humans into lines. Each letter was available in multiple copies for each font and size, in the form of pieces of lead. Morse simply counted the pieces of type available for each letter of the alphabet, assuming that the printers knew their business and stocked their cases with the right quantity of each letter. The wooden type cases were arranged with two rows, the capital letters in the upper one and small letters in the lower one. Printers referred to those from the upper row of the case as “uppercase” letters.

Chapter 3

Compression

In Chapter 1 we examined the fundamental unit of information, the bit, and its various abstract representations: the Boolean bit, the circuit bit, the control bit, the physical bit, the quantum bit, and the classical bit. Our never-ending quest for improvement made us want representations of single bits that are smaller, faster, stronger, smarter, safer, and cheaper.

In Chapter 2 we considered some of the issues surrounding the representation of complex objects by arrays of bits (at this point, Boolean bits). The mapping between the objects to be represented (the symbols) and the array of bits used for this purpose is known as a code. We naturally want codes that are stronger and smaller, i.e., that lead to representations of objects that are both smaller and less susceptible to errors. In this chapter we will consider techniques of compression that can be used for generation of particularly efficient representations. In Chapter 4 we will look at techniques of avoiding errors.

In Chapter 2 we considered systems of the sort shown in Figure 3.1, in which symbols are encoded into bit strings, which are transported (in space and/or time) to a decoder, which then recreates the original symbols.

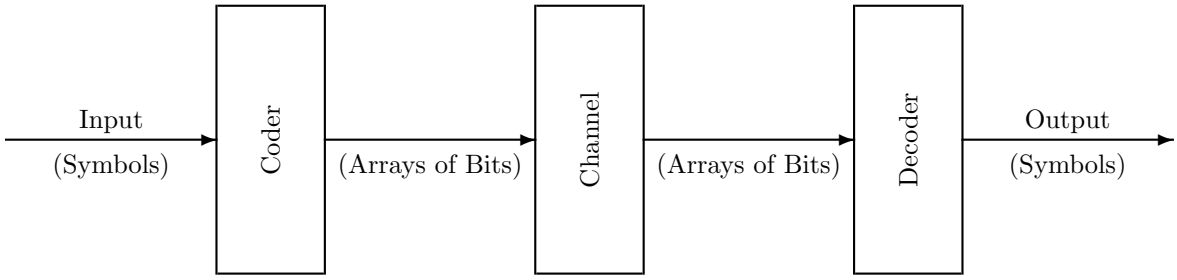


Figure 3.1: Generalized communication system

Typically the same code is used for a sequence of symbols, one after another. The role of data compression is to convert the string of bits representing a succession of symbols into a shorter string for more economical transmission, storage, or processing. The result is the system in Figure 3.2, with both a compressor and an expander. Ideally, the expander would exactly reverse the action of the compressor so that the coder and decoder could be unchanged.

On first thought, this approach might seem surprising. Why is there any reason to believe that the same

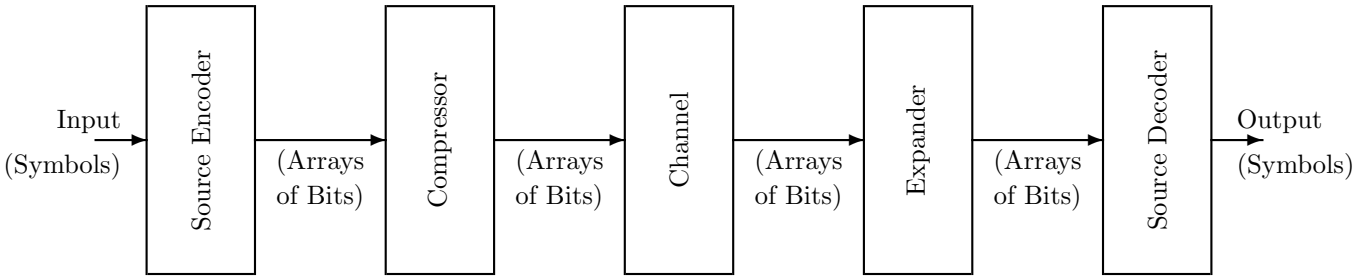


Figure 3.2: More elaborate communication system

information could be contained in a smaller number of bits? We will look at two types of compression, using different approaches:

- **Lossless** or **reversible** compression (which can only be done if the original code was inefficient, for example by having unused bit patterns, or by not taking advantage of the fact that some symbols are used more frequently than others)
- **Lossy** or **irreversible** compression, in which the original symbol, or its coded representation, cannot be reconstructed from the smaller version exactly, but instead the expander produces an approximation that is "good enough"

Six techniques are described below which are astonishingly effective in compressing data files. The first five are reversible, and the last one is irreversible. Each technique has some cases for which it is particularly well suited (the **best cases**) and others for which it is not well suited (the **worst cases**).

3.1 Variable-Length Encoding

In Chapter 2 Morse code was discussed as an example of a source code in which more frequently occurring letters of the English alphabet were represented by shorter codewords, and less frequently occurring letters by longer codewords. On average, messages sent using these codewords are shorter than they would be if all codewords had the same length. Variable-length encoding can be done either in the source encoder or the compressor. A general procedure for variable-length encoding will be given in Chapter 5, so a discussion of this technique is put off until that chapter.

3.2 Run Length Encoding

Suppose a message consists of long sequences of a small number of symbols or characters. Then the message could be encoded as a list of the symbol and the number of times it occurs. For example, the message "a B B B B B a a a B B a a a a" could be encoded as "a 1 B 5 a 3 B 2 a 4". This technique works very well for a relatively small number of circumstances. One example is the German flag, which could be encoded as so many black pixels, so many red pixels, and so many yellow pixels, with a great saving over specifying every pixel. Another example comes from fax technology, where a document is scanned and long groups of white (or black) pixels are transmitted as merely the number of such pixels (since there are only white and black pixels, it is not even necessary to specify the color since it can be assumed to be the other color).

Run length encoding does not work well for messages without repeated sequences of the same symbol. For example, it may work well for drawings and even black-and-white scanned images, but it does not work

Figure 3.3: Flag of Germany (black band on top, red in middle, yellow on bottom)

well for photographs since small changes in shading from one pixel to the next would require many symbols to be defined.

3.3 Static Dictionary

If a code has unused codewords, these may be assigned, as abbreviations, to frequently occurring sequences of symbols. Then such sequences could be encoded with no more bits than would be needed for a single symbol. For example, if English text is being encoded in ASCII and the DEL character is regarded as unnecessary, then it might make sense to assign its codeword 127 to the common word "the". Practical codes offer numerous examples of this technique. The list of the codewords and their meanings is called a codebook, or dictionary. The compression technique considered here uses a dictionary which is **static** in the sense that it does not change from one message to the next.

An example will illustrate this technique. Before the the electric telegraph, there were other schemes for transmitting messages long distances. A mechanical telegraph described in some detail by Wilson[1] was put in place in 1796 by the British Admiralty to communicate between its headquarters in London and various ports, including Plymouth, Yarmouth, and Deal. It consisted of a series of cabins, each within sight of the next one, with six large shutters which could rotated to be horizontal (open) or vertical (closed). See Figure 3.4. In operation, all shutters were in the open position until a message was to be sent. Then all shutters were closed to signal the start of a message (operators at the cabins were supposed to look for new messages every five minutes). Then the message was sent in a sequence of shutter patterns, ending with the all-open pattern.

There were six shutters, and therefore 64 (2^6) patterns. Two of these were control codes (all open was "start" and "stop," and all closed was "idle"). The other 62 were available for 24 letters of the alphabet (26 if J and U were included, which was not essential because they were recent additions to the English alphabet), 10 digits, an end-of-word marker, and an end-of-page marker. This left over 20 unused patterns which were assigned to commonly occurring words or phrases. The particular abbreviations used varied from time to time, but included common words like "the", locations such as "Portsmouth", and other words of importance such as "French", "Admiral", "east", and "frigate". It also included phrases like "Commander of the West India Fleet" and "To sail, the first fair wind, to the southward".

Perhaps the most intriguing entries in the codebook were "Court-Martial to sit" and "Sentence of court-martial to be put into execution". Were courts-martial really common enough and messages about them frequent enough to justify dedicating two out of 64 code patterns to them?

As this example shows, long messages can be shortened considerably with a set of well chosen abbreviations. However, there is an inherent risk: the effect of errors is apt to be much greater. If full text is transmitted, a single error causes a misspelling or at worst the wrong word, which humans can usually detect and correct. With abbreviations, a single incorrect bit could result in a possible but unintended meaning: "east" might be changed to "south", or a single letter might be changed to "Sentence of court-martial to be put into execution" with significant consequences.

This telegraph system worked well. During one demonstration a message went from London to Plymouth and back, a distance of 500 miles, in three minutes. That's 13 times the speed of sound.

If abbreviations are used to compress messages, there must be a codebook showing all the abbreviations in use, that is distributed before the first message is sent. Because it is distributed only once, the cost of

[1] Geoffrey Wilson, "The Old Telegraphs," Phillimore and Co., Ltd., London and Chichester, U.K.; 1976; pp. 11-32.

Figure 3.4: British shutter telegraph cabin, 1797, showing the six shutters closed, and an open window with a telescope to view another cabin (from T. Standage, "The Victorian Internet," Berkley Books, New York; 1998; p. 15).

distribution is low on a per-message basis. But it has to be carefully constructed to work with all messages expected, and cannot be changed to match the needs of individual messages.

This technique works well for sets of messages that are quite similar, as might have been the case with 18th-century naval communications. It is not well suited for more diverse messages. This mechanical telegraph was never adapted for commercial or public use, which would have given it more diverse set of messages without as many words or phrases in common.

3.4 Semi-adaptive Dictionary

The static-dictionary approach requires one dictionary, defined in advance, that applies to all messages. If a new dictionary could be defined for each message, the compression could be greater because the particular sequences of symbols found in the message could be made into dictionary entries.

Doing so would have several drawbacks, however. First, the new dictionary would have to be transmitted along with the encoded message, resulting in increased overhead. Second, the message would have to be analyzed to discover the best set of dictionary entries, and therefore the entire message would have to be available for analysis before any part of it could be encoded (that is, this technique has large **latency**). Third, the computer calculating the dictionary would need to have enough memory to store the entire message.

These disadvantages have limited the use of semi-adaptive dictionary compression schemes.

3.5 Dynamic Dictionary

What would be best for many applications would be an encoding scheme using a dictionary that is calculated on the fly, as the message is processed, does not need to accompany the message, and can be used before the end of the message has been processed. On first consideration, this might seem impossible. However, such a scheme is known and in wide use. It is the LZW compression technique, named after Abraham Lempel, Jacob Ziv, and Terry Welch. Lempel and Ziv actually had a series of techniques, sometimes

referred to as LZ77 and LZ78, but the modification by Welch in 1984 gave the technique all the desired characteristics.

Welch wanted to reduce the number of bits sent to a recording head in disk drives, partly to increase the effective capacity of the disks, and partly to improve the speed of transmission of data. His scheme is described here, for both the encoder and decoder. It has been widely used, in many contexts, to lead to reversible compression of data. When applied to text files on a typical computer, the encoded files are typically half the size of the original ones. It is used in popular compression products such as Stuffit or Disk Doubler. When used on color images of drawings, with large areas that are the exact same color, it can lead to files that are smaller than half the size compared with file formats in which each pixel is stored. The commonly used GIF image format uses LZW compression. When used with photographs, with their gradual changes of color, the savings are much more modest.

Because this is a reversible compression technique, the original data can be reconstructed exactly, without approximation, by the decoder.

The LZW technique seems to have many advantages. However, it had one major disadvantage. It was not freely available—it was patented. The U.S. patent, No. 4,558,302, expired June 20, 2003, and patents in other countries in June, 2004, but before it did, it generated a controversy that is still an unpleasant memory for many because of the way it was handled.

3.5.1 The LZW Patent

Welch worked for Sperry Research Center at the time he developed his technique, and he published a paper[2] describing the technique. Its advantages were quickly recognized, and it was used in a variety of compression schemes, including the Graphics Interchange Format GIF developed in 1987 by CompuServe (a national Internet Service Provider) for the purpose of reducing the size of image files in their computers. Those who defined GIF did not realize that the LZW algorithm, on which GIF was based, was patented. The article by Welch did not warn that a patent was pending. The World Wide Web came into prominence during the early 1990s, and the first graphical browsers accepted GIF images. Consequently, Web site developers routinely used GIF images, thinking the technology was in the public domain, which had been CompuServe's intention.

By 1994, Unisys, the successor company to Sperry, realized the value of this patent, and decided to try to make money from it. They approached CompuServe, who didn't pay much attention at first, apparently not thinking that the threat was real. Finally, CompuServe took Unisys seriously, and the two companies together announced on December 24, 1994, that any developers writing software that creates or reads GIF images would have to license the technology from Unisys. Web site developers were not sure if their use of GIF images made them responsible for paying royalties, and they were not amused at the thought of paying for every GIF image on their sites. Soon Unisys saw that there was a public-relations disaster in the making, and they backed off on their demands. On January 27, 1995, they announced they would not try to collect for use of existing images or for images produced by tools distributed before the end of 1994, but did insist on licensing graphics tools starting in 1995. Images produced by licensed tools would be allowed on the Web without additional payment.

In 1999, Unisys decided to collect from individual Web sites that might contain images from unlicensed tools, at the rate of $5000 per site, with no exception for nonprofits, and no smaller license fees for small, low-traffic sites. It is not known how many Web sites actually paid that amount; it is reported that in the first eight months only one did. The feeling among many Web site developers was one of frustration and anger. Although Unisys avoided a public-relations disaster in 1995, they had one on their hands in 1999. There were very real cultural differences between the free-wheeling Web community, often willing to share freely, and the business community, whose standard practices were designed to help make money.

A non-infringing public-domain image-compression standard called PNG was created to replace GIF, but the browser manufacturers were slow to adopt yet another image format. Also, everybody knew that the patents would expire soon. The controversy has now gone away except in the memory of a few who felt

[2]Welch, T.A. "A Technique for High Performance Data Compression," IEEE Computer, vol. 17, no. 6, pp. 8-19; 1984.

particularly strongly. It is still cited as justification for or against changes in the patent system, or even the concept of software patents in general.

As for PNG, it offers some technical advantages (particularly better transparency features) and, as of 2004, was well supported by almost all browsers, the most significant exception being Microsoft Internet Explorer for Windows.

3.5.2 How does LZW work?

The technique, for both encoding and decoding, is illustrated with a text example in Section 3.7.

3.6 Irreversible Techniques

This section is not yet available. Sorry.

It will include as examples floating-point numbers, JPEG image compression, and MP3 audio compression. The Discrete Cosine Transformation (DCT) used in JPEG compression is discussed in Section 3.8. A demonstration of MP3 compression is available at http://www.mtl.mit.edu/Courses/6.050/2007/notes/mp3.html.

3.7 Detail: LZW Compression

The LZW compression technique is described below and applied to two examples. Both encoders and decoders are considered. The LZW compression algorithm is "reversible," meaning that it does not lose any information—the decoder can reconstruct the original message exactly.

3.7.1 LZW Algorithm, Example 1

Consider the encoding and decoding of the text message

```
itty bitty bit bin
```

(this peculiar phrase was designed to have repeated strings so that the dictionary builds up rapidly).

The initial set of dictionary entries is 8-bit character code with code points 0–255, with ASCII as the first 128 characters, including the ones in Table 3.1 which appear in the string above. Dictionary entry 256 is defined as "clear dictionary" or "start," and 257 as "end of transmission" or "stop." The encoded message is a sequence of numbers, the codes representing dictionary entries. Initially most dictionary entries consist of a single character, but as the message is analyzed new entries are defined that stand for strings of two or more characters. The result is summarized in Table 3.2.

32	space	116	t
98	b	121	y
105	i	256	start
110	n	257	stop

Table 3.1: LZW Example 1 Starting Dictionary

Encoding algorithm: Define a place to keep new dictionary entries while they are being constructed and call it **new-entry**. Start with new-entry empty, and send the start code. Then append to the new-entry the characters, one by one, from the string being compressed. As soon as new-entry fails to match any existing dictionary entry, put new-entry into the dictionary, using the next available code point, and send the code for the string without the last character (this entry is already in the dictionary). Then use the last character received as the first character of the next new-entry. When the input string ends, send the code for whatever is in new-entry followed by the stop code. That's all there is to it.

For the benefit of those who appreciate seeing algorithms written like a computer program, this encoding algorithm is shown in Figure 3.5. When this procedure is applied to the string in question, the first character

```
 0  # encoding algorithm
 1  clear dictionary
 2  send start code
 3  for each character {
 4    if new-entry appended with character is not in dictionary {
 5      send code for new-entry
 6      add new-entry appended with character as new dictionary entry
 7      set new-entry blank
 8    }
 9    append character to new-entry
10  }
11  send code for new-entry
12  send stop code
```

Figure 3.5: LZW encoding algorithm

Encoding				Transmission		Decoding		
Input		New dictionary entry		9-bit characters transmitted		New dictionary entry		Output
105	i	–	–	256	(start)	–	–	–
116	t	258	i t	105	i	–	–	i
116	t	259	t t	116	t	258	i t	t
121	y	260	t y	116	t	259	t t	t
32	space	261	y space	121	y	260	t y	y
98	b	262	space b	32	space	261	y space	space
105	i	263	b i	98	b	262	space b	b
116	t	–	–	–	–	–	–	–
116	t	264	i t t	258	i t	263	b i	i t
121	y	–	–	–	–	–	–	–
32	space	265	t y space	260	t y	264	i t t	t y
98	b		–	–	–	–	–	–
105	i	266	space-b i	262	space b	265	t y space	space b
116	t	–	–	–		–	–	–
32	space	267	i t space	258	i t	266	space b i	i t
98	b	–	–	–		–	–	–
105	i	–	–	–		–	–	–
110	n	268	space b i n	266	space b i	267	i t space	space b i
–	–	–	–	110	n	268	space b i n	n
–	–	–	–	257	(stop)	–	–	–

8-bit characters input

Table 3.2: LZW Example 1 Transmission Summary

is “i” and the string consisting of just that character is already in the dictionary. So the next character is appended to new-entry, and the result is “it” which is not in the dictionary. Therefore the string which was in the dictionary, “i,” is sent and the string “i t” is added to the dictionary, at the next available position, which is 258. The new-entry is reset to be just the last character, which was not sent, so it is “t”. The next character “t” is appended and the result is “tt” which is not in the dictionary. The process repeats until the end of the string is reached.

For a while at the beginning the additional dictionary entries are all two-character strings, and there is a string transmitted for every new character encountered. However, the first time one of those two-character strings is repeated, its code gets sent (using fewer bits than would be required for two characters sent separately) and a new three-character dictionary entry is defined. In this example it happens with the string “i t t” (this message was designed to make this happen earlier than would be expected with normal text). Later in this example, the code for a three-character string gets transmitted, and a four-character dictionary entry defined.

In this example the codes are sent to a receiver which is expected to decode the message and produce as output the original string. The receiver does not have access to the encoder’s dictionary and therefore the decoder must build up its own copy.

Decoding algorithm: If the start code is received, clear the dictionary and set new-entry empty. For the next received code, output the character represented by the code and also place it in new-entry. Then for subsequent codes received, append the first character of the string represented by the code to new-entry, insert the result in the dictionary, then output the string for the received code and also place it in new-entry to start the next dictionary entry. When the stop code is received, nothing needs to be done; new-entry can be abandoned.

This algorithm is shown in program format in Figure 3.6.

```
 0  # decoding algorithm
 1  for each received code until stop code {
 2    if code is start code {
 3      clear dictionary
 4      get next code
 5      get string from dictionary  # it will be a single character
 6    }
 7    else {
 8      get string from dictionary
 9      update last dictionary entry by appending first character of string
10    }
11    add string as new dictionary entry
12    output string
13  }
```

Figure 3.6: LZW decoding algorithm

Note that the coder and decoder each create the dictionary on the fly; the dictionary therefore does not have to be explicitly transmitted, and the coder deals with the text in a single pass.

Does this work, i.e., is the number of bits needed for transmission reduced? We sent 18 8-bit characters (144 bits) in 14 9-bit transmissions (126 bits), a savings of 12.5%, for this very short example. For typical text there is not much reduction for strings under 500 bytes. Larger text files are often compressed by a factor of 2, and drawings even more.

3.7.2 LZW Algorithm, Example 2

Encode and decode the text message

`itty bitty nitty grrritty bit bin`

(again, this peculiar phrase was designed to have repeated strings so that the dictionary forms rapidly; it also has a three-long sequence `rrr` which illustrates one aspect of this algorithm).

The initial set of dictionary entries include the characters in Table 3.3, which are found in the string, along with control characters for start and stop.

32	space	114	r
98	b	116	t
103	g	121	y
105	i	256	start
110	n	257	stop

Table 3.3: LZW Example 2 Starting Dictionary

The same algorithms used in Example 1 can be applied here. The result is shown in Table 3.4. Note that the dictionary builds up quite rapidly, and there is one instance of a four-character dictionary entry transmitted. Was this compression effective? Definitely. A total of 33 8-bit characters (264 bits) were sent in 22 9-bit transmissions (198 bits, even including the start and stop characters), for a saving of 25% in bits.

There is one place in this example where the decoder needs to do something unusual. Normally, on receipt of a transmitted codeword, the decoder can look up its string in the dictionary and then output it and use its first character to complete the partially formed last dictionary entry, and then start the next dictionary entry. Thus only one dictionary lookup is needed. However, the algorithm presented above uses

two lookups, one for the first character, and a later one for the entire string. Why not use just one lookup for greater efficiency?

There is a special case illustrated by the transmission of code 271 in this example, where the string corresponding to the received code is not complete. The first character can be found but then before the entire string is retrieved, the entry must be completed. This happens when a character or a string appears for the first time three times in a row, and is therefore rare. The algorithm above works correctly, at a cost of an extra lookup that is seldom needed and may slow the algorithm down. A faster algorithm with a single dictionary lookup works reliably only if it detects this situation and treats it as a special case.

Encoding				Transmission		Decoding		
Input		New dictionary entry				New dictionary entry		Output
105	i	–	–	256	(start)	–	–	–
116	t	258	i t	105	i	–	–	i
116	t	259	t t	116	t	258	i t	t
121	y	260	t y	116	t	259	t t	t
32	space	261	y space	121	y	260	t y	y
98	b	262	space b	32	space	261	y space	space
105	i	263	b i	98	b	262	space b	b
116	t	–	–	–	–	–	–	–
116	t	264	i t t	258	i t	263	b i	i t
121	y	–	–	–	–	–	–	–
32	space	265	t y space	260	t y	264	i t t	t y
110	n	266	space n	32	space	265	t y space	space
105	i	267	n i	110	n	266	space n	n
116	t	–	–	–	–	–	–	–
116	t	–	–	–	–	–	–	–
121	y	268	i t t y	264	i t t	267	n i	i t t
32	space	–	–	–	–	–	–	–
103	g	269	y space g	261	y space	268	i t t y	y space
114	r	270	g r	103	g	269	y space g	g
114	r	271	r r	114	r	270	g r	r
114	r	–	–	–	–	–	–	–
105	i	272	r r i	271	r r	271	r r	r r
116	t	–	–	–	–	–	–	–
116	t	–	–	–	–	–	–	–
121	y	–	–	–	–	–	–	–
32	space	273	i t t y space	268	i t t y	272	r r i	i t t y
98	b	–	–	–	–	–	–	–
105	i	274	space b i	262	space b	273	i t t y space	space b
116	t	–	–	–	–	–	–	–
32	space	275	i t space	258	i t	274	space b i	it
98	b	–	–	–	–	–	–	–
105	i	–	–	–	–	–	–	–
110	n	276	space b i n	274	space b i	275	i t space	space b i
–	–	–	–	110	n	276	space b i n	n
–	–	–	–	257	(stop)	–	–	–

Table 3.4: LZW Example 2 Transmission Summary

3.8 Detail: 2-D Discrete Cosine Transformation

This section is based on notes written by Luis Pérez-Breva, Feb 3, 2005, and notes from Joseph C. Huang, Feb 25, 2000.

