Information, Compression and Cryptography

Tom Davis
tomrdavis@earthlink.net
http://www.geometer.org/mathcircles
September 10, 2013

1 Introduction

This article concerns the process of communication, either open or secret, and it turns out that both modes have a lot in common. In the first part we'll look at properties of information in general, including questions like:

- What is information?
- How dense is the information?
- How can information be compressed for more efficient transmission?
- How can we check for transmission errors?
- How can we automatically correct transmission errors?

In the following sections we will assume that data can be transmitted without error, but how can that data be transmitted secretly so that only certain persons have the ability to read it?

In almost everything that follows we will assume that the information, secret or not, is transmitted via a message composed of symbols and not by face-to-face talking where it is easy to interrupt the other person and ask for clarification. When the information is supposed to be secret, we'll consider different levels of secrecy and different resources the transmitter and receiver have: does the encoding have to be done by hand, or does each of the people communicating have a supercomputer available?

Finally, when we consider transmission that is supposed to be secret, we'll consider the problem of how to attack the secrecy. If you are protecting information, you need to know how your opponents are going to try to read it, and if your opponents are engaging in secret communication, what strategies might you use to figure out what they're talking about, presumably without letting them know that you can listen in.

2 Information

In this section we're going to talk about information in general, and when we do so, we'll try to consider information in the most general possible sense. For communication, it will usually consist of transmitted messages with the idea being that after the transmission, the recipient has more information than before. But we can also consider information to be the results of an experiment or test: Does a manufactured device work correctly or not? Or information can simply be the result of a random event: Does a flipped coin turn up heads or tails? On what number does a roulette wheel stop?

For our first example consider a transmitted message that consists of a sequence of bits, each being either 0 or 1. At first, we will also assume that there are no errors in the transmission so that the recipient knows for sure that they have received exactly the same message that was sent.

For a message of length n bits, how any different messages can be transferred?

If the message consists of a single bit, there are only two possible messages: 0 and 1. With two bits, there are four messages: 00, 01, 10 and 11. When we add a third bit, there are twice as many as before since each of the four messages made with two bits can be terminated with a 0 or a 1 to make eight total messages. This same reasoning can be extended: every additional bit in a message doubles the number of available messages, so the total number of different messages with n bits is 2^n.

The number 2^n grows very rapidly with n, with 10 bits representing more than 1000 messages and 20 bits, more than a million. With modern computers and high-speed transmission, for most messages it is not too important if the transmission is inefficient, and a fairly inefficient method is usually used to transmit text messages; namely ASCII (see Section 4).

How do these bits correspond to a reasonable idea of "information"?

Just because a message has a lot of bits of data does not mean that it contains a lot of information. Consider the following questionnaire, where each answer is either "yes" or "no", so each effectively consists of one bit:

- Are you the president of the United States?
- Have you won a Nobel Prize?
- Are you more than 110 years old?

If I tell you that somebody answered "no" to all three questions, does that give you much information? Of course not, since that's the result you'd expect from almost anyone in the world. On the other hand, any "yes" answer in the list would be extremely informative: it would narrow down the person to be in a very small class of people.[1]

So a message contains more information depending on how unlikely it is to occur.

2.1 Uncertainty and Entropy

Imagine a situation where there are a fixed number of possible outcomes, and the probability of each is known. Entropy is a way to measure the uncertainty of the situation. If you are certain of the outcome, we'd like this measure to be zero, and the more uncertain the outcome, the larger we'd like the entropy to be. An "outcome" can be the result of an experiment, the outcome of a random event, like flipping a coin, or, usually in our case, the particular message that is transmitted to you.

What we would like to have is some sort of a mathematical expression that indicates, in a reasonable way, the uncertainty of a situation. You can think of uncertainly as the opposite of certainty. In other words, if you are almost certain how an experiment will come out, you will have low uncertainty, and vice-versa: a situation where is is very hard to predict an outcome will have high uncertainty. We'll be looking for a measure where if you are absolutely certain of what's going to happen, the uncertainty will be zero, and as you look at situations whose outcomes are harder and harder to predict, you should have uncertainties that are larger and larger.