The Discrete Cosine Transformation (DCT) is an integral part of the JPEG (Joint Photographic Experts Group) compression algorithm. DCT is used to convert the information in an array of picture elements (pixels) into a form in which the information that is most relevant for human perception can be identified and retained, and the information less relevant may be discarded.

DCT is one of many discrete linear transformations that might be considered for image compression. It has the advantage that fast algorithms (related to the FFT, Fast Fourier Transform) are available.

Several mathematical notations are possible to describe DCT. The most succinct is that using vectors and matrices. A vector is a one-dimensional array of numbers (or other things). It can be denoted as a single character or between large square brackets with the individual elements shown in a vertical column (a row representation, with the elements arranged horizontally, is also possible, but is usually regarded as the transpose of the vector). In these notes we will use bold-face letters ($\mathbf{V}$) for vectors. A matrix is a two-dimensional array of numbers (or other things) which again may be represented by a single character or in an array between large square brackets. In these notes we will use a font called "blackboard bold" ($\mathbb{M}$) for matrices. When it is necessary to indicate particular elements of a vector or matrix, a symbol with one or two subscripts is used. In the case of a vector, the single subscript is an integer in the range from 0 through $n-1$ where n is the number of elements in the vector. In the case of a matrix, the first subscript denotes the row and the second the column, each an integer in the range from 0 through $n-1$ where n is either the number of rows or the number of columns, which are the same only if the matrix is square.

3.8.1 Discrete Linear Transformations

In general, a discrete linear transformation takes a vector as input and returns another vector of the same size. The elements of the output vector are a linear combination of the elements of the input vector, and therefore this transformation can be carried out by matrix multiplication.

For example, consider the following matrix multiplication:

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 5 \\ 3 \end{bmatrix}. \tag{3.1}$$

If the vectors and matrices in this equation are named

$$\mathbb{C} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \mathbf{I} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}, \quad \mathbf{O} = \begin{bmatrix} 5 \\ 3 \end{bmatrix},$$

then Equation 3.1 becomes

$$\mathbf{O} = \mathbb{C}\mathbf{I}. \tag{3.2}$$

We can now think of $\mathbb{C}$ as a discrete linear transformation that transforms the input vector $\mathbf{I}$ into the output vector $\mathbf{O}$. Incidentally, this particular transformation $\mathbb{C}$ is one that transforms the input vector $\mathbf{I}$ into a vector that contains the *sum* (5) and the *difference* (3) of its components.[3]

The procedure is the same for a 3×3 matrix acting on a 3 element vector:

$$\begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} \longrightarrow \begin{bmatrix} o_1 & = \sum_j c_{1,j} i_j \\ o_2 & = \sum_j c_{2,j} i_j \\ o_3 & = \sum_j c_{3,j} i_j \end{bmatrix} \tag{3.3}$$

which again can be written in the succinct form

$$\mathbf{O} = \mathbb{C}\mathbf{I}. \tag{3.4}$$

[3] It happens that $\mathbb{C}$ is $\sqrt{2}$ times the 2×2 Discrete Cosine Transformation matrix defined in Section 3.8.2.

where now the vectors are of size 3 and the matrix is 3×3.

In general, for a transformation of this form, if the matrix $\mathbb{C}$ has an inverse $\mathbb{C}^{-1}$ then the vector $\mathbf{I}$ can be reconstructed from its transform by

$$\mathbf{I} = \mathbb{C}^{-1}\mathbf{O}. \tag{3.5}$$

Equations 3.3 and 3.1 illustrate the linear transformation when the input is a column vector. The procedure for a row-vector is similar, but the order of the vector and matrix is reversed, and the transformation matrix is transposed.[4] This change is consistent with viewing a row vector as the transpose of the column vector. For example:

$$\begin{bmatrix}4 & 1\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \longrightarrow \begin{bmatrix}5 & 3\end{bmatrix}. \tag{3.6}$$

Vectors are useful for dealing with objects that have a one-dimensional character, such as a sound waveform sampled a finite number of times. Images are inherently two-dimensional in nature, and it is natural to use matrices to represent the properties of the pixels of an image. Video is inherently three-dimensional (two space and one time) and it is natural to use three-dimensional arrays of numbers to represent their data. The succinct vector-matrix notation given here extends gracefully to two-dimensional systems, but not to higher dimensions (other mathematical notations can be used).

Extending linear transformations to act on matrices, not just vectors, is not difficult. For example, consider a very small image of six pixels, three rows of two pixels each, or two columns of three pixels each. A number representing some property of each pixel (such as its brightness on a scale of 0 to 1) could form a 3×2 matrix:

$$\begin{bmatrix}i_{1,1} & i_{1,2}\\ i_{2,1} & i_{2,2}\\ i_{3,1} & i_{3,2}\end{bmatrix} \tag{3.7}$$

The most general linear transformation that leads to a 3×2 output matrix would require 36 coefficients. When the arrangement of the elements in a matrix reflects the underlying object being represented, a less general set of linear transformations, that operate on the rows and columns separately, using different matrices $\mathbb{C}$ and $\mathbb{D}$, may be useful:

$$\begin{bmatrix}c_{1,1} & c_{1,2} & c_{1,3}\\ c_{2,1} & c_{2,2} & c_{2,3}\\ c_{3,1} & c_{3,2} & c_{3,3}\end{bmatrix}\begin{bmatrix}i_{1,1} & i_{1,2}\\ i_{2,1} & i_{2,2}\\ i_{3,1} & i_{3,2}\end{bmatrix}\begin{bmatrix}d_{1,1} & d_{1,2}\\ d_{2,1} & d_{2,2}\end{bmatrix} \longrightarrow \begin{bmatrix}o_{1,1} & o_{1,2}\\ o_{2,1} & o_{2,2}\\ o_{3,1} & o_{3,2}\end{bmatrix} \tag{3.8}$$

or, in matrix notation,

$$\mathbb{O} = \mathbb{C}\mathbb{I}\mathbb{D}, \tag{3.9}$$

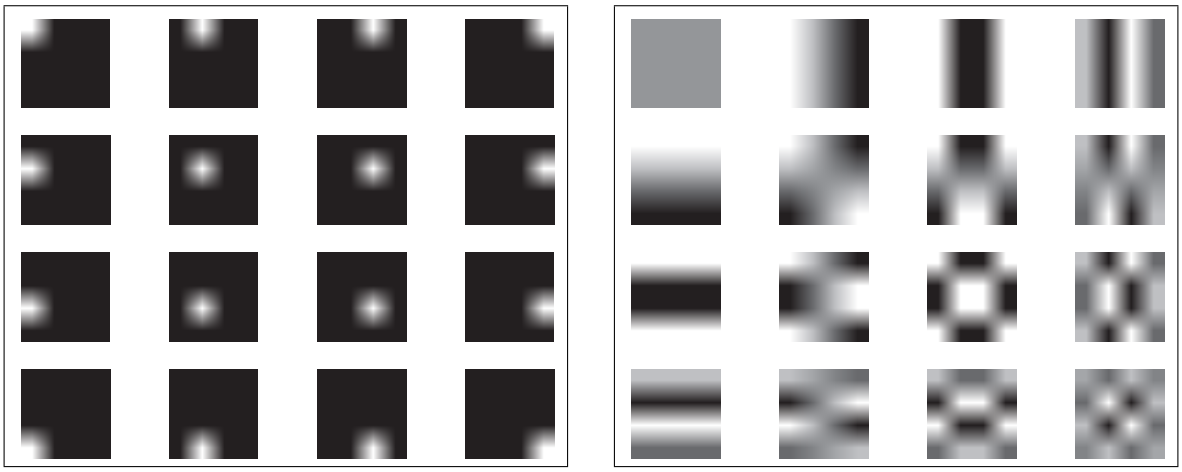
Note that the matrices at left $\mathbb{C}$ and right $\mathbb{D}$ in this case are generally of different size, and may or may not be of the same general character. (An important special case is when $\mathbb{I}$ is square, i.e., it contains the same number of rows and columns. In this case the output matrix $\mathbb{O}$ is also square, and $\mathbb{C}$ and $\mathbb{D}$ are the same size.)

3.8.2 Discrete Cosine Transformation

In the language of linear algebra, the formula

$$\mathbb{Y} = \mathbb{C}\mathbb{X}\mathbb{D} \tag{3.10}$$

[4] Transposing a matrix means flipping its elements about the main diagonal, so the i, j element of the transpose is the j, i element of the original matrix. Transposed matrices are denoted with a superscript T, as in $\mathbb{C}^T$. In general the transpose of a product of two matrices (or vectors) is the product of the two transposed matrices or vectors, in reverse order: $(\mathbb{A}\mathbb{B})^T = \mathbb{B}^T\mathbb{A}^T$.



(a) Transformed Matrices $\mathbb{Y}$ (b) Basis Matrices $\mathbb{X}$

Figure 3.7: (a) 4×4 pixel images representing the coefficients appearing in the matrix $\mathbb{Y}$ from equation 3.15. And, (b) corresponding Inverse Discrete Cosine Transformations, these ICDTs can be interpreted as a the base images that correspond to the coefficients of $\mathbb{Y}$.

represents a transformation of the matrix $\mathbb{X}$ into a matrix of coefficients $\mathbb{Y}$. Assuming that the transformation matrices $\mathbb{C}$ and $\mathbb{D}$ have inverses $\mathbb{C}^{-1}$ and $\mathbb{D}^{-1}$ respectively, the original matrix can be reconstructed from the coefficients by the reverse transformation:

$$\mathbb{X} = \mathbb{C}^{-1}\mathbb{Y}\mathbb{D}^{-1}. \tag{3.11}$$

This interpretation of $\mathbb{Y}$ as coefficients useful for the reconstruction of $\mathbb{X}$ is especially useful for the Discrete Cosine Transformation.

The Discrete Cosine Transformation is a Discrete Linear Transformation of the type discussed above

$$\mathbb{Y} = \mathbb{C}^{T}\mathbb{X}\mathbb{C}, \tag{3.12}$$

where the matrices are all of size $N \times N$ and the two transformation matrices are transposes of each other. The transformation is called the Cosine transformation because the matrix $\mathbb{C}$ is defined as

$$\{\mathbb{C}\}_{m,n} = k_n \cos\left[\frac{(2m+1)n\pi}{2N}\right] \text{ where } k_n = \begin{cases} \sqrt{1/N} & \text{if } n = 0 \\ \sqrt{2/N} & \text{otherwise} \end{cases} \tag{3.13}$$

where $m, n = 0, 1, \ldots (N-1)$. This matrix $\mathbb{C}$ has an inverse which is equal to its transpose:

$$\mathbb{C}^{-1} = \mathbb{C}^{T}. \tag{3.14}$$

Using Equation 3.12 with $\mathbb{C}$ as defined in Equation 3.13, we can compute the DCT $\mathbb{Y}$ of any matrix $\mathbb{X}$, where the matrix $\mathbb{X}$ may represent the pixels of a given image. In the context of the DCT, the inverse procedure outlined in equation 3.11 is called the Inverse Discrete Cosine Transformation (IDCT):

$$\mathbb{X} = \mathbb{C}\mathbb{Y}\mathbb{C}^{T}. \tag{3.15}$$

With this equation, we can compute the set of base matrices of the DCT, that is: *the set of matrices to which each of the elements of* $\mathbb{Y}$ *corresponds via the DCT*. Let us construct the set of all possible images with a single non-zero pixel each. These images will represent the individual coefficients of the matrix $\mathbb{Y}$.

Figure 3.7(a) shows the set for 4×4 pixel images. Figure 3.7(b) shows the result of applying the IDCT to the images in Figure 3.7(a). The set of images in Figure 3.7(b) are called basis because the DCT of any of them will yield a matrix $\mathbb{Y}$ that has a single non-zero coefficient, and thus they represent the base images in which the DCT "decomposes" any input image.

Recalling our overview of Discrete Linear Transformations above, should we want to recover an image $\mathbb{X}$ from its DCT $\mathbb{Y}$ we would just take each element of $\mathbb{Y}$ and multiply it by the corresponding matrix from 3.7(b). Indeed, Figure 3.7(b) introduces a very remarkable property of the DCT basis: it encodes spatial frequency. Compression can be achieved by ignoring those spatial frequencies that have smaller DCT coefficients. Think about the image of a chessboard—it has a high spatial frequency component, and almost all of the low frequency components can be removed. Conversely, blurred images tend to have fewer higher spatial frequency components, and then high frequency components, lower right in the Figure 3.7(b), can be set to zero as an "acceptable approximation". This is the principle for irreversible compression behind JPEG.

Figure 3.8 shows MATLAB code to generate the basis images as shown above for the 4×4, 8×8, and 16×16 DCT.

```
% N is the size of the NxN image being DCTed.
% This code, will consider 4x4, 8x8, and 16x16 images
% and will construct the basis of DCT transforms
for N = [4 8 16];
    % Create The transformation Matrix C
    C = zeros(N,N);
    mrange=0:(N-1);           %m will indicate rows
    nrange=mrange;            %n will indicate columns
    k=ones(N,N)*sqrt(2/N);    %create normalization matrix
    k(:,1)=sqrt(1/N);         %note different normalization for first column
    C=k.*cos((2*mrange+1)'*nrange*pi/(2*N));    %construct the transform matrix
    %note that we are interested in the Inverse Discrete Cosine Transformation
    %so we must invert (i.e. transpose) C
    C=C';
    % Get Basis Matrices
    figure;colormap('gray');  %open the figure and set the color to grayscale
    for m = mrange
        for n = nrange
            %create a matrix that just has one pixel
            Y=zeros(N,N);
            Y(n+1,m+1)=1;

            X = C'*Y*C;       % X is the transformed matrix.
            subplot(N,N,m*N+n+1);imagesc(X); %plot X
            axis square;      %arrange scale of axis
            axis off;         %hide axis
        end
    end
end
```

Figure 3.8: Base matrix generator

Chapter 4

Errors

In Chapter 2 we saw examples of how symbols could be represented by arrays of bits. In Chapter 3 we looked at some techniques of compressing the bit representations of such symbols, or a series of such symbols, so fewer bits would be required to represent them. If this is done while preserving all the original information, the compressions are said to be lossless, or reversible, but if done while losing (presumably unimportant) information, the compression is called lossy, or irreversible. Frequently source coding and compression are combined into one operation.

Because of compression, there are fewer bits carrying the same information, so each bit is more important, and the consequence of an error in a single bit is more serious. All practical systems introduce errors to information that they process (some systems more than others, of course). In this chapter we examine techniques for insuring that such inevitable errors cause no harm.

4.1 Extension of System Model

Our model for information handling will be extended to include "channel coding." The new channel encoder adds bits to the message so that in case it gets corrupted in some way, the channel encoder will know that and possibly even be able to repair the damage.

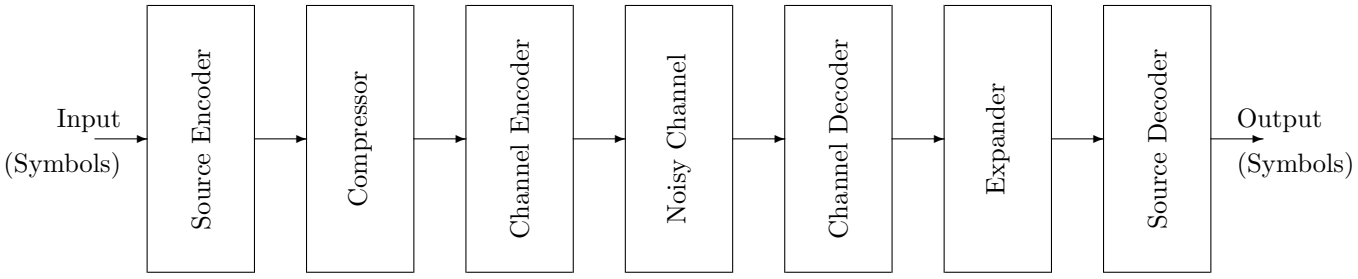


Figure 4.1: Communication system with errors

4.2 How do Errors Happen?

The model pictured above is quite general, in that the purpose might be to transmit information from one place to another (communication), store it for use later (data storage), or even process it so that the output is not intended to be a faithful replica of the input (computation). Different systems involve different physical devices as the channel (for example a communication link, a floppy disk, or a computer). Many physical effects can cause errors. A CD or DVD can get scratched. A memory cell can fail. A telephone line can be noisy. A computer gate can respond to an unwanted surge in power supply voltage.

For our purposes we will model all such errors as a change in one or more bits from 1 to 0 or vice versa. In the usual case where a message consists of several bits, we will usually assume that different bits get corrupted independently, but in some cases errors in adjacent bits are not independent of each other, but instead have a common underlying cause (i.e., the errors may happen in bursts).

4.3 Detection vs. Correction

There are two approaches to dealing with errors. One is to detect the error and then let the person or system that uses the output know that an error has occurred. The other is to have the channel decoder attempt to repair the message by correcting the error. In both cases, extra bits are added to the messages to make them longer. The result is that the message contains redundancy—if it did not, every possible bit pattern would be a legal message and an error would simply change one valid message to another. By changing things so that many (indeed, most) bit patterns do not correspond to legal messages, the effect of an error is usually to change the message to one of the illegal patterns; the channel decoder can detect that there was an error and take suitable action. In fact, if every illegal pattern is, in a sense to be described below, closer to one legal message than any other, the decoder could substitute the closest legal message, thereby repairing the damage.

In everyday life error detection and correction occur routinely. Written and spoken communication is done with natural languages such as English, and there is sufficient redundancy (estimated at 50%) so that even if several letters, sounds, or even words are omitted, humans can still understand the message.

Note that channel encoders, because they add bits to the pattern, generally preserve all the original information, and therefore are reversible. The channel, by allowing errors to occur, actually introduces information (the details of exactly which bits got changed). The decoder is irreversible in that it discards some information, but if well designed it throws out the "bad" information caused by the errors, and keeps the original information. In later chapters we will analyze the information flow in such systems quantitatively.

4.4 Hamming Distance

We need some technique for saying how similar two bit patterns are. In the case of physical quantities such as length, it is quite natural to think of two measurements as being close, or approximately equal. Is there a similar sense in which two patterns of bits are close?

At first it is tempting to say that two bit patterns are close if they represent integers that are adjacent, or floating point numbers that are close. However, this notion is not useful because it is based on particular meanings ascribed to the bit patterns. It is not obvious that two bit patterns which differ in the first bit should be any more or less "different" from each other than two which differ in the last bit.

A more useful definition of the difference between two bit patterns is the number of bits that are different between the two. This is called the Hamming distance, after Richard W. Hamming (1915 – 1998)[1]. Thus 0110 and 1110 are separated by Hamming distance of one. Two patterns which are the same are separated by Hamming distance of zero.

[1]See a biography of Hamming at http://www-groups.dcs.st-andrews.ac.uk/%7Ehistory/Biographies/Hamming.html

Note that a Hamming distance can only be defined between two bit patterns with the same number of bits. It does not make sense to speak of the Hamming distance of a single string of bits, or between two bit strings of different lengths.

Using this definition, the effect of errors introduced in the channel can be described by the Hamming distance between the two bit patterns, one at the input to the channel and the other at the output. No errors means Hamming distance zero, and a single error means Hamming distance one. If two errors occur, this generally means a Hamming distance of two. (Note, however, that if the two errors happen to the same bit, the second would cancel the first, and the Hamming distance would actually be zero.)

The action of an encoder can also be appreciated in terms of Hamming distance. In order to provide error detection, it is necessary that the encoder produce bit patterns so that any two different inputs are separated in the output by Hamming distance at least two—otherwise a single error could convert one legal codeword into another. In order to provide double-error protection the separation of any two valid codewords must be at least three. In order for single-error correction to be possible, all valid codewords must be separated by Hamming distance at least three.

4.5 Single Bits

Transmission of a single bit may not seem important, but it does bring up some commonly used techniques for error detection and correction.

The way to protect a single bit is to send it more than once, and expect that more often than not each bit sent will be unchanged. The simplest case is to send it twice. Thus the message 0 is replaced by 00 and 1 by 11 by the channel encoder. The decoder can then raise an alarm if the two bits are different (that can only happen because of an error). But there is a subtle point. What if there are two errors? If the two errors both happen on the same bit, then that bit gets restored to its original value and it is as though no error happened. But if the two errors happen on different bits then they end up the same, although wrong, and the error is undetected. If there are more errors, then the possibility of undetected changes becomes substantial (an odd number of errors would be detected but an even number would not).

If multiple errors are likely, greater redundancy can help. Thus, to detect double errors, you can send the single bit three times. Unless all three are the same when received by the channel decoder, it is known that an error has occurred, but it is not known how many errors there might have been. And of course triple errors may go undetected.

Now what can be done to allow the decoder to correct an error, not just detect one? If there is known to be at most one error, and if a single bit is sent three times, then the channel decoder can tell whether an error has occurred (if the three bits are not all the same) and it can also tell what the original value was—the process used is sometimes called “majority logic” (choosing whichever bit occurs most often). This technique, called ”triple redundancy” can be used to protect communication channels, memory, or arbitrary computation.

Note that triple redundancy can be used either to correct single errors or to detect double errors, but not both. If you need both, you can use quadruple redundancy—send four identical copies of the bit.

Two important issues are how efficient and how effective these techniques are. As for efficiency, it is convenient to define the **code rate** as the number of bits before channel coding divided by the number after the encoder. Thus the code rate lies between 0 and 1. Double redundancy leads to a code rate of 0.5, and triple redundancy 0.33. As for effectiveness, if errors are very unlikely it may be reasonable to ignore the even more unlikely case of two errors so close together. If so, triple redundancy is very effective. On the other hand, some physical sources of errors may wipe out data in large bursts (think of a physical scratch on a CD) in which case one error, even if unlikely, is apt to be accompanied by a similar error on adjacent bits, so triple redundancy will not be effective.

Figures 4.2 and 4.3 illustrate how triple redundancy protects single errors but can fail if there are two errors.

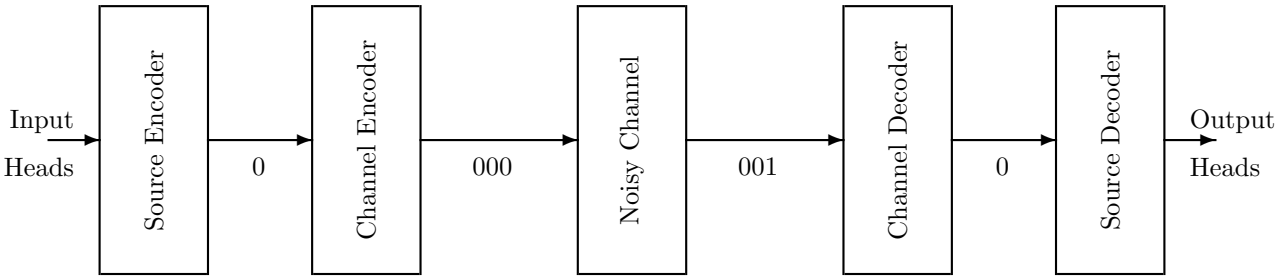


Figure 4.2: Triple redundancy channel encoding and single-error correction decoding, for a channel introducing one bit error.

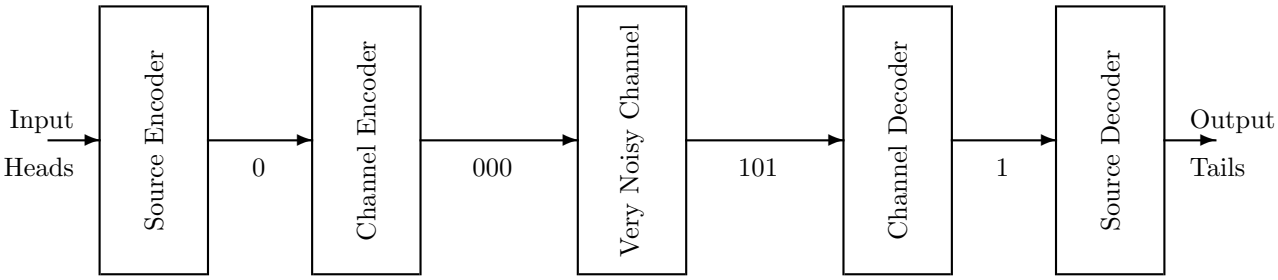


Figure 4.3: Triple redundancy channel encoding and single-error correction decoding, for a channel introducing two bit errors.

4.6 Multiple Bits

To detect errors in a sequence of bits several techniques can be used. Some can perform error correction as well as detection.

4.6.1 Parity

Consider a byte, which is 8 bits. To enable detection of single errors, a "parity" bit (also called a "check bit") can be added, changing the 8-bit string into 9 bits. The added bit would be 1 if the number of bits equal to 1 is odd, and 0 otherwise. Thus the string of 9 bits would always have an even number of bits equal to 1. Then the decoder would simply count the number of 1 bits and if it is odd, know there is an error (or, more generally, an odd number of errors). The decoder could not repair the damage, and indeed could not even tell if the damage might by chance have occurred in the parity bit, in which case the data bits would still be correct. If would also not detect double errors (or more generally an even number of errors). The use of parity bits is efficient, since the code rate is 8/9, but of limited effectiveness. It cannot deal with the case where the channel represents computation and therefore the output is not intended to be the same as the input. It is most often used when the likelihood of an error is very small, and there is no reason to suppose that errors of adjacent bits occur together, and the receiver is able to request a retransmission of the data.

Sometimes parity is used even when retransmission is not possible. On early IBM Personal Computers, memory references were protected by single-bit parity. When an error was detected (very infrequently), the computer crashed.

Error correction is more useful than error detection, but requires more bits and is therefore less efficient. Two of the more common methods are discussed next.

If the data being transmitted is more conveniently thought of as sequences of objects which themselves may be coded in multiple bits, such as letters or digits, the advantages of parity bits can be achieved by adding similar objects rather than parity bits. For example, **check digits** can be added to an array of digits. Section 4.9 discusses some common uses of check digits.

4.6.2 Rectangular Codes

Rectangular codes can provide single error correction and double error detection simultaneously. Suppose we wish to protect a byte of information, the eight data bits D0 D1 D2 D3 D4 D5 D6 D7. Let us arrange these in a rectangular table and add parity bits for each of the two rows and four columns:

D0	D1	D2	D3	PR0
D4	D5	D6	D7	PR1
PC0	PC1	PC2	PC3	P

Table 4.1: Parity Bits

The idea is that each of the parity bits PR0 PR1 PC0 PC1 PC2 PC3 is set so that the overall parity of the particular row or column is even. The total parity bit P is then set so that the right-hand column consisting only of parity bits has itself even parity—this guarantees that the bottom row also has even parity. The 15 bits can be sent through the channel and the decoder analyzes the received bits. It performs a total of 8 parity checks, on the three rows and the five columns. If there is a single error in any one of the bits, then one of the three row parities and one of the five column parities will be wrong. The offending bit can thereby be identified (it lies at the intersection of the row and column with incorrect parity) and changed. If there are two errors, there will be a different pattern of parity failures; double errors can be detected but not corrected. Triple errors can be nasty in that they can mimic a single error of an innocent bit.

Other geometrically inspired codes can be devised, based on arranging the bits in triangles, cubes, pyramids, wedges, or higher-dimensional structures.

4.6.3 Hamming Codes

Suppose we wish to correct single errors, and are willing to ignore the possibility of multiple errors. A set of codes was invented by Richard Hamming with the minimum number of extra parity bits.

Each extra bit added by the channel encoder allows one check of a parity by the decoder and therefore one bit of information that can be used to help identify the location of the error. For example, if three extra bits are used, the three tests could identify up to eight error conditions. One of these would be “no error” so seven would be left to identify the location of up to seven places in the pattern with an error. Thus the data block could be seven bits long. Three of these bits would be added for error checking, leaving four for the payload data. Similarly, if there were four parity bits, the block could be 15 bits long leaving 11 bits for payload.

Codes that have as large a payload as possible for a given number of parity bits are sometimes called “perfect.” It is possible, of course, to have a smaller payload, in which case the resulting Hamming codes would have a lower code rate. For example, since much data processing is focused on bytes, each eight bits long, a convenient Hamming code would be one using four parity bits to protect eight data bits. Thus two bytes of data could be processed, along with the parity bits, in exactly three bytes.

Table 4.2 lists some Hamming codes. The trivial case of 1 parity bit is not shown because there is no room for any data. The first entry is a simple one, and we have seen it already. It is triple redundancy, where a block of three bits is sent for a single data bit. As we saw earlier, this scheme is capable of single-error correction or double-error detection, but not both (this is true of all the Hamming Codes). The second entry is one of considerable interest, since it is the simplest Hamming Code with reasonable efficiency.

Let’s design a (7, 4, 3) Hamming code. There are several ways to do this, but it is probably easiest to start with the decoder. The decoder receives seven bits and performs three parity checks on groups of those

Parity bits	Block size	Payload	Code rate	Block code type
2	3	1	0.33	(3, 1, 3)
3	7	4	0.57	(7, 4, 3)
4	15	11	0.73	(15, 11, 3)
5	31	26	0.84	(31, 26, 3)
6	63	57	0.90	(63, 57, 3)
7	127	120	0.94	(127, 120, 3)
8	255	247	0.97	(255, 247, 3)

Table 4.2: Perfect Hamming Codes

bits with the intent of identifying where an error has occurred, if it has. Let's label the bits 1 through 7. If the results are all even parity, the decoder concludes that no error has occurred. Otherwise, the identity of the changed bit is deduced by knowing which parity operations failed. Of the perfect Hamming codes with three parity bits, one is particularly elegant:

- The first parity check uses bits 4, 5, 6, or 7 and therefore fails if one of them is changed
- The second parity check uses bits 2, 3, 6, or 7 and therefore fails if one of them is changed
- The third parity check uses bits 1, 3, 5, or 7 and therefore fails if one of them is changed

These rules are easy to remember. The three parity checks are part of the binary representation of the location of the faulty bit—for example, the integer 6 has binary representation 1 1 0 which corresponds to the first and second parity checks failing but not the third.

Now consider the encoder. Of these seven bits, four are the original data and three are added by the encoder. If the original data bits are 3 5 6 7 it is easy for the encoder to calculate bits 1 2 4 from knowing the rules given just above—for example, bit 2 is set to whatever is necessary to make the parity of bits 2 3 6 7 even which means 0 if the parity of bits 3 6 7 is already even and 1 otherwise. The encoder calculates the parity bits and arranges all the bits in the desired order. Then the decoder, after correcting a bit if necessary, can extract the data bits and discard the parity bits, which have done their job and are no longer needed.

4.7 Block Codes

It is convenient to think in terms of providing error-correcting protection to a certain amount of data and then sending the result in a block of length n. If the number of data bits in this block is k, then the number of parity bits is $n-k$, and it is customary to call such a code an (n, k) block code. Thus the Hamming Code just described is (7, 4).

It is also customary (and we shall do so in these notes) to include in the parentheses the minimum Hamming distance d between any two valid codewords, or original data items, in the form (n, k, d). The Hamming Code that we just described can then be categorized as a (7, 4, 3) block code.

4.8 Advanced Codes

Block codes with minimum Hamming distance greater than 3 are possible. They can handle more than single errors. Some are known as Bose-Chaudhuri-Hocquenhem (BCH) codes. Of great commercial interest today are a class of codes announced in 1960 by Irving S. Reed and Gustave Solomon of MIT Lincoln Laboratory. These codes deal with bytes of data rather than bits. The (256, 224, 5) and (224, 192, 5) Reed-Solomon codes are used in CD players and can, together, protect against long error bursts.

More advanced channel codes make use of past blocks of data as well as the present block. Both the encoder and decoder for such codes need local memory but not necessarily very much. The data processing

for such advanced codes can be very challenging. It is not easy to develop a code that is efficient, protects against large numbers of errors, is easy to program, and executes rapidly. One important class of codes is known as convolutional codes, of which an important sub-class is trellis codes which are commonly used in data modems.

4.9 Detail: Check Digits

Error detection is routinely used to reduce human error. Many times people must deal with long serial numbers or character sequences that are read out loud or typed at a keyboard. Examples include credit-card numbers, social security numbers, and software registration codes. These actions are prone to error. Extra digits or characters can be included to detect errors, just as parity bits are included in bit strings. Often this is sufficient because when an error is detected the operation can be repeated conveniently.

In other cases, such as telephone numbers or e-mail addresses, no check characters are used, and therefore any sequence may be valid. Obviously more care must be exercised in using these, to avoid dialing a wrong number or sending an e-mail message or fax to the wrong person.

Credit Cards

Credit card numbers have an extra **check digit** calculated in a way specified in 1954 by H. P. Luhn of IBM. It is designed to guard against a common type of error, which is transposition of two adjacent digits.

Credit card numbers typically contain 15 or 16 digits (Luhn's algorithm actually works for any number of digits). The first six digits denote the organization that issued the card. The financial industry discourages public disclosure of these codes, though most are already widely known, certainly to those seriously considering fraud. Of those six digits, the first denotes the economic sector associated with the card, for example 1 and 2 for airlines, 3 for travel and entertainment, and 4, 5, and 6 for banks and stores. The last digit is the check digit, and the other digits denote the individual card account.

Credit card issuers have been assigned their own prefixes in accordance with this scheme. For example, American Express cards have numbers starting with either 34 or 38, Visa with 4, MasterCard with 51, 52, 53, 54, or 55, and Discover with 6011 or 65.

The Luhn procedure tests whether a credit card number, including the check digit, is valid. First, select those digits from the card number that appear in alternate positions, starting with the next-to-last digit. For example, if the card number is 1234 4567 7891, those digits would be 9, 7, 6, 4, 3, and 1. Note how many of those digits are greater than 4 (in this case 3 of them). Then add those digits together (for the example, $9+7+6+4+3+1=30$). Then add all the digits in the card number (in this example, 57). Look at the sum of those three numbers (in this case $3+30+57=90$). If the result is a multiple of 10, as in this example, the card number passes the test and may be valid. Otherwise, it is not.

This procedure detects all single-digit errors, and almost all transpositions of adjacent digits (such as typing "1243" instead of "1234"), but there are many other possible transcription errors that are not caught, for example "3412" instead of "1234". It has a high code rate (only one check digit added to 14 or 15 payload digits) and is simple to use. It cannot be used to correct the error, and is therefore of value only in a context where other means are used for correction.

ISBN

The International Standard Book Number (ISBN) is a 13-digit number that uniquely identifies a book or something similar to a book. Different editions of the same book may have different ISBNs. The book may be in printed form or it may be an e-book, audio cassette, or software. ISBNs are not of much interest to consumers, but they are useful to booksellers, libraries, authors, publishers, and distributors.

The system was created by the British bookseller W. H. Smith in 1966 using 9-digit numbers, then upgraded in 1970 for international use by prepending 0 to existing numbers, and then upgraded in 2007 to 13-digit numbers by prepending 978 and recalculating the check digit.

A book's ISBN appears as a number following the letters "ISBN", most often on the back of the dust jacket or the back cover of a paperback. Typically it is near some bar codes and is frequently rendered in a machine-readable font.

There are five parts to an ISBN (four prior to 2007), of variable length, separated by hyphens. First is the prefix 978 (missing prior to 2007). When the numbers using this prefix are exhausted, the prefix 979 will be used. Next is a country identifier (or groups of countries or areas sharing a common language).

Next is a number that identifies a particular publisher. Then is the identifier of the title, and finally the single check digit. Country identifiers are assigned by the International ISBN Agency, located in Berlin. Publisher identifiers are assigned within the country or area represented, and title identifiers are assigned by the publishers. The check digit is calculated as described below.

For example, consider ISBN 0-9764731-0-0 (which is in the pre-2007 format). The language area code of 0 represents English speaking countries. The publisher 9764731 is the Department of Electrical Engineering and Computer Science, MIT. The item identifier 0 represents the book "The Electron and the Bit." The identifier is 0 resulted from the fact that this book is the first published using an ISBN by this publisher. The fact that 7 digits were used to identify the publisher and only one the item reflects the reality that this is a very small publisher that will probably not need more than ten ISBNs. ISBNs can be purchased in sets of 10 (for $269.95 as of 2007), 100 ($914.95), 1000 ($1429.95) or 10,000 ($3449.95) and the publisher identifiers assigned at the time of purchase would have 7, 6, 5, or 4 digits, respectively. This arrangement conveniently handles many small publishers and a few large publishers.

Publication trends favoring many small publishers rather than fewer large publishers may strain the ISBN system. Small publishers have to buy numbers at least 10 at a time, and if only one or two books are published, the unused numbers cannot be used by another publisher. Organizations that are primarily not publishers but occasionally publish a book are likely to lose the unused ISBNs because nobody can remember where they were last put.

Books published in 2007 and later have a 13-digit ISBN which is designed to be compatible with UPC (Universal Product Code) barcodes widely used in stores. The procedure for finding UPC check digits is used for 13-digit ISBNs. Start with the 12 digits (without the check digit). Add the first, third, fifth, and other digits in odd-number positions together and multiply the sum by 3. Then add the result to the sum of digits in the even-numbered positions (2, 4, 6, 8 10, and 12). Subtract the result from the next higher multiple of 10. The result, a number between 0 and 9, inclusive, is the desired check digit.

This technique yields a code with a large code rate (0.92) that catches all single-digit errors but not all transposition errors.

For books published prior to 2007, the check digit can be calculated by the following procedure. Start with the nine-digit number (without the check digit). Multiply each by its position, with the left-most position being 1, and the right-most position 9. Add those products, and find the sum's residue modulo 11 (that is, the number you have to subtract in order to make the result a multiple of 11). The result is a number between 0 and 10. That is the check digit. For example, for ISBN 0-9764731-0-0, $1 \times 0 + 2 \times 9 + 3 \times 7 + 4 \times 6 + 5 \times 4 + 6 \times 7 + 7 \times 3 + 8 \times 1 + 9 \times 0 = 154$ which is 0 mod 11.

If the check digit is less than ten, it is used in the ISBN. If the check digit is 10, the letter X is used instead (this is the Roman numeral for ten). If you look at several books, you will discover every so often the check digit X.

This technique yields a code with a large code rate (0.9) that is effective in detecting the transposition of two adjacent digits or the alteration of any single digit.

ISSN

The International Standard Serial Number (ISSN) is an 8-digit number that uniquely identifies print or non-print serial publications. An ISSN is written in the form ISSN 1234-5678 in each issue of the serial. They are usually not even noticed by the general public, but are useful to publishers, distributors, and libraries.

ISSNs are used for newspapers, magazines, and many other types of periodicals including journals, society transactions, monographic series, and even blogs. An ISSN applies to the series as a whole, which is expected to continue indefinitely, rather than individual issues. The assignments are permanent—if a serial ceases to publish, the ISSN is not recovered, and if a serial changes its name a new ISSN is required. In the United States ISSNs are issued, one at a time, without charge, by an office at the Library of Congress.

Unlike ISBNs, there is no meaning associated with parts of an ISSN, except that the first seven digits form a unique number, and the last is a check digit. No more than 10,000,000 ISSNs can ever be assigned unless the format is changed. As of 2006, there were 1,284,413 assigned worldwide, including 57,356 issued that year.

The check digit is calculated by the following procedure. Start with the seven-digit number (without the check digit). Multiply each digit by its (backwards) position, with the left-most position being 8 and the right-most position 2. Add up these products and subtract from the next higher multiple of 11. The result is a number between 0 and 10. If it is less than 10, that digit is the check digit. If it is equal to 10, the check digit is X. The check digit becomes the eighth digit of the final ISSN.

Chapter 5

Probability

We have been considering a model of an information handling system in which symbols from an input are encoded into bits, which are then sent across a "channel" to a receiver and get decoded back into symbols. See Figure 5.1.

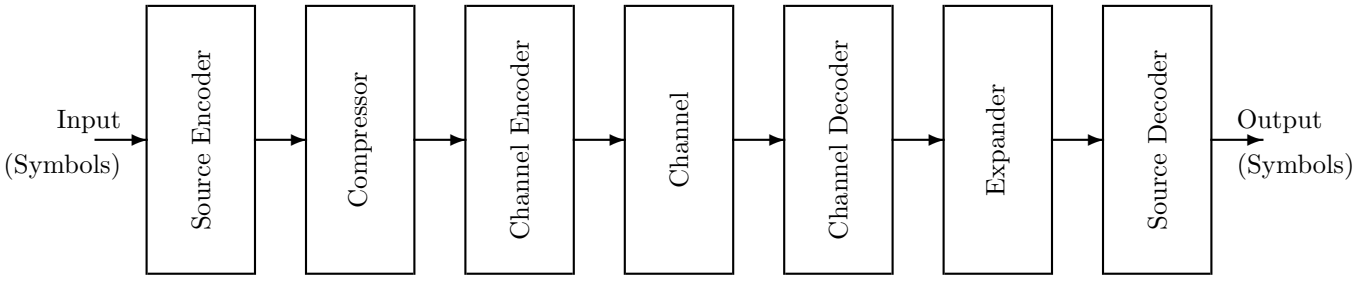


Figure 5.1: Communication system

In earlier chapters of these notes we have looked at various components in this model. Now we return to the source and model it more fully, in terms of probability distributions.

The source provides a symbol or a sequence of symbols, selected from some set. The selection process might be an experiment, such as flipping a coin or rolling dice. Or it might be the observation of actions not caused by the observer. Or the sequence of symbols could be from a representation of some object, such as characters from text, or pixels from an image.

We consider only cases with a finite number of symbols to choose from, and only cases in which the symbols are both mutually exclusive (only one can be chosen at a time) and exhaustive (one is actually chosen). Each choice constitutes an "outcome" and our objective is to trace the sequence of outcomes, and the information that accompanies them, as the information travels from the input to the output. To do that, we need to be able to say what the outcome is, and also our knowledge about some properties of the outcome.

If we know the outcome, we have a perfectly good way of denoting the result. We can simply name the symbol chosen, and ignore all the rest of the symbols, which were not chosen. But what if we do not yet know the outcome, or are uncertain to any degree? How are we supposed to express our state of knowledge if there is uncertainty? We will use the mathematics of probability for this purpose.

To illustrate this important idea, we will use examples based on the characteristics of MIT students. The official count of students at MIT[1] for Fall 2007 includes the data in Table 5.1, which is reproduced in Venn diagram format in Figure 5.2.

	Women	Men	Total
Freshmen	496	577	1,073
Undergraduates	1,857	2,315	4,172
Graduate Students	1,822	4,226	6,048
Total Students	3,679	6,541	10,220

Table 5.1: Demographic data for MIT, Fall 2007

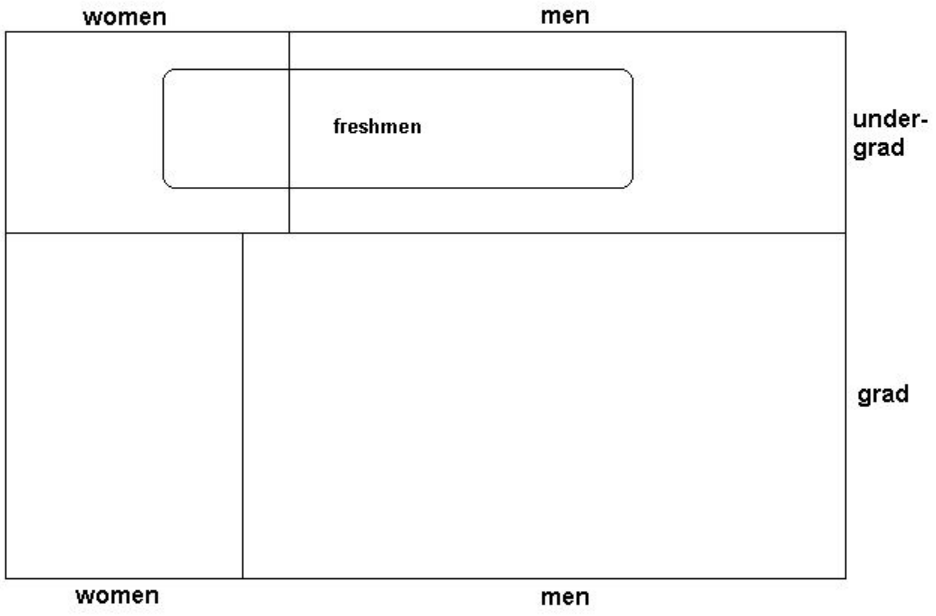


Figure 5.2: A Venn diagram of MIT student data, with areas that should be proportional to the sizes of the subpopulations.

Suppose an MIT freshman is selected (the symbol being chosen is an individual student, and the set of possible symbols is the 1073 freshmen), and you are not informed who it is. You wonder whether it is a woman or a man. Of course if you knew the identity of the student selected, you would know the gender. But if not, how could you characterize your knowledge? What is the likelihood, or probability, that a woman was selected?

Note that 46% of the 2007 freshman class (496/1,073) are women. This is a fact, or a statistic, which may or may not represent the probability the freshman chosen is a woman. If you had reason to believe that all freshmen were equally likely to be chosen, you might decide that the probability of it being a woman is 46%. But what if you are told that the selection is made in the corridor of McCormick Hall (a women's dormitory)? In that case the probability that the freshman chosen is a woman is probably higher than 46%. Statistics and probabilities can both be described using the same mathematics (to be developed next), but they are different things.