For definiteness, let's assume the following situation: There are n possible outcomes, and we'll call them $x_1, x_2, x_3, \dots, x_n$. We will use the notation $p(x_i)$ to indicate the probability that the outcome x_i occurs. Of course we want the list of the x_i to be complete, so we need to have:

$$\sum_{i=1}^{n} p(x_i) = p(x_1) + p(x_2) + p(x_3) + \cdots + p(x_n) = 1.$$

The names of the events (the x_i) don't really matter; all that matters are the values of the $p(x_i)$ which we will, from now on, call p_i. Our goal will be to find a function $H(p_1, p_2, \dots, p_n)$ that measures the uncertainty of that situation.

Exercise: Try to think of some properties that such an uncertainty function H should have.

[1]It is somewhat amusing that the first time I led a math circle on this topic it was for students at the University of California, Berkeley campus, where if I *had* gone out of the building and asked a random person if they had a Nobel prize, the odds of getting a "yes" were probably a thousand times higher than they would be in most places. Of course, even there, the probability of finding a Nobel prize winner would have been tiny.

2.1.1 Properties of H

Here are some obvious properties (and perhaps some not-so-obvious ones) that an uncertainty-measuring function should have:

1. If there is no uncertainty; namely, if there is only one possible outcome, then the value of H should be 0. The uncertainty of any event should be non-negative, and if situation A is less-predictable than situation B, the uncertainty of A should be larger than that of B.

2. The order of the parameters should make no difference. That is, it shouldn't matter which outcome the name as the first, second, third, et cetera. Mathematically, it means:
$$H(p_1, p_2) = H(p_2, p_1)$$
for two possible outcomes, and
$$H(p_1, p_2, p_3) = H(p_1, p_3, p_2) = H(p_2, p_1, p_3) = H(p_2, p_3, p_1) = H(p_3, p_1, p_2) = H(p_3, p_2, p_1)$$
for three outcomes, et cetera. Mathematical functions where the order of the parameters is unimportant are called *symmetric functions*. Another way of stating this idea is to say that $H(p_1, p_2, \ldots, p_n)$ is symmetric. We can mathematically state this for the general case as follows: Let $\pi(i)$ be a permutation (a rearrangement) of the set $\{1, 2, 3, \ldots, n\}$. Then for any such π we have:
$$H(p_1, p_2, p_3, \ldots, p_n) = H(p_{\pi(1)}, p_{\pi(2)}, p_{\pi(3)}, \ldots, p_{\pi(n)}).$$

3. When there are n possible outcomes, the situation that is the most uncertain is when all the p_i are equal. If it's more likely that one outcome will occur than another, then there is less uncertainty. If two equally-able football teams are playing, you are quite uncertain about the outcome, but if the San Francisco 49ers are playing against your high-school football team, there is very little uncertainly about what will happen.

   We can state this mathematically as follows: For any set of $p_i \geq 0$ such that $p_1 + p_2 + \cdots + p_n = 1$ then:
$$H(1/n, 1/n, 1/n, \ldots, 1/n) \geq H(p_1, p_2, p_3, \ldots, p_n).$$

4. Similarly, if you have more equally-likely outcomes, the uncertainty increases:
$$H(1/n, 1/n, \ldots, 1/n) \leq H(1/(n+1), 1/(n+1), \ldots, 1/(n+1)),$$
(where the first H has n parameters and the second, $n + 1$).

5. Adding an outcome to the list of possible outcomes which has zero probability of occurring will not change the value of H. In other words, it will have no effect on the uncertainty. Mathematically:
$$H(p_1, p_2, \ldots, p_n) = H(p_1, p_2, \ldots, p_n, 0),$$
(where the first H has n parameters and the second, $n + 1$).