[1] all students: http://web.mit.edu/registrar/www/stats/yreportfinal.html,
all women: http://web.mit.edu/registrar/www/stats/womenfinal.html

5.1 Events

Like many branches of mathematics or science, probability theory has its own nomenclature in which a word may mean something different from or more specific than its everyday meaning. Consider the two words **event**, which has several everyday meanings, and **outcome**. Merriam-Webster's Collegiate Dictionary gives these definitions that are closest to the technical meaning in probability theory:

- **outcome**: something that follows as a result or consequence
- **event**: a subset of the possible outcomes of an experiment

In our context, **outcome** is the symbol selected, whether or not it is known to us. While it is wrong to speak of the outcome of a selection that has not yet been made, it is all right to speak of the set of possible outcomes of selections that are contemplated. In our case this is the set of all symbols. As for the term **event**, its most common everyday meaning, which we do not want, is something that happens. Our meaning, which is quoted above, is listed last in the dictionary. We will use the word in this restricted way because we need a way to estimate or characterize our knowledge of various properties of the symbols. These properties are things that either do or do not apply to each symbol, and a convenient way to think of them is to consider the set of all symbols being divided into two subsets, one with that property and one without. When a selection is made, then, there are several events. One is the outcome itself. This is called a **fundamental event**. Others are the selection of a symbol with particular properties.

Even though, strictly speaking, an event is a set of possible outcomes, it is common in probability theory to call the experiments that produce those outcomes events. Thus we will sometimes refer to a selection as an event.

For example, suppose an MIT freshman is selected. The specific person chosen is the outcome. The fundamental event would be that person, or the selection of that person. Another event would be the selection of a woman (or a man). Another event might be the selection of someone from California, or someone older than 18, or someone taller than six feet. More complicated events could be considered, such as a woman from Texas, or a man from Michigan with particular SAT scores.

The special event in which any symbol at all is selected, is certain to happen. We will call this event the **universal event**, after the name for the corresponding concept in set theory. The special "event" in which no symbol is selected is called the **null event**. The null event cannot happen because an outcome is only defined after a selection is made.

Different events may or may not overlap, in the sense that two or more could happen with the same outcome. A set of events which do not overlap is said to be **mutually exclusive**. For example, the two events that the freshman chosen is (1) from Ohio, or (2) from California, are mutually exclusive.

Several events may have the property that at least one of them happens when any symbol is selected. A set of events, one of which is sure to happen, is known as **exhaustive**. For example, the events that the freshman chosen is (1) younger than 25, or (2) older than 17, are exhaustive, but not mutually exclusive.

A set of events that are both mutually exclusive and exhaustive is known as a **partition**. The partition that consists of all the fundamental events will be called the **fundamental partition**. In our example, the two events of selecting a woman and selecting a man form a partition, and the fundamental events associated with each of the 1073 personal selections form the fundamental partition.

A partition consisting of a small number of events, some of which may correspond to many symbols, is known as a **coarse-grained partition** whereas a partition with many events is a **fine-grained partition**. The fundamental partition is as fine-grained as any. The partition consisting of the universal event and the null event is as coarse-grained as any.

Although we have described events as though there is always a fundamental partition, in practice this partition need not be used.

5.2 Known Outcomes

Once you know an outcome, it is straightforward to denote it. You merely specify which symbol was selected. If the other events are defined in terms of the symbols, you then know which of those events has occurred. However, until the outcome is known you cannot express your state of knowledge in this way. And keep in mind, of course, that your knowledge may be different from another person's knowledge, i.e., knowledge is subjective, or as some might say, "observer-dependent."

Here is a more complicated way of denoting a known outcome, that is useful because it can generalize to the situation where the outcome is not yet known. Let i be an index running over a partition. Because the number of symbols is finite, we can consider this index running from 0 through $n-1$, where n is the number of events in the partition. Then for any particular event A_i in the partition, define $p(A_i)$ to be either 1 (if the corresponding outcome is selected) or 0 (if not selected). Within any partition, there would be exactly one i for which $p(A_i) = 1$ and all the other $p(A_i)$ would be 0. This same notation can apply to events that are not in a partition—if the event A happens as a result of the selection, then $p(A) = 1$ and otherwise $p(A) = 0$.

It follows from this definition that $p(\text{universal event}) = 1$ and $p(\text{null event}) = 0$.

5.3 Unknown Outcomes

If the symbol has not yet been selected, or you do not yet know the outcome, then each $p(A)$ can be given a number between 0 and 1, higher numbers representing a greater belief that this event will happen, and lower numbers representing a belief that this event will probably not happen. If you are certain that some event A is impossible then $p(A) = 0$. If and when the outcome is learned, each $p(A)$ can be adjusted to 0 or 1. Again note that $p(A)$ depends on your state of knowledge and is therefore subjective.

The ways these numbers should be assigned to best express our knowledge will be developed in later chapters. However, we do require that they obey the fundamental axioms of probability theory, and we will call them probabilities (the set of probabilities that apply to a partition will be called a probability distribution). By definition, for any event A

$$0 \leq p(A) \leq 1 \tag{5.1}$$

In our example, we can then characterize our understanding of the gender of a freshman not yet selected (or not yet known) in terms of the probability $p(W)$ that the person selected is a woman. Similarly, $p(CA)$ might denote the probability that the person selected is from California.

To be consistent with probability theory, if some event A happens only upon the occurrence of any of certain other events A_i that are mutually exclusive (for example because they are from a partition) then $p(A)$ is the sum of the various $p(A_i)$ of those events:

$$p(A) = \sum_i p(A_i) \tag{5.2}$$

where i is an index over the events in question. This implies that for any partition, since $p(\text{universal event}) = 1$,

$$1 = \sum_i p(A_i) \tag{5.3}$$

where the sum here is over all events in the partition.

5.4 Joint Events and Conditional Probabilities

You may be interested in the probability that the symbol chosen has two different properties. For example, what is the probability that the freshman chosen is a woman from Texas? Can we find this, $p(W, TX)$, if we

know the probability that the choice is a woman, $p(W)$, and the probability that the choice is from Texas, $p(TX)$?

Not in general. It might be that 47% of the freshmen are women, and it might be that (say) 5% of the freshmen are from Texas, but those facts alone do not guarantee that there are any women freshmen from Texas, let alone how many there might be.

However, if it is known or assumed that the two events are independent (the probability of one does not depend on whether the other event occurs), then the probability of the joint event (both happening) can be found. It is the product of the probabilities of the two events. In our example, if the percentage of women among freshmen from Texas is known to be the same as the percentage of women among all freshmen, then

$$p(W, TX) = p(W)p(TX) \tag{5.4}$$

Since it is unusual for two events to be independent, a more general formula for joint events is needed. This formula makes use of "conditional probabilities," which are probabilities of one event given that another event is known to have happened. In our example, the conditional probability of the selection being a woman, given that the freshman selected is from Texas, is denoted $p(W \mid TX)$ where the vertical bar, read "given," separates the two events—the conditioning event on the right and the conditioned event on the left. If the two events are independent, then the probability of the conditioned event is the same as its normal, or "unconditional" probability.

In terms of conditional probabilities, the probability of a joint event is the probability of one of the events times the probability of the other event given that the first event has happened:

$$\begin{aligned} p(A, B) &= p(B)p(A \mid B) \\ &= p(A)p(B \mid A) \end{aligned} \tag{5.5}$$

Note that either event can be used as the conditioning event, so there are two formulas for this joint probability. Using these formulas you can calculate one of the conditional probabilities from the other, even if you don't care about the joint probability.

This formula is known as Bayes' Theorem, after Thomas Bayes, the eighteenth century English mathematician who first articulated it. We will use Bayes' Theorem frequently. This theorem has remarkable generality. It is true if the two events are physically or logically related, and it is true if they are not. It is true if one event causes the other, and it is true if that is not the case. It is true if the outcome is known, and it is true if the outcome is not known.

Thus the probability $p(W, TX)$ that the student chosen is a woman from Texas is the probability $p(TX)$ that a student from Texas is chosen, times the probability $p(W \mid TX)$ that a woman is chosen given that the choice is a Texan. It is also the probability $P(W)$ that a woman is chosen, times the probability $p(TX \mid W)$ that someone from Texas is chosen given that the choice is a woman.

$$\begin{aligned} p(W, TX) &= p(TX)p(W \mid TX) \\ &= p(W)p(TX \mid W) \end{aligned} \tag{5.6}$$

As another example, consider the table of students above, and assume that one is picked from the entire student population "at random" (meaning with equal probability for all individual students). What is the probability $p(M, G)$ that the choice is a male graduate student? This is a joint probability, and we can use Bayes' Theorem if we can discover the necessary conditional probability.

The fundamental partition in this case is the 10,206 fundamental events in which a particular student is chosen. The sum of all these probabilities is 1, and by assumption all are equal, so each probability is 1/10,220 or about 0.01%.

The probability that the selection is a graduate student $p(G)$ is the sum of all the probabilities of the 048 fundamental events associated with graduate students, so $p(G) = 6{,}048/10{,}220$.

Given that the selection is a graduate student, what is the conditional probability that the choice is a man? We now look at the set of graduate students and the selection of one of them. The new fundamental partition is the 6,048 possible choices of a graduate student, and we see from the table above that 4,226 of these are men. The probabilities of this new (conditional) selection can be found as follows. The original choice was "at random" so all students were equally likely to have been selected. In particular, all graduate students were equally likely to have been selected, so the new probabilities will be the same for all 6,048. Since their sum is 1, each probability is 1/6,048. The event of selecting a man is associated with 4,226 of these new fundamental events, so the conditional probability $p(M \mid G) = 4{,}226/6{,}048$. Therefore from Bayes' Theorem:

$$\begin{aligned} p(M,G) &= p(G)p(M \mid G) \\ &= \frac{6{,}048}{10{,}220} \times \frac{4{,}226}{6{,}048} \\ &= \frac{4{,}226}{10{,}220} \end{aligned} \tag{5.7}$$

This problem can be approached the other way around: the probability of choosing a man is $p(M) = 6{,}541/10{,}220$ and the probability of the choice being a graduate student given that it is a man is $p(G \mid M) = 4{,}226/6{,}541$ so (of course the answer is the same)

$$\begin{aligned} p(M,G) &= p(M)p(G \mid M) \\ &= \frac{6{,}541}{10{,}220} \times \frac{4{,}226}{6{,}541} \\ &= \frac{4{,}226}{10{,}220} \end{aligned} \tag{5.8}$$

5.5 Averages

Suppose we are interested in knowing how tall the freshman selected in our example is. If we know who is selected, we could easily discover his or her height (assuming the height of each freshmen is available in some data base). But what if we have not learned the identity of the person selected? Can we still estimate the height?

At first it is tempting to say we know nothing about the height since we do not know who is selected. But this is clearly not true, since experience indicates that the vast majority of freshmen have heights between 60 inches (5 feet) and 78 inches (6 feet 6 inches), so we might feel safe in estimating the height at, say, 70 inches. At least we would not estimate the height as 82 inches.

With probability we can be more precise and calculate an estimate of the height without knowing the selection. And the formula we use for this calculation will continue to work after we learn the actual selection and adjust the probabilities accordingly.

Suppose we have a partition with events A_i each of which has some value for an attribute like height, say h_i. Then the average value (also called the expected value) H_{av} of this attribute would be found from the probabilities associated with each of these events as

$$H_{av} = \sum_i p(A_i)h_i \tag{5.9}$$

where the sum is over the partition.

This sort of formula can be used to find averages of many properties, such as SAT scores, weight, age, or net wealth. It is not appropriate for properties that are not numerical, such as gender, eye color, personality, or intended scholastic major.

Note that this definition of average covers the case where each event in the partition has a value for the attribute like height. This would be true for the height of freshmen only for the fundamental partition. We would like a similar way of calculating averages for other partitions, for example the partition of men and women. The problem is that not all men have the same height, so it is not clear what to use for h_i in Equation 5.9.

The solution is to define an average height of men in terms of a finer grained partition such as the fundamental partition. Bayes' Theorem is useful in this regard. Note that the probability that freshman i is chosen given the choice is known to be a man is

$$p(A_i \mid M) = \frac{p(A_i)p(M \mid A_i)}{p(M)} \tag{5.10}$$

where $p(M \mid A_i)$ is particularly simple—it is either 1 or 0 depending on whether freshman i is a man or a woman. Then the average height of male freshmen is

$$H_{av}(M) = \sum_i p(A_i \mid M)h_i \tag{5.11}$$

and similarly for the women,

$$H_{av}(W) = \sum_i p(A_i \mid W)h_i \tag{5.12}$$

Then the average height of all freshmen is given by a formula exactly like Equation 5.9:

$$H_{av} = p(M)H_{av}(M) + p(W)H_{av}(W) \tag{5.13}$$

These formulas for averages are valid if all $p(A_i)$ for the partition in question are equal (e.g., if a freshman is chosen "at random"). But they are more general—they are also valid for any probability distribution $p(A_i)$.

The only thing to watch out for is the case where one of the events has probability equal to zero, e.g., if you wanted the average height of freshmen from Nevada and there didn't happen to be any.

5.6 Information

We want to express quantitatively the information we have or lack about the choice of symbol. After we learn the outcome, we have no uncertainty about the symbol chosen or about its various properties, and which events might have happened as a result of this selection. However, before the selection is made or at least before we know the outcome, we have some uncertainty. How much?

After we learn the outcome, the information we now possess could be told to another by specifying the symbol chosen. If there are two possible symbols (such as heads or tails of a coin flip) then a single bit could be used for that purpose. If there are four possible events (such as the suit of a card drawn from a deck) the outcome can be expressed in two bits. More generally, if there are n possible outcomes then $\log_2 n$ bits are needed.

The notion here is that the amount of information we learn upon hearing the outcome is the minimum number of bits that could have been used to tell us, i.e., to specify the symbol. This approach has some merit but has two defects.

First, an actual specification of one symbol by means of a sequence of bits requires an integral number of bits. What if the number of symbols is not an integral power of two? For a single selection, there may not be much that can be done, but if the source makes repeated selections and these are all to be specified, they can be grouped together to recover the fractional bits. For example if there are five possible symbols, then three bits would be needed for a single symbol, but the 25 possible combinations of two symbols could be communicated with five bits (2.5 bits per symbol), and the 125 combinations of three symbols could get by with seven bits (2.33 bits per symbol). This is not much greater than $\log_2(5)$ which is 2.32 bits.

Second, different events may have different likelihoods of being selected. We have seen how to model our state of knowledge in terms of probabilities. If we already know the result (one $p(A_i)$ equals 1 and all others equal 0), then no further information is gained because there was no uncertainty before. Our definition of information should cover that case.

Consider a class of 32 students, of whom two are women and 30 are men. If one student is chosen and our objective is to know which one, our uncertainty is initially five bits, since that is what would be necessary to specify the outcome. If a student is chosen at random, the probability of each being chosen is 1/32. The choice of student also leads to a gender event, either "woman chosen" with probability $p(W) = 2/32$ or "man chosen" with probability $p(M) = 30/32$.

How much information do we gain if we are told that the choice is a woman but not told which one? Our uncertainty is reduced from five bits to one bit (the amount necessary to specify which of the two women it was). Therefore the information we have gained is four bits. What if we are told that the choice is a man but not which one? Our uncertainty is reduced from five bits to $\log_2(30)$ or 4.91 bits. Thus we have learned 0.09 bits of information.

The point here is that if we have a partition whose events have different probabilities, we learn different amounts from different outcomes. If the outcome was likely we learn less than if the outcome was unlikely. We illustrated this principle in a case where each outcome left unresolved the selection of an event from an underlying, fundamental partition, but the principle applies even if we don't care about the fundamental partition. The information learned from outcome i is $\log_2(1/p(A_i))$. Note from this formula that if $p(A_i) = 1$ for some i, then the information learned from that outcome is 0 since $\log_2(1) = 0$. This is consistent with what we would expect.

If we want to quantify our uncertainty before learning an outcome, we cannot use any of the information gained by specific outcomes, because we would not know which to use. Instead, we have to average over all possible outcomes, i.e., over all events in the partition with nonzero probability. The average information per event is found by multiplying the information for each event A_i by $p(A_i)$ and summing over the partition:

$$I = \sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) \tag{5.14}$$

This quantity, which is of fundamental importance for characterizing the information of sources, is called the **entropy** of a source. The formula works if the probabilities are all equal and it works if they are not; it works after the outcome is known and the probabilities adjusted so that one of them is 1 and all the others 0; it works whether the events being reported are from a fundamental partition or not.

In this and other formulas for information, care must be taken with events that have zero probability. These cases can be treated as though they have a very small but nonzero probability. In this case the logarithm, although it approaches infinity for an argument approaching infinity, does so very slowly. The product of that factor times the probability approaches zero, so such terms can be directly set to zero even though the formula might suggest an indeterminate result, or a calculating procedure might have a "divide by zero" error.

5.7 Properties of Information

It is convenient to think of physical quantities as having dimensions. For example, the dimensions of velocity are length over time, and so velocity is expressed in meters per second. In a similar way it is convenient to think of information as a physical quantity with dimensions. Perhaps this is a little less natural, because probabilities are inherently dimensionless. However, note that the formula uses logarithms to the base 2. The choice of base amounts to a scale factor for information. In principle any base k could be used, and related to our definition by the identity

$$\log_k(x) = \frac{\log_2(x)}{\log_2(k)} \tag{5.15}$$

With base-2 logarithms the information is expressed in bits. Later, we will find natural logarithms to be useful.

If there are two events in the partition with probabilities p and $(1-p)$, the information per symbol is

$$I = p\log_2\left(\frac{1}{p}\right) + (1-p)\log_2\left(\frac{1}{1-p}\right) \tag{5.16}$$

which is shown, as a function of p, in Figure 5.3. It is largest (1 bit) for $p = 0.5$. Thus the information is a maximum when the probabilities of the two possible events are equal. Furthermore, for the entire range of probabilities between $p = 0.4$ and $p = 0.6$ the information is close to 1 bit. It is equal to 0 for $p = 0$ and for $p = 1$. This is reasonable because for such values of p the outcome is certain, so no information is gained by learning it.

For partitions with more than two possible events the information per symbol can be higher. If there are n possible events the information per symbol lies between 0 and $\log_2(n)$ bits, the maximum value being achieved when all probabilities are equal.

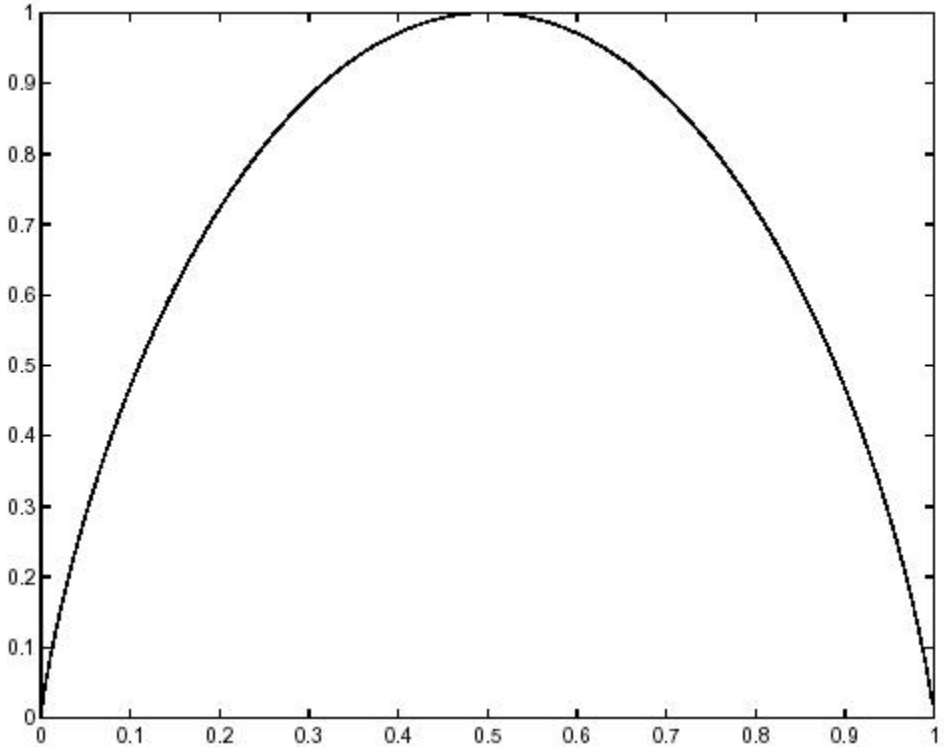


Figure 5.3: Entropy of a source with two symbols as a function of p, one of the two probabilities

5.8 Efficient Source Coding

If a source has n possible symbols then a fixed-length code for it would require $\log_2(n)$ (or the next higher integer) bits per symbol. The average information per symbol I cannot be larger than this but might be smaller, if the symbols have different probabilities. Is it possible to encode a stream of symbols from such a source with fewer bits on average, by using a variable-length code with fewer bits for the more probable symbols and more bits for the less probable symbols?

Certainly. Morse Code is an example of a variable length code which does this quite well. There is a general procedure for constructing codes of this sort which are very efficient (in fact, they require an average of less than $I+1$ bits per symbol, even if I is considerably below $\log_2(n)$. The codes are called Huffman codes after MIT graduate David Huffman (1925 - 1999), and they are widely used in communication systems. See Section 5.10.

5.9 Detail: Life Insurance

An example of statistics and probability in everyday life is their use in life insurance. We consider here only one-year term insurance (insurance companies are very creative in marketing more complex policies that combine aspects of insurance, savings, investment, retirement income, and tax minimization).

When you take out a life insurance policy, you pay a premium of so many dollars and, if you die during the year, your beneficiaries are paid a much larger amount. Life insurance can be thought of in many ways.

From a gambler's perspective, you are betting that you will die and the insurance company is betting that you will live. Each of you can estimate the probability that you will die, and because probabilities are subjective, they may differ enough to make such a bet seem favorable to both parties (for example, suppose you know about a threatening medical situation and do not disclose it to the insurance company). Insurance companies use mortality tables such as Table 5.2 (shown also in Figure 5.4) for setting their rates. (Interestingly, insurance companies also sell annuities, which from a gambler's perspective are bets the other way around—the company is betting that you will die soon, and you are betting that you will live a long time.)

Another way of thinking about life insurance is as a financial investment. Since insurance companies on average pay out less than they collect (otherwise they would go bankrupt), investors would normally do better investing their money in another way, for example by putting it in a bank.

Most people who buy life insurance, of course, do not regard it as either a bet or an investment, but rather as a safety net. They know that if they die, their income will cease and they want to provide a partial replacement for their dependents, usually children and spouses. The premium is small because the probability of death is low during the years when such a safety net is important, but the benefit in the unlikely case of death may be very important to the beneficiaries. Such a safety net may not be as important to very rich people (who can afford the loss of income), single people without dependents, or older people whose children have grown up.

Figure 5.4 and Table 5.2 show the probability of death during one year, as a function of age, for the cohort of U. S. residents born in 1988 (data from The Berkeley Mortality Database[2]).

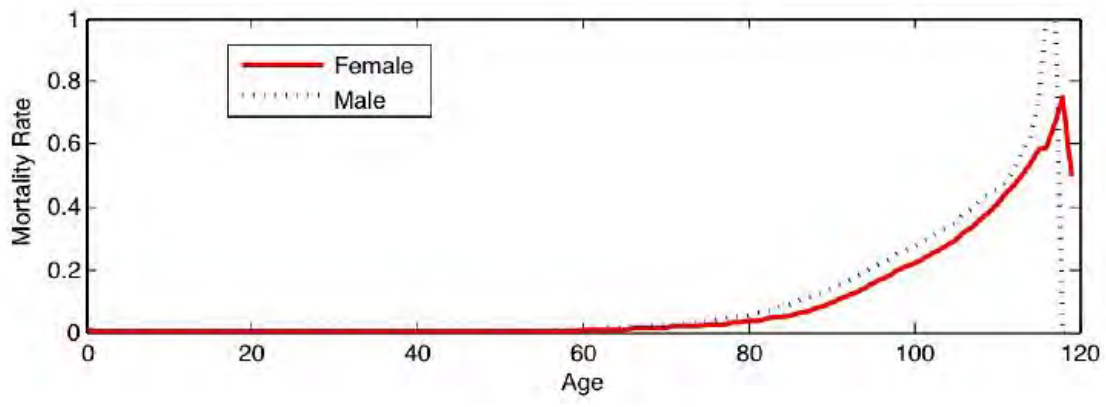


Figure 5.4: Probability of death during one year for U. S. residents born in 1988.

[2]The Berkeley Mortality Database can be accessed online: http://www.demog.berkeley.edu/ bmd/states.html

Age	Female	Male	Age	Female	Male	Age	Female	Male
0	0.008969	0.011126	40	0.000945	0.002205	80	0.035107	0.055995
1	0.000727	0.000809	41	0.001007	0.002305	81	0.038323	0.061479
2	0.000384	0.000526	42	0.00107	0.002395	82	0.041973	0.067728
3	0.000323	0.000415	43	0.001144	0.002465	83	0.046087	0.074872
4	0.000222	0.000304	44	0.001238	0.002524	84	0.050745	0.082817
5	0.000212	0.000274	45	0.001343	0.002605	85	0.056048	0.091428
6	0.000182	0.000253	46	0.001469	0.002709	86	0.062068	0.100533
7	0.000162	0.000233	47	0.001616	0.002856	87	0.06888	0.110117
8	0.000172	0.000213	48	0.001785	0.003047	88	0.076551	0.120177
9	0.000152	0.000162	49	0.001975	0.003295	89	0.085096	0.130677
10	0.000142	0.000132	50	0.002198	0.003566	90	0.094583	0.141746
11	0.000142	0.000132	51	0.002454	0.003895	91	0.105042	0.153466
12	0.000162	0.000203	52	0.002743	0.004239	92	0.116464	0.165847
13	0.000202	0.000355	53	0.003055	0.00463	93	0.128961	0.179017
14	0.000263	0.000559	54	0.003402	0.00505	94	0.142521	0.193042
15	0.000324	0.000793	55	0.003795	0.005553	95	0.156269	0.207063
16	0.000395	0.001007	56	0.004245	0.006132	96	0.169964	0.221088
17	0.000426	0.001161	57	0.004701	0.006733	97	0.183378	0.234885
18	0.000436	0.001254	58	0.005153	0.007357	98	0.196114	0.248308
19	0.000426	0.001276	59	0.005644	0.008028	99	0.208034	0.261145
20	0.000406	0.001288	60	0.006133	0.008728	100	0.220629	0.274626
21	0.000386	0.00131	61	0.006706	0.009549	101	0.234167	0.289075
22	0.000386	0.001312	62	0.007479	0.010629	102	0.248567	0.304011
23	0.000396	0.001293	63	0.008491	0.012065	103	0.263996	0.319538
24	0.000417	0.001274	64	0.009686	0.013769	104	0.280461	0.337802
25	0.000447	0.001245	65	0.011028	0.015702	105	0.298313	0.354839
26	0.000468	0.001226	66	0.012368	0.017649	106	0.317585	0.375342
27	0.000488	0.001237	67	0.013559	0.019403	107	0.337284	0.395161
28	0.000519	0.001301	68	0.014525	0.020813	108	0.359638	0.420732
29	0.00055	0.001406	69	0.015363	0.022053	109	0.383459	0.439252
30	0.000581	0.001532	70	0.016237	0.023393	110	0.408964	0.455882
31	0.000612	0.001649	71	0.017299	0.025054	111	0.437768	0.47619
32	0.000643	0.001735	72	0.018526	0.027029	112	0.466216	0.52
33	0.000674	0.00179	73	0.019972	0.029387	113	0.494505	0.571429
34	0.000705	0.001824	74	0.02163	0.032149	114	0.537037	0.625
35	0.000747	0.001859	75	0.023551	0.035267	115	0.580645	0.75
36	0.000788	0.001904	76	0.02564	0.038735	116	0.588235	1
37	0.00083	0.001961	77	0.027809	0.042502	117	0.666667	1
38	0.000861	0.002028	78	0.030011	0.046592	118	0.75	0
39	0.000903	0.002105	79	0.032378	0.051093	119	0.5	0

Table 5.2: Mortality table for U. S. residents born in 1988

5.10 Detail: Efficient Source Code

Sometimes source coding and compression for communication systems of the sort shown in Figure 5.1 are done together (it is an open question whether there are practical benefits to combining source and channel coding). For sources with a finite number of symbols, but with unequal probabilities of appearing in the input stream, there is an elegant, simple technique for source coding with minimum redundancy.

Example of a Finite Source

Consider a source which generates symbols which are MIT letter grades, with possible values A, B, C, D, and F. You are asked to design a system which can transmit a stream of such grades, produced at the rate of one symbol per second, over a communications channel that can only carry two boolean digits, each 0 or 1, per second.[3]

First, assume nothing about the grade distribution. To transmit each symbol separately, you must encode each as a sequence of bits (boolean digits). Using 7-bit ASCII code is wasteful; we have only five symbols, and ASCII can handle 128. Since there are only five possible values, the grades can be coded in three bits per symbol. But the channel can only process two bits per second.

However, three bits is more than needed. The entropy, assuming there is no information about the probabilities, is at most $\log_2(5) = 2.32$ bits. This is also $\sum_i p(A_i) \log_2(1/p(A_i))$ where there are five such p_i, each equal to 1/5. Why did we need three bits in the first case? Because we had no way of transmitting a partial bit. To do better, we can use "block coding." We group the symbols in blocks of, say, three. The information in each block is three times the information per symbol, or 6.97 bits. Thus a block can be transmitted using 7 boolean bits (there are 125 distinct sequences of three grades and 128 possible patterns available in 7 bits). Of course we also need a way of signifying the end, and a way of saying that the final word transmitted has only one valid grade (or two), not three.

But this is still too many bits per second for the channel to handle. So let's look at the probability distribution of the symbols. In a typical "B-centered" MIT course with good students, the grade distribution might be as shown in Table 5.3. Assuming this as a probability distribution, what is the information per symbol and what is the average information per symbol? This calculation is shown in Table 5.4. The information per symbol is 1.840 bits. Since this is less than two bits perhaps the symbols can be encoded to use this channel.

A	B	C	D	F
25%	50%	12.5%	10%	2.5%

Table 5.3: Distribution of grades for a typical MIT course

Symbol	Probability	Information	Contribution to average
	p	$\log\left(\frac{1}{p}\right)$	$p \log\left(\frac{1}{p}\right)$
A	0.25	2 bits	0.5 bits
B	0.50	1 bit	0.5 bits
C	0.125	3 bits	0.375 bits
D	0.10	3.32 bits	0.332 bits
F	0.025	5.32 bits	0.133 bits
Total	1.00		1.840 bits

Table 5.4: Information distribution for grades in an average MIT distribution

[3]Boolean digits, or binary digits, are usually called "bits." The word "bit" also refers to a unit of information. When a boolean digit carries exactly one bit of information there may be no confusion. But inefficient codes or redundant codes may have boolean digit sequences that are longer than the minimum and therefore carry less than one bit of information per bit. This same confusion attends other units of measure, for example meter, second, etc.

Huffman Code

David A. Huffman (August 9, 1925 - October 6, 1999) was a graduate student at MIT. To solve a homework assignment for a course he was taking from Prof. Robert M. Fano, he devised a way of encoding symbols with different probabilities, with minimum redundancy and without special symbol frames, and hence most compactly. He described it in *Proceedings of the IRE*, September 1962. His algorithm is very simple. The objective is to come up with a "codebook" (a string of bits for each symbol) so that the average code length is minimized. Presumably infrequent symbols would get long codes, and common symbols short codes, just like in Morse code. The algorithm is as follows (you can follow along by referring to Table 5.5):

1. **Initialize:** Let the partial code for each symbol initially be the empty bit string. Define corresponding to each symbol a "symbol-set," with just that one symbol in it, and a probability equal to the probability of that symbol.

2. **Loop-test:** If there is exactly one symbol-set (its probability must be 1) you are done. The codebook consists of the codes associated with each of the symbols in that symbol-set.

3. **Loop-action:** If there are two or more symbol-sets, take the two with the lowest probabilities (in case of a tie, choose any two). Prepend the codes for those in one symbol-set with 0, and the other with 1. Define a new symbol-set which is the union of the two symbol-sets just processed, with probability equal to the sum of the two probabilities. Replace the two symbol-sets with the new one. The number of symbol-sets is thereby reduced by one. Repeat this loop, including the loop test, until only one symbol-set remains.

Note that this procedure generally produces a variable-length code. If there are n distinct symbols, at least two of them have codes with the maximum length.

For our example, we start out with five symbol-sets, and reduce the number of symbol-sets by one each step, until we are left with just one. The steps are shown in Table 5.5, and the final codebook in Table 5.6.

Start: (A='-' p=0.25) (B='-' p=0.5) (C='-' p=0.125) (D='-' p=0.1) (F='-' p=0.025)
Next: (A='-' p=0.25) (B='-' p=0.5) (C='-' p=0.125) (D='1' F='0' p=0.125)
Next: (A='-' p=0.25) (B='-' p=0.5) (C='1' D='01' F='00' p=0.25)
Next: (B='-' p=0.5) (A='1' C='01' D='001' F='000' p=0.5)
Final: (B='1' A='01' C='001' D='0001' F='0000' p=1.0)

Table 5.5: Huffman coding for MIT course grade distribution, where '-' denotes the empty bit string

Symbol	Code
A	0 1
B	1
C	0 0 1
D	0 0 0 1
F	0 0 0 0

Table 5.6: Huffman Codebook for typical MIT grade distribution

Is this code really compact? The most frequent symbol (B) is given the shortest code and the least frequent symbols (D and F) the longest codes, so on average the number of bits needed for an input stream which obeys the assumed probability distribution is indeed short, as shown in Table 5.7.

Compare this table with the earlier table of information content, Table 5.4. Note that the average coded length per symbol, 1.875 bits, is greater than the information per symbol, which is 1.840 bits. This is because the symbols D and F cannot be encoded in fractional bits. If a block of several symbols were considered

Symbol	Code	Probability	Code length	Contribution to average
A	01	0.25	2	0.5
B	1	0.50	1	0.5
C	001	0.125	3	0.375
D	0001	0.1	3.32	0.4
F	0000	0.025	5.32	0.1
Total		1.00		1.875 bits

Table 5.7: Huffman coding of typical MIT grade distribution

together, the average length of the Huffman code could be closer to the actual information per symbol, but not below it.

The channel can handle two bits per second. By using this code, you can transmit over the channel slightly more than one symbol per second on average. You can achieve your design objective.

There are at least six practical things to consider about Huffman codes:

- A burst of D or F grades might occur. It is necessary for the encoder to store these bits until the channel can catch up. How big a buffer is needed for storage? What will happen if the buffer overflows?
- The output may be delayed because of a buffer backup. The time between an input and the associated output is called the "latency." For interactive systems you want to keep latency low. The number of bits processed per second, the "throughput," is more important in other applications that are not interactive.
- The output will not occur at regularly spaced intervals, because of delays caused by bursts. In some real-time applications like audio, this may be important.
- What if we are wrong in our presumed probability distributions? One large course might give fewer A and B grades and more C and D. Our coding would be inefficient, and there might be buffer overflow.
- The decoder needs to know how to break the stream of bits into individual codes. The rule in this case is, break after '1' or after '0000', whichever comes first. However, there are many possible Huffman codes, corresponding to different choices for 0 and 1 prepending in step 3 of the algorithm above. Most do not have such simple rules. It can be hard (although it is always possible) to know where the inter-symbol breaks should be put.
- The codebook itself must be given, in advance, to both the encoder and decoder. This might be done by transmitting the codebook over the channel once.

Another Example

Freshmen at MIT are on a "pass/no-record" system during their first semester on campus. Grades of A, B, and C are reported on transcripts as P (pass), and D and F are not reported (for our purposes we will designate this as no-record, N). Let's design a system to send these P and N symbols to a printer at the fastest average rate. Without considering probabilities, 1 bit per symbol is needed. But the probabilities (assuming the typical MIT grade distribution in Table 5.3) are $p(P) = p(A) + p(B) + p(C) = 0.875$, and $p(N) = p(D) + p(F) = 0.125$. The information per symbol is therefore not 1 bit but only $0.875 \times log_2(1/0.875) + 0.125 \times log_2(1/0.125) = 0.544$ bits. Huffman coding on single symbols does not help. We need to take groups of bits together. For example eleven grades as a block would have 5.98 bits of information and could in principle be encoded to require only 6 bits to be sent to the printer.

Chapter 6

Communications

We have been considering a model of an information handling system in which symbols from an input are encoded into bits, which are then sent across a "channel" to a receiver and get decoded back into symbols, as shown in Figure 6.1.

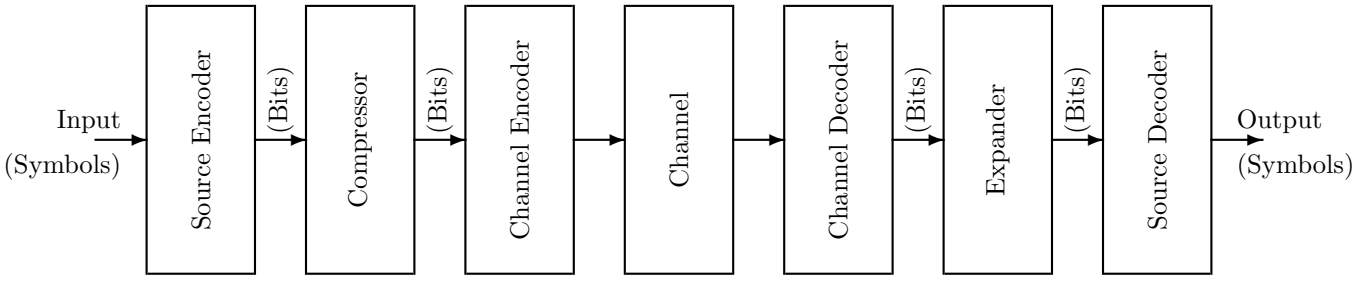


Figure 6.1: Communication system

In this chapter we focus on how fast the information that identifies the symbols can be transferred to the output. The symbols themselves, of course, are not transmitted, but only the information necessary to identify them. This is what is necessary for the stream of symbols to be recreated at the output.

We will model both the source and the channel in a little more detail, and then give three theorems relating to the source characteristics and the channel capacity.

6.1 Source Model

The source is assumed to produce symbols at a rate of R symbols per second. Each symbol is chosen from a finite set of possible symbols, and the index i ranges over the possible symbols. The event of the selection of symbol i will be denoted A_i.

Let us suppose that each event A_i (i.e., the selection of the symbol i) is represented by a different codeword C_i with a length L_i. For fixed-length codes such as ASCII all L_i are the same, whereas for variable-length codes, such as Huffman codes, they are generally different. Since the codewords are patterns of bits, the

number available of each length is limited. For example, there are only four distinct two-bit codewords possible, namely 00, 01, 10, and 11.

An important property of such codewords is that none can be the same as the first portion of another, longer, codeword—otherwise the same bit pattern might result from two or more different messages, and there would be ambiguity. A code that obeys this property is called a **prefix-condition code**, or sometimes an **instantaneous code**.

6.1.1 Kraft Inequality

Since the number of distinct codes of short length is limited, not all codes can be short. Some must be longer, but then the prefix condition limits the available short codes even further. An important limitation on the distribution of code lengths L_i was given by L. G. Kraft, an MIT student, in his 1949 Master's thesis. It is known as the Kraft inequality:

$$\sum_i \frac{1}{2^{L_i}} \leq 1 \tag{6.1}$$

Any valid set of distinct codewords obeys this inequality, and conversely for any proposed L_i that obey it, a code can be found.

For example, suppose a code consists of the four distinct two-bit codewords 00, 01, 10, and 11. Then each $L_i = 2$ and each term in the sum in Equation 6.1 is $1/2^2 = 1/4$. In this case the equation evaluates to an equality, and there are many different ways to assign the four codewords to four different symbols. As another example, suppose there are only three symbols, and the proposed codewords are 00 01, and 11. In this case the Kraft inequality is an inequality. However, because the sum is less than 1, the code can be made more efficient by replacing one of the codewords with a shorter one. In particular, if the symbol represented by 11 is now represented by 1 the result is still a prefix-condition code but the sum would be $(1/2^2) + (1/2^2) + (1/2) = 1$.

The Kraft inequality can be proven easily. Let L_{max} be the length of the longest codeword of a prefix-condition code. There are exactly $2^{L_{max}}$ different patterns of 0 and 1 of this length. Thus

$$\sum_i \frac{1}{2^{L_{max}}} = 1 \tag{6.2}$$

where this sum is over these patterns (this is an unusual equation because the quantity being summed does not depend on i). At least one of these patterns is one of the codewords, but unless this happens to be a fixed-length code there are other shorter codewords. For each shorter codeword of length k ($k < L_{max}$) there are exactly $2^{L_{max}-k}$ patterns that begin with this codeword, and none of those is a valid codeword (because this code is a prefix-condition code). In the sum of Equation 6.2 replace the terms corresponding to those patterns by a single term equal to $1/2^k$. The sum is unchanged. Continue this process with other short codewords. When it is complete, there are terms in the sum corresponding to every codeword, and the sum is still equal to 1. There may be other terms corresponding to patterns that are not codewords—if so, eliminate them from the sum in Equation 6.2. The result is exactly the sum in Equation 6.1 and is less than or equal to 1. The proof is complete.

6.2 Source Entropy

As part of the source model, we assume that each symbol selection is independent of the other symbols chosen, so that the probability $p(A_i)$ does not depend on what symbols have previously been chosen (this model can, of course, be generalized in many ways). The uncertainty of the identity of the next symbol chosen H is the average information gained when the next symbol is made known:

$$H = \sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) \tag{6.3}$$

This quantity is also known as the entropy of the source, and is measured in bits per symbol. The information rate, in bits per second, is $H \cdot R$ where R is the rate at which the source selects the symbols, measured in symbols per second.

6.2.1 Gibbs Inequality

Here we present the Gibbs Inequality, named after the American physicist J. Willard Gibbs (1839–1903)[1], which will be useful to us in later proofs. This inequality states that the entropy is smaller than or equal to any other average formed using the same probabilities but a different function in the logarithm. Specifically,

$$\sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) \leq \sum_i p(A_i) \log_2 \left(\frac{1}{p'(A_i)} \right) \tag{6.4}$$

where $p(A_i)$ is any probability distribution (we will use it for source events and other distributions) and $p'(A_i)$ is any other probability distribution, or more generally any set of numbers such that

$$0 \leq p'(A_i) \leq 1 \tag{6.5}$$

and

$$\sum_i p'(A_i) \leq 1. \tag{6.6}$$

As is true for all probability distributions,

$$\sum_i p(A_i) = 1. \tag{6.7}$$

Equation 6.4 can be proven by noting that the natural logarithm has the property that it is less than or equal to a straight line that is tangent to it at any point, (for example the point $x = 1$ is shown in Figure 6.2). This property is sometimes referred to as concavity or convexity. Thus

$$\ln x \leq (x - 1) \tag{6.8}$$

and therefore, by converting the logarithm base e to the logarithm base 2, we have

$$\log_2 x \leq (x - 1) \log_2 e \tag{6.9}$$

Then

$$\begin{aligned}
\sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) - \sum_i p(A_i) \log_2 \left(\frac{1}{p'(A_i)} \right) &= \sum_i p(A_i) \log_2 \left(\frac{p'(A_i)}{p(A_i)} \right) \\
&\leq \log_2 e \sum_i p(A_i) \left[\frac{p'(A_i)}{p(A_i)} - 1 \right] \\
&= \log_2 e \left(\sum_i p'(A_i) - \sum_i p(A_i) \right) \\
&= \log_2 e \left(\sum_i p'(A_i) - 1 \right) \\
&\leq 0
\end{aligned} \tag{6.10}$$

[1] See a biography of Gibbs at http://www-groups.dcs.st-andrews.ac.uk/%7Ehistory/Biographies/Gibbs.html

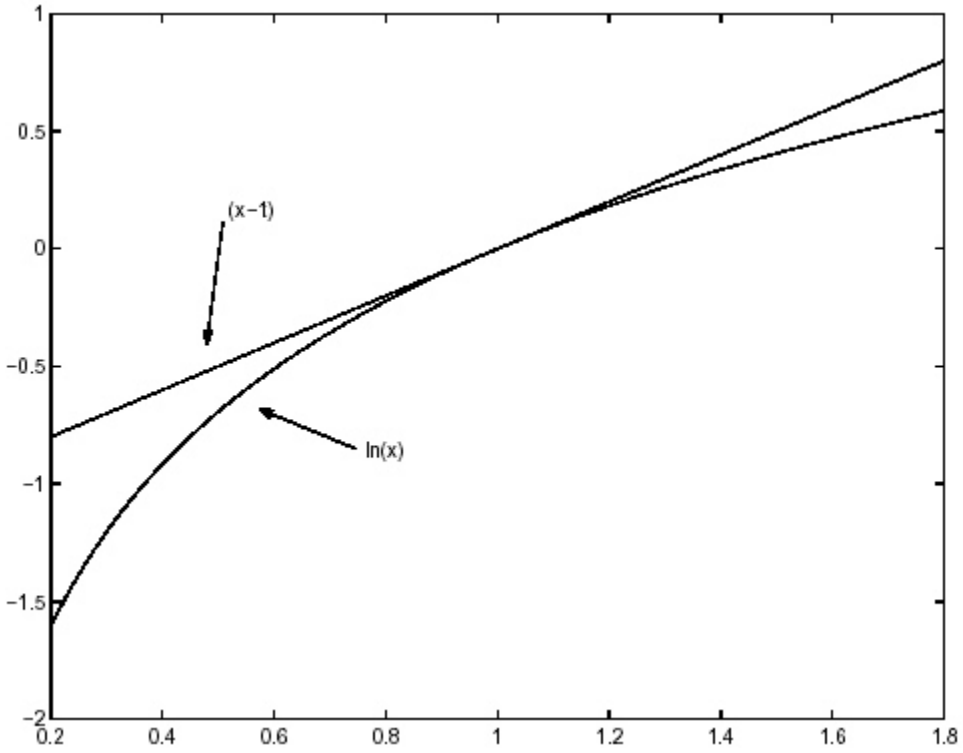


Figure 6.2: Graph of the inequality $\ln x \leq (x-1)$

6.3 Source Coding Theorem

Now getting back to the source model, note that the codewords have an average length, in bits per symbol,

$$L = \sum_i p(A_i) L_i \tag{6.11}$$

For maximum speed the lowest possible average codeword length is needed. The assignment of high-probability symbols to the short codewords can help make L small. Huffman codes are optimal codes for this purpose. However, there is a limit to how short the average codeword can be. Specifically, the Source Coding Theorem states that the average information per symbol is always less than or equal to the average length of a codeword:

$$H \leq L \tag{6.12}$$

This inequality is easy to prove using the Gibbs and Kraft inequalities. Use the Gibbs inequality with $p'(A_i) = 1/2^{L_i}$ (the Kraft inequality assures that the $p'(A_i)$, besides being positive, add up to no more than 1). Thus

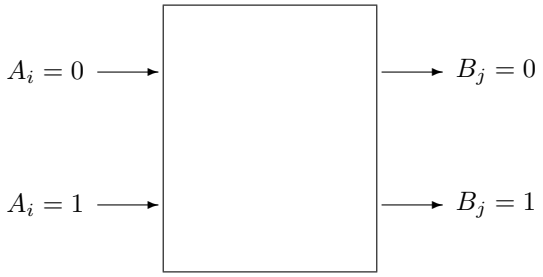


Figure 6.3: Binary Channel

$$\begin{aligned} H &= \sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) \\ &\leq \sum_i p(A_i) \log_2 \left(\frac{1}{p'(A_i)} \right) \\ &= \sum_i p(A_i) \log_2 2^{L_i} \\ &= \sum_i p(A_i) L_i \\ &= L \end{aligned} \tag{6.13}$$

The Source Coding Theorem can also be expressed in terms of rates of transmission in bits per second by multiplying Equation 6.12 by the symbols per second R:

$$HR \leq LR \tag{6.14}$$

6.4 Channel Model

A communication channel accepts input bits and produces output bits. We model the input as the selection of one of a finite number of input states (for the simplest channel, two such states), and the output as a similar event. The language of probability theory will be useful in describing channels. If the channel perfectly changes its output state in conformance with its input state, it is said to be **noiseless** and in that case nothing affects the output except the input. Let us say that the channel has a certain maximum rate W at which its output can follow changes at the input (just as the source has a rate R at which symbols are selected).