6. If there are two completely independent situations, the first with outcomes having probabilities $p_1, p_2, \ldots, p_n$ and the other having probabilities $q_1, q_2, \ldots, q_m$ then when both situations have occurred, there are mn possible outcomes, and since the situations are independent, the probabilities of these mn outcomes will be:
$$p_1q_1, p_1q_2, \ldots p_1q_m, p_2q_1, p_2q_2, \ldots, p_2q_m, \ldots, p_nq_1, \ldots p_nq_m.$$
The total uncertainty should be the sum of the two uncertainties, so:
$$H(p_1q_1, p_1q_2, \ldots, p_nq_n) = H(p1, p_2, \ldots, p_n) + H(q_1, q_2, \ldots, q_m).$$

7. H should be continuous in its variables. In other words, if one set of probabilities is very close to another set, then the values of H for those two sets should also be close. (The standard calculus definition of continuity applies, if you know what that is.)

8. Here is a final condition that's a little more complicated. First we'll state the result mathematically and then try to describe it.

   Suppose $p_i \geq 0$, $q_i \geq 0$, $p = p_1 + \ldots + p_n$, $q = q_1 + \ldots q_m$, and $p + q = 1$. Then:

   $$H(p_1, \ldots, p_n, q_1, \ldots, q_m) = H(p, q) + H(p_1/p, p_2/p, \ldots p_n/p) + H(q_1/q, q_2/q, \ldots, q_m/q).$$

   In English, this basically says that if a situation with $n + m$ outcomes is divided into two classes, one having total probability p and the other, total probability q, then the total uncertainty is the sum of three things: the uncertainty of whether the first or the second class occurs, the uncertainly of the result if we were restricted to only the first class and the uncertainly of the result if we were restricted to only the second class. This is a little technical, so don't worry if it is not at first obvious.

2.1.2 Logarithms

If you know what a logarithm is, all you really need to know to continue is that in information theory, it is almost always best to use base-2 logarithms, and you can now skip to the next section, keeping in mind that all our logarithms will be base-2. If don't know what a logarithm is, or would like a quick refresher, read on for a very short introduction.

Logarithms can be taken in any base, but since we will be using the base-2 version in this paper, that is the version we'll talk about here. If you replace the 2 in the following formulas by an a, then we'd be talking about logarithms base-a. Here is the definition of a logarithm (base-2), where we use the small 2 as a subscript for the $\log$ to indicate base-2:

If $2^x = y$ then $\log_2 y = x$.

That's it!

But let's look at a few properties to make it easier to think about. First, a couple of examples. Check to see that these formulas make sense according to the definition above:

$$\begin{aligned}
\log_2 2 &= 1 \\
\log_2 4 &= 2 \\
\log_2 8 &= 3 \\
\log_2 1024 &= 10 \\
\log_2 1/2 &= -1 \\
\log_2 1/4 &= -2 \\
\log_2 1/1024 &= -10 \\
\log_2 \sqrt{2} &= 1/2 \\
\log_2 \sqrt[3]{2} &= 1/3 \\
\log_2 \sqrt[3]{4} &= 2/3
\end{aligned}$$

So basically, taking a logarithm of a number x asks, "To what power do I need to raise 2 to obtain x? It's sort of like undoing exponentiation. Now here are some key properties, all of which can be derived directly from the definition:

- It only makes sense (at least if we stick to the real numbers) to take logarithms of positive numbers. So $\log_2 0$ or $\log_2 -4$ are undefined.

- Larger numbers have larger logarithms.
- Numbers larger than 1 have positive logarithms and numbers smaller than 1 have logarithms that are negative.
- Extremely tiny numbers can have very large (but negative) logarithms. Mathematically, we'd say:

$$\lim_{x\to 0} \log_2 x = -\infty.$$

- $\log_2(x \cdot y) = \log_2 x + \log_2 y$. In other words, if you're working with the logarithms of numbers, you can effectively multiply the numbers by adding their logarithms.
- Almost equivalent to the statement above: $\log_2(x/y) = \log_2 x - \log_2 y$.
- Similarly: $\log_2(1/x) = -\log_2 x$.
- Every positive number has a logarithm, although generally they do not have a nice form, and all we can do is calculate a numerical approximation for them. For example, $\log_2 10 \approx 3.321928094887$. You can see that this is "reasonable," since $2^3 = 8$ and $2^4 = 16$, and since 10 is between 8 and 16 and closer to 8, its logarithm should be between 3 and 4 and closer to 3.
- If you have a scientific calculator that can find logaritms in some other base, but not base-2, you can calculate the base-2 value. Most scientific calculators can find logarithms either in base-10 or base-e, and the following formula holds for any positive x:

$$\log_2 x = \frac{\log_{10} x}{\log_{10} 2} = \frac{\log_e x}{\log_e 2}.$$

  (There is nothing special about the bases 10 or e above – if you can find logarithms in some particular base, you can find them in any base by means of a similar calculation.)

Exercise: Try to derive some or all of the properties above directly from the definition: If $2^x = y$ then $\log_2 y = x$.

2.1.3 A Formula for H

We will not prove this, but it turns out that there is basically only one type of function that satisfies all of the conditions above, and the total uncertainty, called the "entropy," has the following form:

$$H(p_1, p_2, \ldots, p_n) = -(p_1 \log p_1 + p_2 \log p_2 + p_n \log p_n) = -\sum_{i=1}^{n} p_i \log p_i.$$

The logarithms above can be taken to any base, but in information theory, it is customary (and we'll see why later) to use logarithms base-2. If we used other bases, the resulting values of H would simply be multiples of each other. The negative sign makes sense because all probabilities are at most one, so the logarithms are all zero or negative.

In the expression above, we will also assume that if some probability p_i is zero, then the term $p_i \log p_i$ is also zero in spite of the fact that the logarithm of 0 is undefined.

It is worth spending a bit of time checking so see that the conditions for an uncertainty measure are satisfied by that formula. Some of the checks, for example, the fact that the entropy is maximized when all the probabilities are equal, requires a bit of calculus, but you can check with some actual numbers to convince yourself, even without calculus, that it is the case. For example:

$$\begin{aligned} H(.5, .5) &= -(.5 \log_2(.5) + .5 \log_2(.5)) = 1.000000 \\ H(.51, .49) &= -(.51 \log_2(.51) + .49 \log_2(.49)) = 0.999711 \end{aligned}$$

2.2 Information

Again, consider a situation where the probabilities of the possible outcomes are $p_1, p_2, \ldots, p_n$. If the i^{th} outcome, with probability p_i occurs, how much information shall we say that result gives us? It would be nice to have a reasonable formula $I(p)$ that gives us the information gained if a result having probability p occurs. Intuitively, the smaller p is, the larger should be $I(p)$ since when a rare event occurs, you generally learn more. Similarly if $p = 1$ (you are certain of the outcome) then $I(p)$ should be zero, since you learn nothing by doing the experiment. If you do get some information, then $I(p) > 0$. (Remember the earlier question about whether one is the president of the United States.)

Also (hopefully) intuitively, if you repeat an experiment twice and obtain two results, one with probability p and the next time with probability q (and the results could, of course be the same with $p = q$) then it is reasonable to expect that the information from the combined results ought to be the sum of the information obtained from both of them. Thus:

$$I(pq) = I(p) + I(q).$$

We could make another list of reasonable properties for I as we did for H above, like the fact that I ought to be continuous, positive, et cetera, but it will turn out that a very reasonable measure of information is given by the following formula, where the base of the logarithm is again arbitrary, so we will use base-2:

$$I(p) = -\log p.$$

(The negative sign in front of the logarithm above is because we want information to be zero or positive. Since all probabilities are at most 1, their logarithms will all be zero or negative.)

Here is another way to convince yourself that the information should be related to the logarithm of the number of different messages instead of to the number of different messages. Suppose you have a transmission line that sends one bit (0 or 1) every second. A certain amount of information will be transmitted. Now suppose we have four copies of the same transmission line running in parallel. It's reasonable to assume that you are now getting information four times as fast. But if we look at what arrives every second, we could receive up to $16 = 2^4$ different patterns of bits from the four lines. But we are only receiving $4 = \log_2 16$ times the amount of information.