We will use the index i to run over the input states, and j to index the output states. We will refer to the input events as A_i and the output events as B_j. You may picture the channel as something with inputs and outputs, as in Figure 6.3, but note that the inputs are not normal signal inputs or electrical inputs to systems, but instead mutually exclusive events, only one of which is true at any one time. For simple channels such a diagram is simple because there are so few possible choices, but for more complicated structures there may be so many possible inputs that the diagrams become impractical (though they may be useful as a conceptual model). For example, a logic gate with three inputs, each of which could be 0 or 1, would have eight inputs in a diagram of this sort. The **binary** channel has two mutually exclusive input states and is the one pictured in Figure 6.3.

For a noiseless channel, where each of n possible input states leads to exactly one output state, each new input state (R per second) can be specified with $\log_2 n$ bits. Thus for the binary channel, $n = 2$, and so the new state can be specified with one bit. The maximum rate at which information supplied to the input can

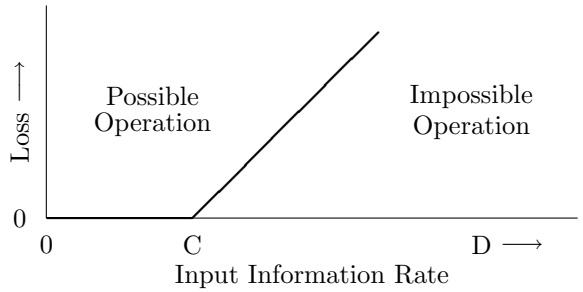


Figure 6.4: Channel Loss Diagram. For an input data rate D, either less than or greater than the channel capacity C, the minimum possible rate at which information is lost is the greater of 0 and $D - C$

affect the output is called the **channel capacity** $C = W \log_2 n$ bits per second. For the binary channel, $C = W$.

If the input is changed at a rate R less than W (or, equivalently, if the information supplied at the input is less than C) then the output can follow the input, and the output events can be used to infer the identity of the input symbols at that rate. If there is an attempt to change the input more rapidly, the channel cannot follow (since W is by definition the maximum rate at which changes at the input affect the output) and some of the input information is lost.

6.5 Noiseless Channel Theorem

If the channel does not introduce any errors, the relation between the information supplied to the input and what is available on the output is very simple. Let the input information rate, in bits per second, be denoted D (for example, D might be the entropy per symbol of a source H expressed in bits per symbol, times the rate of the source R in symbols per second). If $D \leq C$ then the information available at the output can be as high as D (the information at the input), and if $D > C$ then the information available at the output cannot exceed C and so an amount at least equal to $D - C$ is lost. This result is pictured in Figure 6.4.

Note that this result places a limit on how fast information can be transmitted across a given channel. It does not indicate how to achieve results close to this limit. However, it is known how to use Huffman codes to efficiently represent streams of symbols by streams of bits. If the channel is a binary channel it is simply a matter of using that stream of bits to change the input. For other channels, with more than two possible input states, operation close to the limit involves using multiple bits to control the input rapidly.

Achieving high communication speed may (like Huffman code) require representing some infrequently occurring symbols with long codewords. Therefore the rate at which individual bits arrive at the channel input may vary, and even though the average rate may be acceptable, there may be bursts of higher rate, if by coincidence several low-probability symbols happen to be adjacent. It may be necessary to provide temporary storage buffers to accommodate these bursts, and the symbols may not materialize at the output of the system at a uniform rate. Also, to encode the symbols efficiently it may be necessary to consider several of them together, in which case the first symbol would not be available at the output until several symbols had been presented at the input. Therefore high speed operation may lead to high latency. Different communication systems have different tolerance for latency or bursts; for example, latency of more than about 100 milliseconds is annoying in a telephone call, whereas latency of many minutes may be tolerable in e-mail. A list of the needs of some practical communication systems, shown in Section 6.9, reveals a wide variation in required speed, throughput, latency, etc.

6.6 Noisy Channel

If the channel introduces noise then the output is not a unique function of the input. We will model this case by saying that for every possible input (which are mutually exclusive states indexed by i) there may be more than one possible output outcome. Which actually happens is a matter of chance, and we will model the channel by the set of probabilities that each of the output events B_j occurs when each of the possible input events A_i happens. These **transition probabilities** c_{ji} are, of course, probabilities, but they are properties of the channel and do not depend on the probability distribution $p(A_i)$ of the input. Like all probabilities, they have values between 0 and 1

$$0 \leq c_{ji} \leq 1 \tag{6.15}$$

and may be thought of as forming a matrix with as many columns as there are input events, and as many rows as there are output events. Because each input event must lead to exactly one output event,

$$1 = \sum_j c_{ji} \tag{6.16}$$

for each i. (In other words, the sum of c_{ji} in each column i is 1.) If the channel is noiseless, for each value of i exactly one of the various c_{ji} is equal to 1 and all others are 0.

When the channel is driven by a source with probabilities $p(A_i)$, the conditional probabilities of the output events, conditioned on the input events, is

$$p(B_j \mid A_i) = c_{ji} \tag{6.17}$$

The unconditional probability of each output $p(B_j)$ is

$$p(B_j) = \sum_i c_{ji} p(A_i) \tag{6.18}$$

The backward conditional probabilities $p(A_i \mid B_j)$ can be found using Bayes' Theorem:

$$\begin{aligned} p(A_i, B_j) &= p(B_j) p(A_i \mid B_j) \\ &= p(A_i) p(B_j \mid A_i) \\ &= p(A_i) c_{ji} \end{aligned} \tag{6.19}$$

The simplest noisy channel is the symmetric binary channel, for which there is a (hopefully small) probability ε of an error, so

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} 1-\varepsilon & \varepsilon \\ \varepsilon & 1-\varepsilon \end{bmatrix} \tag{6.20}$$

This binary channel is called **symmetric** because the probability of an error for both inputs is the same. If $\varepsilon = 0$ then this channel is noiseless (it is also noiseless if $\varepsilon = 1$, in which case it behaves like an inverter). Figure 6.3 can be made more useful for the noisy channel if the possible transitions from input to output are shown, as in Figure 6.5.

If the output B_j is observed to be in one of its (mutually exclusive) states, can the input A_i that caused it be determined? In the absence of noise, yes; there is no uncertainty about the input once the output is known. However, with noise there is some residual uncertainty. We will calculate this uncertainty in terms of the transition probabilities c_{ji} and define the information that we have learned about the input as a result of knowing the output as the **mutual information**. From that we will define the channel capacity C.

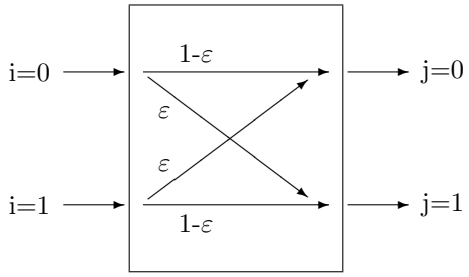


Figure 6.5: Symmetric binary channel

Before we know the output, what is our uncertainty U_{before} about the identity of the input event? This is the entropy of the input:

$$U_{\text{before}} = \sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) \tag{6.21}$$

After some particular output event B_j has been observed, what is the residual uncertainty $U_{\text{after}}(B_j)$ about the input event? A similar formula applies, with $p(A_i)$ replaced by the conditional backward probability $p(A_i \mid B_j)$:

$$U_{\text{after}}(B_j) = \sum_i p(A_i \mid B_j) \log_2 \left(\frac{1}{p(A_i \mid B_j)} \right) \tag{6.22}$$

The amount we learned in the case of this particular output event is the difference between U_{before} and $U_{\text{after}}(B_j)$. The mutual information M is defined as the average, over all outputs, of the amount so learned,

$$M = U_{\text{before}} - \sum_j p(B_j) U_{\text{after}}(B_j) \tag{6.23}$$

It is not difficult to prove that $M \geq 0$, i.e., that our knowledge about the input is not, on average, made more uncertain by learning the output event. To prove this, the Gibbs inequality is used, for each j:

$$\begin{aligned} U_{\text{after}}(B_j) &= \sum_i p(A_i \mid B_j) \log_2 \left(\frac{1}{p(A_i \mid B_j)} \right) \\ &\leq \sum_i p(A_i \mid B_j) \log_2 \left(\frac{1}{p(A_i)} \right) \end{aligned} \tag{6.24}$$

This use of the Gibbs inequality is valid because, for each j, $p(A_i \mid B_j)$ is a probability distribution over i, and $p(A_i)$ is another probability distribution over i, different from the one doing the average. This inequality holds for every value of j and therefore for the average over all j:

$$\begin{aligned}
\sum_j p(B_j)U_{\text{after}}(B_j) &\leq \sum_j p(B_j)\sum_i p(A_i \mid B_j)\log_2\left(\frac{1}{p(A_i)}\right) \\
&= \sum_{ji} p(B_j)p(A_i \mid B_j)\log_2\left(\frac{1}{p(A_i)}\right) \\
&= \sum_{ij} p(B_j \mid A_i)p(A_i)\log_2\left(\frac{1}{p(A_i)}\right) \\
&= \sum_i p(A_i)\log_2\left(\frac{1}{p(A_i)}\right) \\
&= U_{\text{before}} \qquad (6.25)
\end{aligned}$$

We are now in a position to find M in terms of the input probability distribution and the properties of the channel. Substitution in Equation 6.23 and simplification leads to

$$M = \sum_j \left(\sum_i p(A_i)c_{ji}\right)\log_2\left(\frac{1}{\sum_i p(A_i)c_{ji}}\right) - \sum_{ij} p(A_i)c_{ji}\log_2\left(\frac{1}{c_{ji}}\right) \qquad (6.26)$$

Note that Equation 6.26 was derived for the case where the input "causes" the output. At least, that was the way the description went. However, such a cause-and-effect relationship is not necessary. The term **mutual information** suggests (correctly) that it is just as valid to view the output as causing the input, or to ignore completely the question of what causes what. Two alternate formulas for M show that M can be interpreted in either direction:

$$\begin{aligned}
M &= \sum_i p(A_i)\log_2\left(\frac{1}{p(A_i)}\right) - \sum_j p(B_j)\sum_i p(A_i \mid B_j)\log_2\left(\frac{1}{p(A_i \mid B_j)}\right) \\
&= \sum_j p(B_j)\log_2\left(\frac{1}{p(B_j)}\right) - \sum_i p(A_i)\sum_j p(B_j \mid A_i)\log_2\left(\frac{1}{p(B_j \mid A_i)}\right) \qquad (6.27)
\end{aligned}$$

Rather than give a general interpretation of these or similar formulas, let's simply look at the symmetric binary channel. In this case both $p(A_i)$ and $p(B_j)$ are equal to 0.5 and so the first term in the expression for M in Equation 6.26 is 1 and the second term is found in terms of ε:

$$M = 1 - \varepsilon\log_2\left(\frac{1}{\varepsilon}\right) - (1-\varepsilon)\log_2\left(\frac{1}{(1-\varepsilon)}\right) \qquad (6.28)$$

which happens to be 1 bit minus the entropy of a binary source with probabilities ε and $1-\varepsilon$. This is a cup-shaped curve that goes from a value of 1 when $\varepsilon = 0$ down to 0 at $\varepsilon = 0.5$ and then back up to 1 when $\varepsilon = 1$. See Figure 6.6. The interpretation of this result is straightforward. When $\varepsilon = 0$ (or when $\varepsilon = 1$) the input can be determined exactly whenever the output is known, so there is no loss of information. The mutual information is therefore the same as the input information, 1 bit. When $\varepsilon = 0.5$ each output is equally likely, no matter what the input is, so learning the output tells us nothing about the input. The mutual information is 0.

6.7 Noisy Channel Capacity Theorem

The channel capacity of a noisy channel is defined in terms of the mutual information M. However, in general M depends not only on the channel (through the transfer probabilities c_{ji}) but also on the input

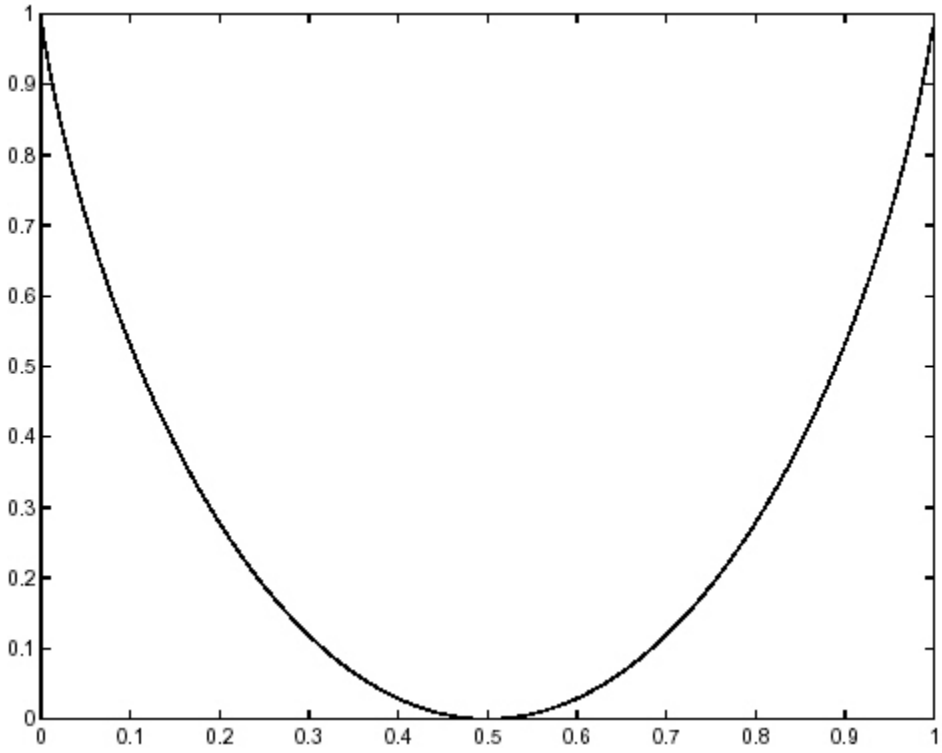


Figure 6.6: Mutual information, in bits, as a function of bit-error probability ε

probability distribution $p(A_i)$. It is more useful to define the channel capacity so that it depends only on the channel, so $M_{\max}$, the maximum mutual information that results from any possible input probability distribution, is used. In the case of the symmetric binary channel, this maximum occurs when the two input probabilities are equal. Generally speaking, going away from the symmetric case offers few if any advantages in engineered systems, and in particular the fundamental limits given by the theorems in this chapter cannot be evaded through such techniques. Therefore the symmetric case gives the right intuitive understanding.

The channel capacity is defined as

$$C = M_{\max} W \tag{6.29}$$

where W is the maximum rate at which the output state can follow changes at the input. Thus C is expressed in bits per second.

The channel capacity theorem was first proved by Shannon in 1948. It gives a fundamental limit to the rate at which information can be transmitted through a channel. If the input information rate in bits per second D is less than C then it is possible (perhaps by dealing with long sequences of inputs together) to code the data in such a way that the error rate is as low as desired. On the other hand, if $D > C$ then this is not possible; in fact the maximum rate at which information about the input can be inferred from learning the output is C. This result is exactly the same as the result for the noiseless channel, shown in Figure 6.4.

This result is really quite remarkable. A capacity figure, which depends only on the channel, was defined and then the theorem states that a code which gives performance arbitrarily close to this capacity can be found. In conjunction with the source coding theorem, it implies that a communication channel can be designed in two stages—first, the source is encoded so that the average length of codewords is equal to its entropy, and second, this stream of bits can then be transmitted at any rate up to the channel capacity with arbitrarily low error. The channel capacity is not the same as the native rate at which the input can change, but rather is degraded from that value because of the noise.

Unfortunately, the proof of this theorem (which is not given here) does not indicate how to go about

finding such a code. In other words, it is not a constructive proof, in which the assertion is proved by displaying the code. In the half century since Shannon published this theorem, there have been numerous discoveries of better and better codes, to meet a variety of high speed data communication needs. However, there is not yet any general theory of how to design codes from scratch (such as the Huffman procedure provides for source coding).

Claude Shannon (1916–2001) is rightly regarded as the greatest figure in communications in all history.[2] He established the entire field of scientific inquiry known today as information theory. He did this work while at Bell Laboratories, after he graduated from MIT with his S.M. in electrical engineering and his Ph.D. in mathematics. It was he who recognized the binary digit as the fundamental element in all communications. In 1956 he returned to MIT as a faculty member. In his later life he suffered from Alzheimer's disease, and, sadly, was unable to attend a symposium held at MIT in 1998 honoring the 50th anniversary of his seminal paper.

6.8 Reversibility

It is instructive to note which operations discussed so far involve loss of information, and which do not.

Some Boolean operations had the property that the input could not be deduced from the output. The *AND* and *OR* gates are examples. Other operations were reversible—the *EXOR* gate, when the output is augmented by one of the two inputs, is an example.

Some sources may be encoded so that all possible symbols are represented by different codewords. This is always possible if the number of symbols is finite. Other sources have an infinite number of possible symbols, and these cannot be encoded exactly. Among the techniques used to encode such sources are binary codes for integers (which suffer from overflow problems) and floating-point representation of real numbers (which suffer from overflow and underflow problems and also from limited precision).

Some compression algorithms are reversible in the sense that the input can be recovered exactly from the output. One such technique is LZW, which is used for text compression and some image compression, among other things. Other algorithms achieve greater efficiency at the expense of some loss of information. Examples are JPEG compression of images and MP3 compression of audio.

Now we have seen that some communication channels are noiseless, and in that case there can be perfect transmission at rates up to the channel capacity. Other channels have noise, and perfect, reversible communication is not possible, although the error rate can be made arbitrarily small if the data rate is less than the channel capacity. For greater data rates the channel is necessarily irreversible.

In all these cases of irreversibility, information is lost, (or at best kept unchanged). Never is information increased in any of the systems we have considered.

Is there a general principle at work here?

[2] See a biography of Shannon at http://www-groups.dcs.st-andrews.ac.uk/%7Ehistory/Biographies/Shannon.html

6.9 Detail: Communication System Requirements

The model of a communication system that we have been developing is shown in Figure 6.1. The source is assumed to emit a stream of symbols. The channel may be a physical channel between different points in space, or it may be a memory which stores information for retrieval at a later time, or it may even be a computation in which the information is processed in some way.

Naturally, different communication systems, though they all might be well described by our model, differ in their requirements. The following table is an attempt to illustrate the range of requirements that are reasonable for modern systems. It is, of course, not complete.

The systems are characterized by four measures: throughput, latency, tolerance of errors, and tolerance to nonuniform rate (bursts). Throughput is simply the number of bits per second that such a system should, to be successful, accommodate. Latency is the time delay of the message; it could be defined either as the delay of the start of the output after the source begins, or a similar quantity about the end of the message (or, for that matter, about any particular features in the message). The numbers for throughput, in MB (megabytes) or kb (kilobits) are approximate.

	Throughput (per second)	Maximum Latency	Errors Tolerated?	Bursts Tolerated?
Computer Memory	many MB	microseconds	no	yes
Hard disk	MB or higher	milliseconds	no	yes
Conversation	??	50 ms	yes; feedback error control	annoying
Telephone	20 kb	100 ms	noise tolerated	no
Radio broadcast	??	seconds	some noise tolerated	no
Instant message	low	seconds	no	yes
Compact Disc	1.4 MB	2 s	no	no
Internet	1 MB	5 s	no	yes
Print queue	1 MB	30 s	no	yes
Fax	14.4 kb	minutes	errors tolerated	yes
Shutter telegraph station	??	5 min	no	yes
E-mail	N/A	1 hour	no	yes
Overnight delivery	large	1 day	no	yes
Parcel delivery	large	days	no	yes
Snail mail	large	days	no	yes

Table 6.1: Various Communication Systems

Chapter 7

Processes

The model of a communication system that we have been developing is shown in Figure 7.1. This model is also useful for some computation systems. The source is assumed to emit a stream of symbols. The channel may be a physical channel between different points in space, or it may be a memory which stores information for retrieval at a later time, or it may be a computation in which the information is processed in some way.