Now let's ask a *very* interesting question. If an experiment or situation plays out, what is the average expected amount of information we will receive as a result? For each of the possible outcomes, we need to multiply the probability of that outcome by the amount of information that becomes available as a result, and that will be the expected amount of information received. For our experiment with outcomes having probabilities $p_1, p_2, \ldots, p_n$, that value will be:

$$-(p_1 \log p_1 + p_2 \log p_2 + \cdots p_n \log p_n) = -\sum_{i=1}^{n} p_i \log p_i = H(p_1, p_2, \ldots p_n).$$

Thus the average amount of information gained from running an experiment or from a result occurring in a situation or the amount of information obtained in a message is the same as the entropy of the experiment/situation/message! So if you are trying to set up an experiment that will tell you the most in the long run as it it run over and over again, you should try to set it up so that the outcomes have probabilities that are as close to equal as possible. Or send symbols in a message that are as close to equi-probable as possible, et cetera.

Let's look at a couple of examples:

2.3 Twenty Questions

Next, consider the game of "Twenty Questions" where your opponent thinks of an animal and you can ask 20 yes-no questions to try to figure out what animal it is. From the discussion above, we know that with some perfect set of questions, there are $2^{20} = 1048576$ different sequences of 20 yes-no answers, so in principle, the guesser could distinguish among 1048576 different animals. If you want to win as often as possible, the best strategy with each question is to split the number of possibilities as close to in half as possible; in other words, you'd like to maximize the amount of information each answer gives you, no matter what it is.

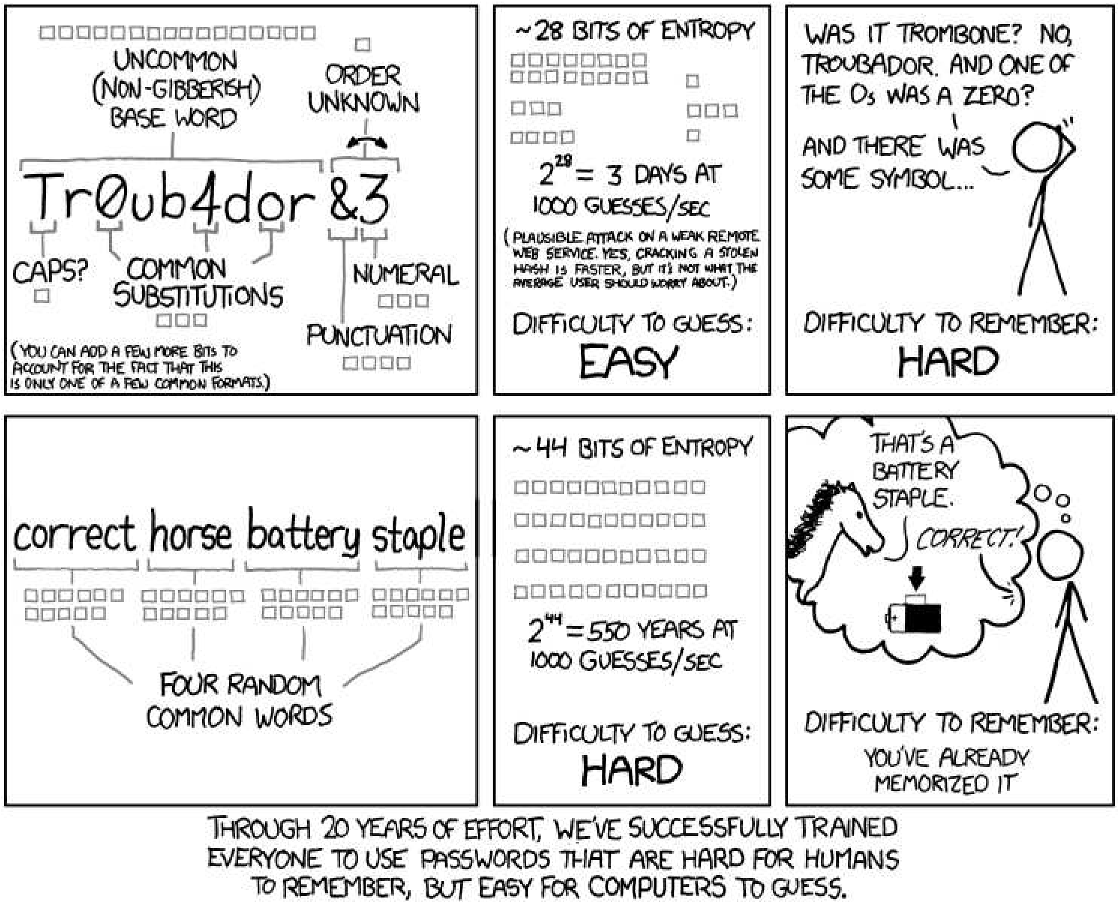
To obtain the most information out of your first question, you would like to construct one whose answer is as close to having a "yes" or "no" probability as possible. To make this obvious, the worst possible sort of initial question might be, "Is it a lion?" If your opponent knows 10000 different animals and selects one at random, then your probability of getting a "yes" is 0.0001 and is 0.9999 of getting a "no", yielding a very tiny gain in information, on average.

Or perhaps even more obvious: I have picked a number from 1 to 1048576 and you have 20 yes-no questions to find it. If each question splits the number of possibilities exactly in half, you'll be certain to get it. Your first question might be, "Is it smaller than or equal to 524288?" Each guess will split the remaining possible group of numbers exactly in half and at the end you'll be left with exactly one of them. If your first question is, "Is it 417?" and it isn't, even with the best possible play after that you'll only win about half the time.

Here's an even easier way to play 20 questions to find a number between (let's adjust the range by 1) 0 and 1048755: Every number in the range can be represented by a 19-digit binary number (perhaps with leading zeroes). The first question is: "Is the first binary digit equal to 1?" Then ask about the second, third, …, and obviously, at the end, you know all 19 binary digits and can guess correctly for your 20^{th} guess.

2.4 Entropy As Bits/Symbol

From the web cartoon XKCD:


UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor &3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

If the logarithm is taken to be base-2, then the entropy of a set of messages can be interpreted as the average number of bits per message transmitted. Let's consider a couple of examples.

First, assume that there are only two messages, and they both have probability 0.5. In this case where one is indicated by a 0 and the other by a 1, the average amount of information (with the logarithm in base-2) is:

$$-(.5 \log .5 + .5 \log .5) = 1.$$

Suppose now there are four possible messages, all equally-likely, and they are transmitted by the four possible bit patterns: 00, 01, 10 and 11. The entropy of this situation (which is also the average amount of information transmitted) is:

$$-(.25 \log .25 + .25 \log .25 + .25 \log .25 + .25 \log .25) = -4(.25 \log .25) = 2.$$

It is also easy to check that with 2^n equally-likely messages that can be sent with n bits, the average amount of information transferred is:

$$-2^n(1/2^n)\log(1/2^n) = n.$$

Basically, all the results above show that in the best possible case, n bits of data can provide an amount of information equal to n. That's why the entropy/information content, when the logarithm is base-2, can be referred to as the number of bits of data.

If the messages are not equally-likely, then there is some waste with the schemes above, and the amount of waste depends on how far from equally-likely the messages are. So if we have four messages with unequal probabilities and we use the obvious 2 bits to transmit each one, the average amount of information passed will be less than 2 bits.

Exercise: Show that if one of the events is certain, then the amount of information passed by transmitting it is zero. What is the amount of information passed, in bits, for a 1-bit message where the probability of one outcome is 0.25 and of the other is 0.75?

Exercise: Suppose you need to transmit information about the status of some equipment which is almost always (say 99% of the time) working correctly, but when there is a problem, you would like to report what type of problem it is. Say that when there is a problem, it is problem A 50% of the time, but it could also be problem B or C (25% of the time each). Show that the entropy of this situation if you do it the obvious way (with four 2-bit messages) is about $.0958$ (less than $1/10$ of a bit per message, although it costs you 2 bits). Is there a better way? (The best solution will be presented in the following Section 3.