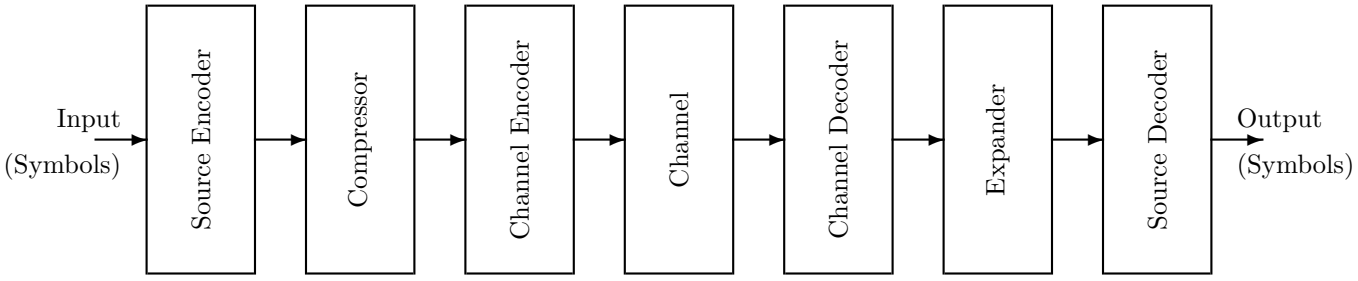


Figure 7.1: Communication system

Figure 7.1 shows the module inputs and outputs and how they are connected. A diagram like this is very useful in portraying an overview of the operation of a system, but other representations are also useful. In this chapter we develop two abstract models that are general enough to represent each of these boxes in Figure 7.1, but show the flow of information quantitatively.

Because each of these boxes in Figure 7.1 processes information in some way, it is called a **processor** and what it does is called a **process**. The processes we consider here are

- **Discrete:** The inputs are members of a set of mutually exclusive possibilities, only one of which occurs at a time, and the output is one of another discrete set of mutually exclusive events.
- **Finite:** The set of possible inputs is finite in number, as is the set of possible outputs.
- **Memoryless:** The process acts on the input at some time and produces an output based on that input, ignoring any prior inputs.

- **Nondeterministic:** The process may produce a different output when presented with the same input a second time (the model is also valid for deterministic processes). Because the process is nondeterministic the output may contain random **noise**.
- **Lossy:** It may not be possible to "see" the input from the output, i.e., determine the input by observing the output. Such processes are called **lossy** because knowledge about the input is lost when the output is created (the model is also valid for lossless processes).

7.1 Types of Process Diagrams

Different diagrams of processes are useful for different purposes. The four we use here are all **recursive**, meaning that a process may be represented in terms of other, more detailed processes of the same sort, interconnected. Conversely, two or more connected processes may be represented by a single higher-level process with some of the detailed information suppressed. The processes represented can be either deterministic (noiseless) or nondeterministic (noisy), and either lossless or lossy.

- **Block Diagram:** Figure 7.1 (previous page) is a block diagram. It shows how the processes are connected, but very little about how the processes achieve their purposes, or how the connections are made. It is useful for viewing the system at a highly abstract level. An interconnection in a block diagram can represent many bits.
- **Circuit Diagram:** If the system is made of logic gates, a useful diagram is one showing such gates interconnected. For example, Figure 7.2 is an *AND* gate. Each input and output represents a wire with a single logic value, with, for example, a high voltage representing 1 and a low voltage 0. The number of possible bit patterns of a logic gate is greater than the number of physical wires; each wire could have two possible voltages, so for n-input gates there would be 2^n possible input states. Often, but not always, the components in logic circuits are deterministic.
- **Probability Diagram:** A process with n single-bit inputs and m single-bit outputs can be modeled by the probabilities relating the 2^n possible input bit patterns and the 2^m possible output patterns. For example, Figure 7.3 (next page) shows a gate with two inputs (four bit patterns) and one output. An example of such a gate is the *AND* gate, and its probability model is shown in Figure 7.4. Probability diagrams are discussed further in Section 7.2.
- **Information Diagram:** A diagram that shows explicitly the information flow between processes is useful. In order to handle processes with noise or loss, the information associated with them can be shown. Information diagrams are discussed further in Section 7.6.

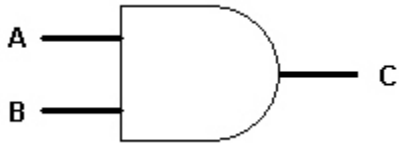


Figure 7.2: Circuit diagram of an *AND* gate

7.2 Probability Diagrams

The probability model of a process with n inputs and m outputs, where n and m are integers, is shown in Figure 7.5. The n input states are mutually exclusive, as are the m output states. If this process is implemented by logic gates the input would need at least $\log_2(n)$ but not as many as n wires.

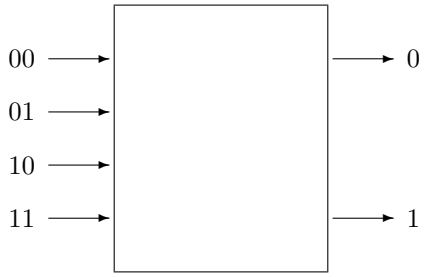


Figure 7.3: Probability model of a two-input one-output gate.

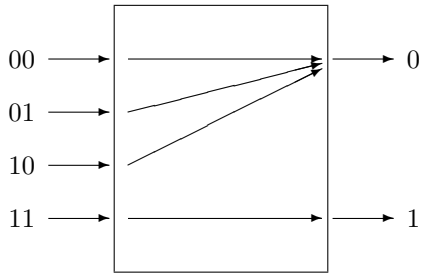


Figure 7.4: Probability model of an *AND* gate

This model for processes is conceptually simple and general. It works well for processes with a small number of bits. It was used for the binary channel in Chapter 6.

Unfortunately, the probability model is awkward when the number of input bits is moderate or large. The reason is that the inputs and outputs are represented in terms of mutually exclusive sets of events. If the events describe signals on, say, five wires, each of which can carry a high or low voltage signifying a boolean 1 or 0, there would be 32 possible events. It is much easier to draw a logic gate, with five inputs representing physical variables, than a probability process with 32 input states. This "exponential explosion" of the number of possible input states gets even more severe when the process represents the evolution of the state of a physical system with a large number of atoms. For example, the number of molecules in a mole of gas is Avogadro's number $N_A = 6.02 \times 10^{23}$. If each atom had just one associated boolean variable, there would be 2^{N_A} states, far greater than the number of particles in the universe. And there would not be time to even list all the particles, much less do any calculations: the number of microseconds since the big bang is less than 5×10^{23}. Despite this limitation, the probability diagram model is useful conceptually.

Let's review the fundamental ideas in communications, introduced in Chapter 6, in the context of such diagrams.

We assume that each possible input state of a process can lead to one or more output state. For each

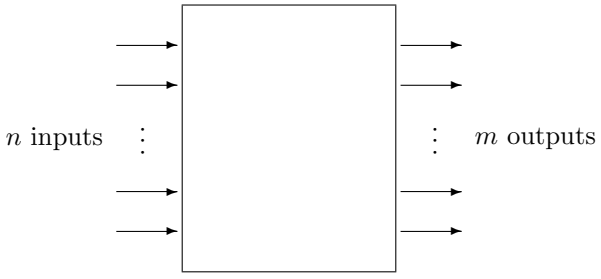


Figure 7.5: Probability model

input i denote the probability that this input leads to the output j as c_{ji}. These **transition probabilities** c_{ji} can be thought of as a table, or matrix, with as many columns as there are input states, and as many rows as output states. We will use i as an index over the input states and j over the output states, and denote the event associated with the selection of input i as A_i and the event associated with output j as B_j.

The transition probabilities are properties of the process, and do not depend on the inputs to the process. The transition probabilities lie between 0 and 1, and for each i their sum over the output index j is 1, since for each possible input event exactly one output event happens. If the number of input states is the same as the number of output states then c_{ji} is a square matrix; otherwise it has more columns than rows or vice versa.

$$0 \leq c_{ji} \leq 1 \tag{7.1}$$

$$1 = \sum_j c_{ji} \tag{7.2}$$

This description has great generality. It applies to a deterministic process (although it may not be the most convenient—a truth table giving the output for each of the inputs is usually simpler to think about). For such a process, each column of the c_{ji} matrix contains one element that is 1 and all the other elements are 0. It also applies to a nondeterministic channel (i.e., one with noise). It applies to the source encoder and decoder, to the compressor and expander, and to the channel encoder and decoder. It applies to logic gates and to devices which perform arbitrary memoryless computation (sometimes called "combinational logic" in distinction to "sequential logic" which can involve prior states). It even applies to transitions taken by a physical system from one of its states to the next. It applies if the number of output states is greater than the number of input states (for example channel encoders) or less (for example channel decoders).

If a process input is determined by random events A_i with probability distribution $p(A_i)$ then the various other probabilities can be calculated. The conditional output probabilities, conditioned on the input, are

$$p(B_j \mid A_i) = c_{ji} \tag{7.3}$$

The unconditional probability of each output $p(B_j)$ is

$$p(B_j) = \sum_i c_{ji} p(A_i) \tag{7.4}$$

Finally, the joint probability of each input with each output $p(A_i, B_j)$ and the backward conditional probabilities $p(A_i \mid B_j)$ can be found using Bayes' Theorem:

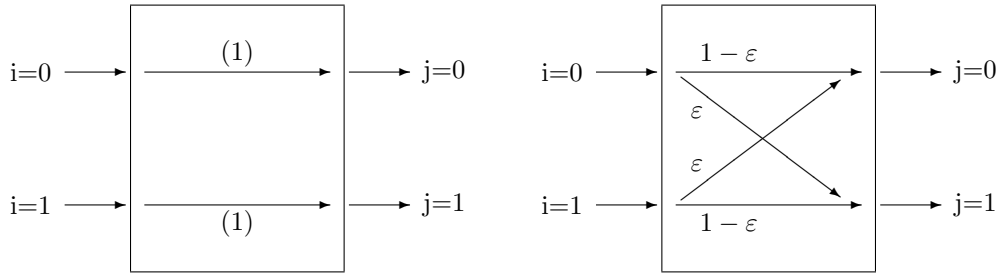
$$\begin{aligned} p(A_i, B_j) &= p(B_j)p(A_i \mid B_j) && (7.5)\\ &= p(A_i)p(B_j \mid A_i) && (7.6)\\ &= p(A_i)c_{ji} && (7.7) \end{aligned}$$

7.2.1 Example: *AND* Gate

The *AND* gate is deterministic (it has no noise) but is lossy, because knowledge of the output is not generally sufficient to infer the input. The transition matrix is

$$\begin{bmatrix} c_{0(00)} & c_{0(01)} & c_{0(10)} & c_{0(11)} \\ c_{1(00)} & c_{1(01)} & c_{1(10)} & c_{1(11)} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.8}$$

The probability model for this gate is shown in Figure 7.4.



(a) No errors (b) Symmetric errors

Figure 7.6: Probability models for error-free binary channel and symmetric binary channel

7.2.2 Example: Binary Channel

The binary channel is well described by the probability model. Its properties, many of which were discussed in Chapter 6, are summarized below.

Consider first a noiseless binary channel which, when presented with one of two possible input values 0 or 1, transmits this value faithfully to its output. This is a very simple example of a discrete memoryless process. We represent this channel by a probability model with two inputs and two outputs. To indicate the fact that the input is replicated faithfully at the output, the inner workings of the box are revealed, in Figure 7.6(a), in the form of two paths, one from each input to the corresponding output, and each labeled by the probability (1). The transition matrix for this channel is

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{7.9}$$

The input information I for this process is 1 bit if the two values are equally likely, or if $p(A_0) \neq p(A_1)$ the input information is

$$I = p(A_0) \log_2 \left(\frac{1}{p(A_0)} \right) + p(A_1) \log_2 \left(\frac{1}{p(A_1)} \right) \tag{7.10}$$

The output information J has a similar formula, using the output probabilities $p(B_0)$ and $p(B_1)$. Since the input and output are the same in this case, it is always possible to infer the input when the output has been observed. The amount of information out J is the same as the amount in I: $J = I$. This noiseless channel is effective for its intended purpose, which is to permit the receiver, at the output, to infer the value at the input.

Next, let us suppose that this channel occasionally makes errors. Thus if the input is 1 the output is not always 1, but with the "bit error probability" ε is flipped to the "wrong" value 0, and hence is "correct" only with probability $1 - \varepsilon$. Similarly, for the input of 0, the probability of error is ε. Then the transition matrix is

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} 1-\varepsilon & \varepsilon \\ \varepsilon & 1-\varepsilon \end{bmatrix} \tag{7.11}$$

This model, with random behavior, is sometimes called the Symmetric Binary Channel (SBC), symmetric in the sense that the errors in the two directions (from 0 to 1 and vice versa) are equally likely. The probability diagram for this channel is shown in Figure 7.6(b), with two paths leaving from each input, and two paths converging on each output.

Clearly the errors in the SBC introduce some uncertainty into the output over and above the uncertainty that is present in the input signal. Intuitively, we can say that noise has been added, so that the output is composed in part of desired signal and in part of noise. Or we can say that some of our information is lost in the channel. Both of these effects have happened, but as we will see they are not always related; it is

possible for processes to introduce noise but have no loss, or vice versa. In Section 7.3 we will calculate the amount of information lost or gained because of noise or loss, in bits.

Loss of information happens because it is no longer possible to tell with certainty what the input signal is, when the output is observed. Loss shows up in drawings like Figure 7.6(b) where two or more paths converge on the same output. Noise happens because the output is not determined precisely by the input. Noise shows up in drawings like Figure 7.6(b) where two or more paths diverge from the same input. Despite noise and loss, however, some information can be transmitted from the input to the output (i.e., observation of the output can allow one to make some inferences about the input).

We now return to our model of a general discrete memoryless nondeterministic lossy process, and derive formulas for noise, loss, and information transfer (which will be called "mutual information"). We will then come back to the symmetric binary channel and interpret these formulas.

7.3 Information, Loss, and Noise

For the general discrete memoryless process, useful measures of the amount of information presented at the input and the amount transmitted to the output can be defined. We suppose the process state is represented by random events A_i with probability distribution $p(A_i)$. The information at the input I is the same as the entropy of this source. (We have chosen to use the letter I for input information not because it stands for "input" or "information" but rather for the index i that goes over the input probability distribution. The output information will be denoted J for a similar reason.)

$$I = \sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) \tag{7.12}$$

This is the amount of uncertainty we have about the input if we do not know what it is, or before it has been selected by the source.

A similar formula applies at the output. The output information J can also be expressed in terms of the input probability distribution and the channel transition matrix:

$$\begin{aligned} J &= \sum_j p(B_j) \log_2 \left(\frac{1}{p(B_j)} \right) \\ &= \sum_j \left(\sum_i c_{ji} p(A_i) \right) \log_2 \left(\frac{1}{\sum_i c_{ji} p(A_i)} \right) \end{aligned} \tag{7.13}$$

Note that this measure of information at the output J refers to the identity of the output state, not the input state. It represents our uncertainty about the output state before we discover what it is. If our objective is to determine the input, J is not what we want. Instead, we should ask about the uncertainty of our knowledge of the input state. This can be expressed from the vantage point of the output by asking about the uncertainty of the input state given one particular output state, and then averaging over those states. This uncertainty, for each j, is given by a formula like those above but using the reverse conditional probabilities $p(A_i \mid B_j)$

$$\sum_i p(A_i \mid B_j) \log_2 \left(\frac{1}{p(A_i \mid B_j)} \right) \tag{7.14}$$

Then your average uncertainty about the input after learning the output is found by computing the average over the output probability distribution, i.e., by multiplying by $p(B_j)$ and summing over j

$$\begin{aligned} L &= \sum_j p(B_j) \sum_i p(A_i \mid B_j) \log_2 \left(\frac{1}{p(A_i \mid B_j)} \right) \\ &= \sum_{ij} p(A_i, B_j) \log_2 \left(\frac{1}{p(A_i \mid B_j)} \right) \end{aligned} \tag{7.15}$$

Note that the second formula uses the joint probability distribution $p(A_i, B_j)$. We have denoted this average uncertainty by L and will call it "loss." This term is appropriate because it is the amount of information about the input that is not able to be determined by examining the output state; in this sense it got "lost" in the transition from input to output. In the special case that the process allows the input state to be identified uniquely for each possible output state, the process is "lossless" and, as you would expect, $L = 0$.

It was proved in Chapter 6 that $L \leq I$ or, in words, that the uncertainty after learning the output is less than (or perhaps equal to) the uncertainty before. This result was proved using the Gibbs inequality.

The amount of information we learn about the input state upon being told the output state is our uncertainty before being told, which is I, less our uncertainty after being told, which is L. We have just shown that this amount cannot be negative, since $L \leq I$. As was done in Chapter 6, we denote the amount we have learned as $M = I - L$, and call this the "mutual information" between input and output. This is an important quantity because it is the amount of information that gets through the process.

To recapitulate the relations among these information quantities:

$$I = \sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) \tag{7.16}$$

$$L = \sum_j p(B_j) \sum_i p(A_i \mid B_j) \log_2 \left(\frac{1}{p(A_i \mid B_j)} \right) \tag{7.17}$$

$$M = I - L \tag{7.18}$$

$$0 \leq M \leq I \tag{7.19}$$

$$0 \leq L \leq I \tag{7.20}$$

Processes with outputs that can be produced by more than one input have loss. These processes may also be nondeterministic, in the sense that one input state can lead to more than one output state. The symmetric binary channel with loss is an example of a process that has loss and is also nondeterministic. However, there are some processes that have loss but are deterministic. An example is the *AND* logic gate, which has four mutually exclusive inputs 00 01 10 11 and two outputs 0 and 1. Three of the four inputs lead to the output 0. This gate has loss but is perfectly deterministic because each input state leads to exactly one output state. The fact that there is loss means that the *AND* gate is not reversible.

There is a quantity similar to L that characterizes a nondeterministic process, whether or not it has loss. The output of a nondeterministic process contains variations that cannot be predicted from knowing the input, that behave like noise in audio systems. We will define the noise N of a process as the uncertainty in the output, given the input state, averaged over all input states. It is very similar to the definition of loss, but with the roles of input and output reversed. Thus

$$\begin{aligned} N &= \sum_i p(A_i) \sum_j p(B_j \mid A_i) \log_2 \left(\frac{1}{p(B_j \mid A_i)} \right) \\ &= \sum_i p(A_i) \sum_j c_{ji} \log_2 \left(\frac{1}{c_{ji}} \right) \end{aligned} \tag{7.21}$$

Steps similar to those above for loss show analogous results. What may not be obvious, but can be proven easily, is that the mutual information M plays exactly the same sort of role for noise as it does for loss. The formulas relating noise to other information measures are like those for loss above, where the mutual information M is the same:

$$J = \sum_i p(B_j) \log_2 \left(\frac{1}{p(B_j)} \right) \tag{7.22}$$

$$N = \sum_i p(A_i) \sum_j c_{ji} \log_2 \left(\frac{1}{c_{ji}} \right) \tag{7.23}$$

$$M = J - N \tag{7.24}$$

$$0 \leq M \leq J \tag{7.25}$$

$$0 \leq N \leq J \tag{7.26}$$

It follows from these results that

$$J - I = N - L \tag{7.27}$$

7.3.1 Example: Symmetric Binary Channel

For the SBC with bit error probability ε, these formulas can be evaluated, even if the two input probabilities $p(A_0)$ and $p(A_1)$ are not equal. If they happen to be equal (each 0.5), then the various information measures for the SBC in bits are particularly simple:

$$I = 1 \text{ bit} \tag{7.28}$$

$$J = 1 \text{ bit} \tag{7.29}$$

$$L = N = \varepsilon \log_2 \left(\frac{1}{\varepsilon} \right) + (1 - \varepsilon) \log_2 \left(\frac{1}{1 - \varepsilon} \right) \tag{7.30}$$

$$M = 1 - \varepsilon \log_2 \left(\frac{1}{\varepsilon} \right) - (1 - \varepsilon) \log_2 \left(\frac{1}{1 - \varepsilon} \right) \tag{7.31}$$

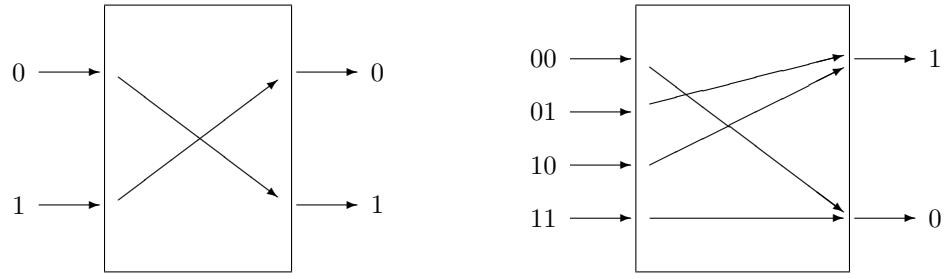
The errors in the channel have destroyed some of the information, in the sense that they have prevented an observer at the output from knowing with certainty what the input is. They have thereby permitted only the amount of information $M = I - L$ to be passed through the channel to the output.

7.4 Deterministic Examples

This probability model applies to any system with mutually exclusive inputs and outputs, whether or not the transitions are random. If all the transition probabilities c_{ji} are equal to either 0 or 1, then the process is deterministic.

A simple example of a deterministic process is the NOT gate, which implements Boolean negation. If the input is 1 the output is 0 and vice versa. The input and output information are the same, $I = J$ and there is no noise or loss: $N = L = 0$. The information that gets through the gate is $M = I$. See Figure 7.7(a).

A slightly more complex deterministic process is the exclusive or, XOR gate. This is a Boolean function of two input variables and therefore there are four possible input values. When the gate is represented by



(a) *NOT* gate (b) *XOR* gate

Figure 7.7: Probability models of deterministic gates

a circuit diagram, there are two input wires representing the two inputs. When the gate is represented as a discrete process using a probability diagram like Figure 7.7(b), there are four mutually exclusive inputs and two mutually exclusive outputs. If the probabilities of the four inputs are each 0.25, then $I = 2$ bits, and the two output probabilities are each 0.5 so $J = 1$ bit. There is therefore 1 bit of loss, and the mutual information is 1 bit. The loss arises from the fact that two different inputs produce the same output; for example if the output 1 is observed the input could be either 01 or 10. There is no noise introduced into the output because each of the transition parameters is either 0 or 1, i.e., there are no inputs with multiple transition paths coming from them.

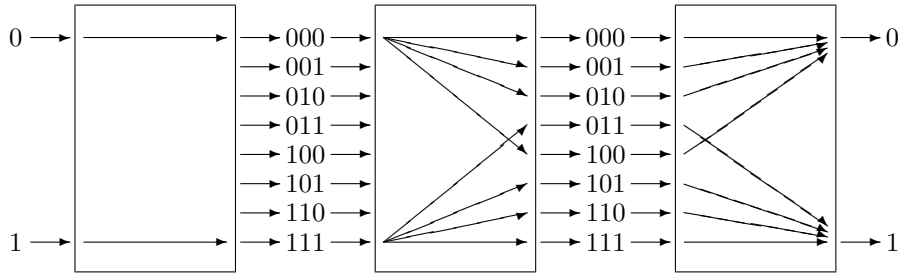
Other, more complex logic functions can be represented in similar ways. However, for logic functions with n physical inputs, a probability diagram is awkward if n is larger than 3 or 4 because the number of inputs is 2^n.