3 Data Compression

3.1 An Easy Example: Telephone Numbers

Let's start with a familiar example: telephone numbers. Ideally, to make a phone call, you would like to have to dial as few numbers as possible. Most telephone numbers in the United States are seven digits long, partly because seven digits is about the most you can expect the average human to remember between the time he looks it up and the time he dials.

Of course, even if all possible 7-digit numbers were usable, that's only 10^7, or ten million numbers, and there are far more telephones than that in the United States. So what is done is that the country is divided into regions, and each region is assigned a 3-digit area code. If you are calling a number within your region, you don't need to include the area code, but if you want to call outside your region, you need 10 digits: the area code plus the 7-digit number identifying the phone inside the region.

But there's a problem: how does the phone company computer know that you are dialing a 7-digit or a 10-digit number? After all, you might be dialing 10 digits, but pause for a moment after dialing 7 of them. The answer is that no 7-digit numbers nor 3-digit area codes begin with the digits 0 or 1. If you want to dial outside your area code, you need to dial a 1 first, then the area code, then the 7-digit local phone number. (This may be handled differently in different parts of the country.) Since no area codes or local numbers begin with a 1, then if the first number you dial is 1, the phone company computer knows that the next three digits are the area code and the following 7 are local.

(Of course since we cannot have a 0 or a 1 as the initial digit of a local phone number, there are only $8 \times 10^6 = 8,000,000$ different possible local numbers instead of $10^7 = 10,000,000$ of them.)

It used to be that the local 7-digit numbers had a "prefix" and a 4-digit number (as in the Marvelettes' song, "Beechwood 4-5789"). The first two letters could be remembered as two letters on the phone where the digit 2 was ABC, digit 3 was DEF, and so on. Thus "Beechwood 4-5789" was 234-5789. Since neither the 0 nor 1 had any letters assigned to it, it was impossible to have a 0 or a 1 as the first or second digit of prefix. (I just know the system where I grew up, in Denver, and the "prefixes" were effectively three digits. The first two were letters, like "FR", for "Fremont", but it was *always* "FR-7"; there was never an "FR-6" or "FR-8". Maybe it was different in other places.) Because of that, it was initially easy to determine whether the first three digits were an area code or not, since all the area codes had a 0 or a 1 as their second digit. So as soon as the telephone company's computer saw you dial "213…", the 1 in the second position made it clear that you were dialing an area code, and it turned out that it was in the Los Angeles area[2]. This restriction has been dropped to make more local numbers, so the leading 1 is now required.

There are even more problems if you want to dial internationally: different countries have different systems (perhaps not 7-digit numbers nor area codes, but something else). The solution is that if you begin by dialing 011, the next information to dial is a country code, and after that, the US phone company, since it knows that the call is international, can interpret the next digits as a country code, and then just send the following numbers to that country's local phone system for it to figure out.

Notice how this system will tend to reduce the total number of digits that are dialed: most of your calls are to friends or associates in the local area, so most of your calls require only 7 digits. Less likely is for the calls to be in another area code, in which case you need to dial 11 digits (the leading 1, then three digits of area code, plus seven digits of local number). In what is usually the rarest case, international calls, you need to dial enough numbers to get out of the US system, and then you have to dial whatever is necessary to connect to the foreign telephone.

In fact, this system can even be simplified in some cases. Suppose you work for a company with fewer than, say, 900 employees. For some jobs, it is likely that most of your calls will be to others within the company, so you can dial three digits and that's enough to connect you to the correct person within the company. Of course, if you want an outside line, you dial, say, a 0, followed by whatever you would do to make the outside call, be it local, in another area code, or international. The reason for the limit of 900 is that none of the internal extensions can begin with 0, since a leading 0 indicates an outside line. If the company has fewer than 9000 employees, then a 4-digit extension will handle all of them, et cetera.