7.4.1 Error Correcting Example

The Hamming Code encoder and decoder can be represented as discrete processes in this form. Consider the (3, 1, 3) code, otherwise known as triple redundancy. The encoder has one 1-bit input (2 values) and a 3-bit output (8 values). The input 1 is wired directly to the output 111 and the input 0 to the output 000. The other six outputs are not connected, and therefore occur with probability 0. See Figure 7.8(a). The encoder has $N = 0$, $L = 0$, and $M = I = J$. Note that the output information is not three bits even though three physical bits are used to represent it, because of the intentional redundancy.

The output of the triple redundancy encoder is intended to be passed through a channel with the possibility of a single bit error in each block of 3 bits. This noisy channel can be modelled as a nondeterministic process with 8 inputs and 8 outputs, Figure 7.8(b). Each of the 8 inputs is connected with a (presumably) high-probability connection to the corresponding output, and with low probability connections to the three other values separated by Hamming distance 1. For example, the input 000 is connected only to the outputs 000 (with high probability) and 001, 010, and 100 each with low probability. This channel introduces noise since there are multiple paths coming from each input. In general, when driven with arbitrary bit patterns, there is also loss. However, when driven from the encoder of Figure 7.8(a), the loss is 0 bits because only two of the eight bit patterns have nonzero probability. The input information to the noisy channel is 1 bit and the output information is greater than 1 bit because of the added noise. This example demonstrates that the value of both noise and loss depend on both the physics of the channel and the probabilities of the input signal.

The decoder, used to recover the signal originally put into the encoder, is shown in Figure 7.8(c). The transition parameters are straightforward—each input is connected to only one output. The decoder has loss (since multiple paths converge on each of the ouputs) but no noise (since each input goes to only one output).



(a) Encoder (b) Channel (c) Decoder

Figure 7.8: Triple redundancy error correction

7.5 Capacity

In Chapter 6 of these notes, the channel capacity was defined. This concept can be generalized to other processes.

Call W the maximum rate at which the input state of the process can be detected at the output. Then the rate at which information flows through the process can be as large as WM. However, this product depends on the input probability distribution $p(A_i)$ and hence is not a property of the process itself, but on how it is used. A better definition of process capacity is found by looking at how M can vary with different input probability distributions. Select the largest mutual information for any input probability distribution, and call that M_{max}. Then the **process capacity** C is defined as

$$C = WM_{max} \tag{7.32}$$

It is easy to see that M_{max} cannot be arbitrarily large, since $M \leq I$ and $I \leq \log_2 n$ where n is the number of distinct input states.

In the example of symmetric binary channels, it is not difficult to show that the probability distribution that maximizes M is the one with equal probability for each of the two input states.

7.6 Information Diagrams

An information diagram is a representation of one or more processes explicitly showing the amount of information passing among them. It is a useful way of representing the input, output, and mutual information and the noise and loss. Information diagrams are at a high level of abstraction and do not display the detailed probabilities that give rise to these information measures.

It has been shown that all five information measures, I, J, L, N, and M are nonnegative. It is not necessary that L and N be the same, although they are for the symmetric binary channel whose inputs have equal probabilities for 0 and 1. It is possible to have processes with loss but no noise (e.g., the XOR gate), or noise but no loss (e.g., the noisy channel for triple redundancy).

It is convenient to think of information as a physical quantity that is transmitted through this process much the way physical material may be processed in a production line. The material being produced comes in to the manufacturing area, and some is lost due to errors or other causes, some contamination may be added (like noise) and the output quantity is the input quantity, less the loss, plus the noise. The useful product is the input minus the loss, or alternatively the output minus the noise. The flow of information through a discrete memoryless process is shown using this paradigm in Figure 7.9.

An interesting question arises. Probabilities depend on your current state of knowledge, and one observer's knowledge may be different from another's. This means that the loss, the noise, and the information transmitted are all observer-dependent. Is it OK that important engineering quantities like noise and loss depend on who you are and what you know? If you happen to know something about the input that

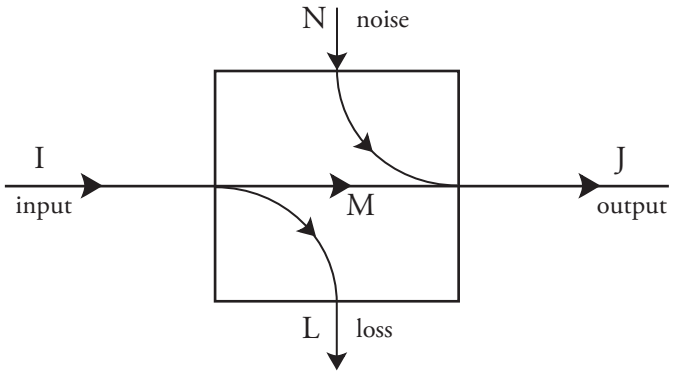


Figure 7.9: Information flow in a discrete memoryless process

your colleague does not, is it OK for your design of a nondeterministic process to be different, and to take advantage of your knowledge? This question is something to think about; there are times when your knowledge, if correct, can be very valuable in simplifying designs, but there are other times when it is prudent to design using some worst-case assumption of input probabilities so that in case the input does not conform to your assumed probabilities your design still works.

Information diagrams are not often used for communication systems. There is usually no need to account for the noise sources or what happens to the lost information. However, such diagrams are useful in domains where noise and loss cannot occur. One example is reversible computing, a style of computation in which the entire process can, in principle, be run backwards. Another example is quantum communications, where information cannot be discarded without affecting the environment.

7.6.1 Notation

Different authors use different notation for the quantities we have here called I, J, L, N, and M. In his original paper Shannon called the input probability distribution x and the output distribution y. The input information I was denoted $H(x)$ and the output information J was $H(y)$. The loss L (which Shannon called "equivocation") was denoted $H_y(x)$ and the noise N was denoted $H_x(y)$. The mutual information M was denoted R. Shannon used the word "entropy" to refer to information, and most authors have followed his lead.

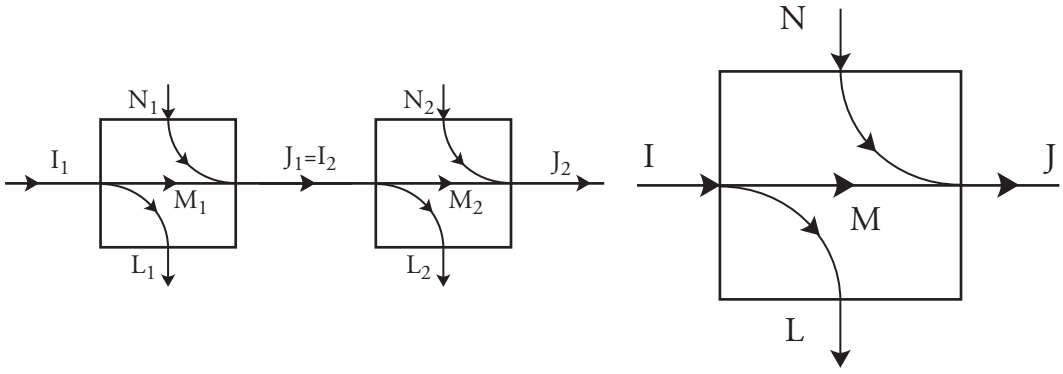
Frequently information quantities are denoted by I, H, or S, often as functions of probability distributions, or "ensembles." In physics entropy is often denoted S.

Another common notation is to use A to stand for the input probability distribution, or ensemble, and B to stand for the output probability distribution. Then I is denoted $I(A)$, J is $I(B)$, L is $I(A|B)$, N is $I(B|A)$, and M is $I(A;B)$. If there is a need for the information associated with A and B jointly (as opposed to conditionally) it can be denoted $I(A,B)$ or $I(AB)$.

7.7 Cascaded Processes

Consider two processes in **cascade**. This term refers to having the output from one process serve as the input to another process. Then the two cascaded processes can be modeled as one larger process, if the "internal" states are hidden. We have seen that discrete memoryless processes are characterized by values of I, J, L, N, and M. Figure 7.10(a) shows a cascaded pair of processes, each characterized by its own parameters. Of course the parameters of the second process depend on the input probabilities it encounters, which are determined by the transition probabilities (and input probabilities) of the first process.

But the cascade of the two processes is itself a discrete memoryless process and therefore should have its own five parameters, as suggested in Figure 7.10(b). The parameters of the overall model can be calculated



(a) The two processes (b) Equivalent single process

Figure 7.10: Cascade of two discrete memoryless processes

either of two ways. First, the transition probabilities of the overall process can be found from the transition probabilities of the two models that are connected together; in fact the matrix of transition probabilities is merely the matrix product of the two transition probability matrices for process 1 and process 2. All the parameters can be calculated from this matrix and the input probabilities.

The other approach is to seek formulas for I, J, L, N, and M of the overall process in terms of the corresponding quantities for the component processes. This is trivial for the input and output quantities: $I = I_1$ and $J = J_2$. However, it is more difficult for L and N. Even though L and N cannot generally be found exactly from L_1, L_2, N_1 and N_2, it is possible to find upper and lower bounds for them. These are useful in providing insight into the operation of the cascade.

It can be easily shown that since $I = I_1$, $J_1 = I_2$, and $J = J_2$,

$$L - N = (L_1 + L_2) - (N_1 + N_2) \tag{7.33}$$

It is then straightforward (though perhaps tedious) to show that the loss L for the overall process is not always equal to the sum of the losses for the two components $L_1 + L_2$, but instead

$$0 \leq L_1 \leq L \leq L_1 + L_2 \tag{7.34}$$

so that the loss is bounded from above and below. Also,

$$L_1 + L_2 - N_1 \leq L \leq L_1 + L_2 \tag{7.35}$$

so that if the first process is noise-free then L is exactly $L_1 + L_2$.

There are similar formulas for N in terms of $N_1 + N_2$:

$$0 \leq N_2 \leq N \leq N_1 + N_2 \tag{7.36}$$

$$N_1 + N_2 - L_2 \leq N \leq N_1 + N_2 \tag{7.37}$$

Similar formulas for the mutual information of the cascade M follow from these results:

$$M_1 - L_2 \leq M \leq M_1 \leq I \tag{7.38}$$
$$M_1 - L_2 \leq M \leq M_1 + N_1 - L_2 \tag{7.39}$$
$$M_2 - N_1 \leq M \leq M_2 \leq J \tag{7.40}$$
$$M_2 - N_1 \leq M \leq M_2 + L_2 - N_1 \tag{7.41}$$

Other formulas for M are easily derived from Equation 7.19 applied to the first process and the cascade, and Equation 7.24 applied to the second process and the cascade:

$$\begin{aligned} M &= M_1 + L_1 - L \\ &= M_1 + N_1 + N_2 - N - L_2 \\ &= M_2 + N_2 - N \\ &= M_2 + L_2 + L_1 - L - N_1 \end{aligned} \tag{7.42}$$

where the second formula in each case comes from the use of Equation 7.33.

Note that M cannot exceed either M_1 or M_2, i.e., $M \leq M_1$ and $M \leq M_2$. This is consistent with the interpretation of M as the information that gets through—information that gets through the cascade must be able to get through the first process and also through the second process.

As a special case, if the second process is lossless, $L_2 = 0$ and then $M = M_1$. In that case, the second process does not lower the mutual information below that of the first process. Similarly if the first process is noiseless, then $N_1 = 0$ and $M = M_2$.

The channel capacity C of the cascade is, similarly, no greater than either the channel capacity of the first process or that of the second process: $C \leq C_1$ and $C \leq C_2$. Other results relating the channel capacities are not a trivial consequence of the formulas above because C is by definition the maximum M over all possible input probability distributions—the distribution that maximizes M_1 may not lead to the probability distribution for the input of the second process that maximizes M_2.

Chapter 8

Inference

In Chapter 7 the process model was introduced as a way of accounting for flow of information through processes that are discrete, finite, and memoryless, and which may be nondeterministic and lossy. Although the model was motivated by the way many communication systems work, it is more general.

Formulas were given for input information I, loss L, mutual information M, noise N, and output information J. Each of these is measured in bits, although in a setting in which many symbols are chosen, one after another, they may be multiplied by the rate of symbol selection and then expressed in bits per second. The information flow is shown in Figure 8.1. All these quantities depend on the input probability distribution $p(A_i)$.

If the input probabilities are already known, and a particular output outcome is observed, it is possible to make inferences about the input event that led to that outcome. Sometimes the input event can be identified with certainty, but more often the inferences are in the form of changes in the initial input probabilities. This is typically how communication systems work—the output is observed and the "most likely" input event is inferred. Inference in this context is sometime referred to as **estimation**. It is the topic of Section 8.1.

On the other hand, if the input probabilities are not known, this approach does not work. We need a way to get the initial probability distribution. An approach that is based on the information analysis is discussed in Section 8.2 and in subsequent chapters of these notes. This is the Principle of Maximum Entropy.

8.1 Estimation

It is often necessary to determine the input event when only the output event has been observed. This is the case for communication systems, in which the objective is to infer the symbol emitted by the source so that it can be reproduced at the output. It is also the case for memory systems, in which the objective is to recreate the original bit pattern without error.

In principle, this estimation is straightforward if the input probability distribution $p(A_i)$ and the conditional output probabilities, conditioned on the input events, $p(B_j \mid A_i) = c_{ji}$, are known. These "forward" conditional probabilities c_{ji} form a matrix with as many rows as there are output events, and as many columns as there are input events. They are a property of the process, and do not depend on the input probabilities $p(A_i)$.

The unconditional probability $p(B_j)$ of each output event B_j is

$$p(B_j) = \sum_i c_{ji} p(A_i) \tag{8.1}$$

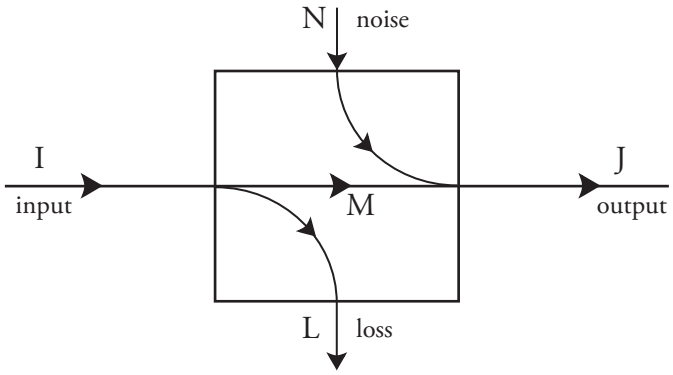


Figure 8.1: Information flow in a discrete memoryless process

and the joint probability of each input with each output $p(A_i, B_j)$ and the backward conditional probabilities $p(A_i \mid B_j)$ can be found using Bayes' Theorem:

$$\begin{aligned} p(A_i, B_j) &= p(B_j)p(A_i \mid B_j) \\ &= p(A_i)p(B_j \mid A_i) \\ &= p(A_i)c_{ji} \end{aligned} \tag{8.2}$$

Now let us suppose that a particular output event B_j has been observed. The input event that "caused" this output can be estimated only to the extent of giving a probability distribution over the input events. For each input event A_i the probability that it was the input is simply the backward conditional probability $p(A_i \mid B_j)$ for the particular output event B_j, which can be written using Equation 8.2 as

$$p(A_i \mid B_j) = \frac{p(A_i)c_{ji}}{p(B_j)} \tag{8.3}$$

If the process has no loss ($L = 0$) then for each j exactly one of the input events A_i has nonzero probability, and therefore its probability $p(A_i \mid B_j)$ is 1. In the more general case, with nonzero loss, estimation consists of refining a set of input probabilities so they are consistent with the known output. Note that this approach only works if the original input probability distribution is known. All it does is refine that distribution in the light of new knowledge, namely the observed output.

It might be thought that the new input probability distribution would have less uncertainty than that of the original distribution. Is this always true?

The uncertainty of a probability distribution is, of course, its entropy as defined earlier. The uncertainty (about the input event) before the output event is known is

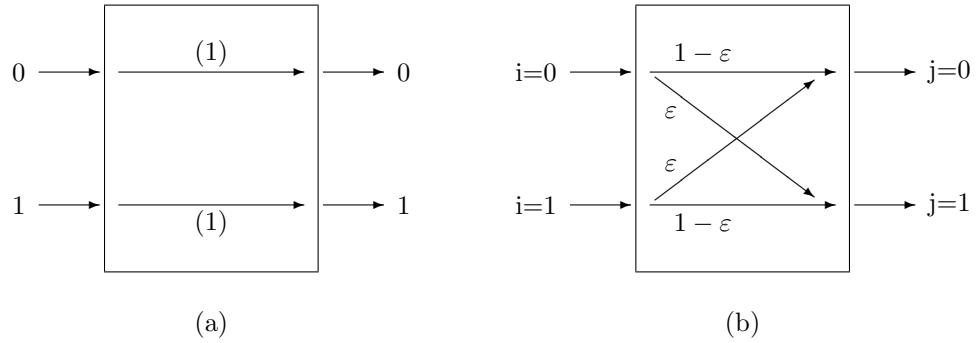
$$U_{\text{before}} = \sum_i p(A_i) \log_2 \left(\frac{1}{p(A_i)} \right) \tag{8.4}$$

The residual uncertainty, after some particular output event is known, is

$$U_{\text{after}}(B_j) = \sum_i p(A_i \mid B_j) \log_2 \left(\frac{1}{p(A_i \mid B_j)} \right) \tag{8.5}$$

The question, then is whether $U_{\text{after}}(B_j) \leq U_{\text{before}}$. The answer is often, but not always, yes. However, it is not difficult to prove that the average (over all output states) of the residual uncertainty is less than the original uncertainty:

$$\sum_j p(B_j) U_{\text{after}}(B_j) \leq U_{\text{before}} \tag{8.6}$$



(a) (b)

Figure 8.2: (a) Binary Channel without noise (b) Symmetric Binary Channel, with errors

In words, this statement says that **on average**, our uncertainty about the input state is never increased by learning something about the output state. In other words, on average, this technique of inference helps us get a better estimate of the input state.

Two of the following examples will be continued in subsequent chapters including the next chapter on the Principle of Maximum Entropy—the symmetric binary channel and Berger's Burgers.

8.1.1 Symmetric Binary Channel

The noiseless, lossless binary channel shown in Figure 8.2(a) is a process with two input values which may be called 0 and 1, two output values similarly named, and a transition matrix c_{ji} which guarantees that the output equals the input:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{8.7}$$

This channel has no loss and no noise, and the mutual information, input information, and output information are all the same.

The symmetric binary channel (Figure 8.2(b)) is similar, but occasionally makes errors. Thus if the input is 1 the output is not always 1, but with the "bit error probability" ε is flipped to the "wrong" value 0, and hence is "correct" only with probability $1 - \varepsilon$. Similarly, for the input of 0, the probability of error is ε. Then the transition matrix is

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} 1-\varepsilon & \varepsilon \\ \varepsilon & 1-\varepsilon \end{bmatrix} \tag{8.8}$$

This channel is symmetric in the sense that the errors in both directions (from 0 to 1 and vice versa) are equally likely.

Because of the loss, the input event associated with, say, output event B_0 cannot be determined with certainty. Nevertheless, the formulas above can be used. In the important case where the two input probabilities are equal (and therefore each equals 0.5) an output of 0 implies that the input event A_0 has probability $1-\varepsilon$ and input event A_1 has probability ε. Thus if, as would be expected in a channel designed for low-error communication, ε is small, then it would be reasonable to infer that the input that produced the output event B_0 was the event A_0.

8.1.2 Non-symmetric Binary Channel

A non-symmetric binary channel is one in which the error probabilities for inputs 0 and 1 are different, i.e., $c_{01} \neq c_{10}$. We illustrate non-symmetric binary channels with an extreme case based on a medical test for Huntington's Disease.

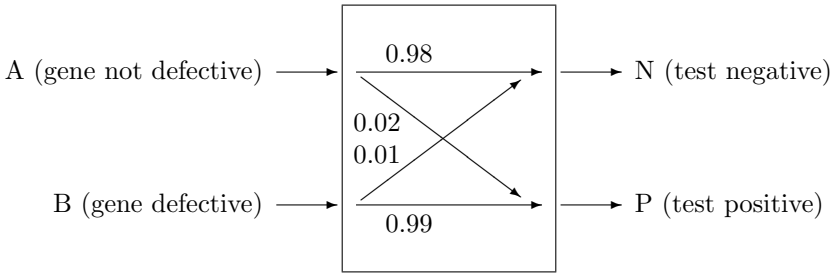


Figure 8.3: Huntington's Disease Test

Huntington's Disease is a rare, progressive, hereditary brain disorder with no known cure. It was named after Dr. George Huntington (1850–1915), a Long Island physician who published a description in 1872. It is caused by a defective gene, which was identified in 1983. Perhaps the most famous person afflicted with the disease was the songwriter Woodie Guthrie.

The biological child of a person who carries the defective gene has a 50% chance of inheriting the defective gene. For the population as a whole, the probability of carrying the defective gene is much lower. According to the Huntington's Disease Society of America, http://www.hdsa.org, "More than a quarter of a million Americans have HD or are 'at risk' of inheriting the disease from an affected parent." This is about 1/1000 of the population, so for a person selected at random, with an unknown family history, it is reasonable to estimate the probability of carrying the affected gene at 1/2000.

People carrying the defective gene all eventually develop the disease unless they die of another cause first. The symptoms most often appear in middle age, in people in their 40's or 50's, perhaps after they have already produced a family and thereby possibly transmitted the defective gene to another generation. Although the disease is not fatal, those in advanced stages generally die from its complications. Until recently, people with a family history of the disease were faced with a life of uncertainty, not knowing whether they carried the defective gene, and not knowing how to manage their personal and professional lives.

In 1993 a test was developed which can tell if you carry the defective gene. Unfortunately the test is not perfect; there is a probability of a false positive (reporting you have it when you actually do not) and of a false negative (reporting your gene is not defective when it actually is). For our purposes we will assume that the test only gives a yes/no answer, and that the probability of a false positive is 2% and the probability of a false negative is 1%. (The real test is actually better—it also estimates the severity of the defect, which is correlated with the age at which the symptoms start.)

If you take the test and learn the outcome, you would of course like to infer whether you will develop the disease eventually. The techniques developed above can help.

Let us model the test as a discrete memoryless process, with input A (no defective gene) and B (defective gene), and outputs P (positive) and N (negative). The process, shown in Figure 8.3, is not a symmetric binary channel, because the two error probabilities are not equal.

First, consider the application of this test to someone with a family history, for which $p(A) = p(B) = 0.5$. Then, if the test is negative, the probability, for that person, of having the defect is $1/99 = 0.0101$ and the probability of not having it is $98/99 = 0.9899$. On the other hand, if the test is positive, the probability, for that person, of carrying the defective gene is $99/101 = 0.9802$ and the probability of not doing so is $2/101 = 0.0198$. The test is very effective, in that the two outputs imply, to high probability, different inputs.

An interesting question that is raised by the existence of this test but is not addressed by our mathematical model is whether a person with a family history would elect to take the test, or whether he or she would prefer to live not knowing what the future holds in store. The development of the test was funded by a group including Guthrie's widow and headed by a man named Milton Wexler (1908–2007), who was concerned about his daughters because his wife and her brothers all had the disease. Wexler's daughters, whose situation inspired the development of the test, decided not to take it.