Let's consider the situation where all we're concerned with is calls within the United States. Suppose that 90% of the numbers you call are within your area code and 10% are outside it. That means that 90% of the time you'll have to dial 7 digits and 10% of the time, 11 digits. On average, the number of digits you need to dial is:

$$0.90 \times 7 + 0.10 \times 11 = 7.4$$

If you just required the area code every time, even for local calls, then you wouldn't need the leading 1, so every call would require 10 digits. When you call local numbers more often, you'll save on the number of digits required, and in this example, you save 2.6 digits per call, on average.

3.2 Applications for Compression

There are many situations where it is useful to compress information. Compressed information takes less storage space and can be transmitted more rapidly. It might seem like the huge capacity of today's computer disks or the increase in speed of computer networks has made it uimportant, but when we humans have a higher capacity, we just find more

[2]With the old rotary-dial phones, it was faster to dial shorter numbers, so area codes for the more populated areas had easier-to-dial area codes. For example, 212 = New York, 312 = Chicago, 313 = Detroit.

intensive uses for it. At first text was relatively easy to send, but pictures were difficult since they contained so much information. With increases in transmission speed and compression techniques like jpeg, pictures became reasonable to transmit, so we started transmitting sound and video. As soon as we could transmit short, jerky videos efficiently, we started asking for more frames per second or more resolution. There seems to be no end to our desire to transmit more information and that, faster.

You can always think of things that would be nice. Wouldn't it be nice to have all the books in the Library of Congress saved on a single small device? Or all the music ever written, or all the movies ever produced? At high quality?

The only reasonable way to study compression is not to begin with these terribly complicated examples, but to learn how to compress data efficiently with small examples and then to see how the lessons we learn there can be extended to larger, real-world applications. Instead of the Unicode alphabet of about $65,000$ characters, let's see how we would deal with a language with just two or three or four characters.

So be patient for the next couple of sections where the examples probably seem very contrived; they will lead us to methods that could be applied to the problem of transmitting the entire Library of Congress.

3.3 Transmitting Messages

Let's start a very simple system: you wish to send messages over a communication line, and you want to transmit the messages as fast as possible. Let's assume that you can only send your messages as a series of 1's and 0's (which is exactly the case for almost all digital transmissions these days). A single 0 or 1 is called a "bit" of information.

We will be very flexible at first in what we mean by a "message". Suppose, for example, you have an elevator, and there is a light that goes on if the elevator is there. Every second, the elevator needs to send one of two messages: "I am there" or "I am not there". We can imagine that the elevator sends a 0 if it is not there and a 1 if it is there. Just by looking at this single bit of information, the elevator control software can decide whether to turn on the light or not. In this case, this is about the best you can do: every message is exactly one bit in length.

But let's suppose there are more than two messages: the two above, plus one more that means the elevator is locked on a floor and another that means it is broken. Now there are four possible messages, so one bit of information isn't sufficient, but we can easily encode the messages with two bits of information:

I am not there	00
I am there	01
Locked	10
Broken	11

To interpret the situation, you need to look at two bits at a time, and every message will be two bits long.

In a situation like this, however, the locked and broken situations are very rare, and if the building has a lot of floors, then the message "I am not there" will be far more common than the message "I am there". Just to have some numbers, let's assume that 90% of the time the message is "I am not there", 8% of the time the message is "I am there", and 1% of the time each of the other two messages are sent. Consider the following encodings for the messages:

I am not there	0
I am there	10
Locked	110
Broken	111

It may seem strange to have different-length encodings for the messages, but notice first that this is a bit like the telephone numbers: the most common message is sent with the least number of bits, and there is never a question about the message.

Can see why there is never any confusion about the message? The answer is in this footnote[3].

[3] If you see a 0 first, then the message is over and the elevator is not there. If you see a 1, you have to wait for the next bit. If that next bit is a 0, you know the message and you're done, but if not, you need to look at the third bit. Now, in any case, the message is over and you can start looking

With this encoding mechanism, what is the length of the average message in bits? Well, it is:

$$0.90 \times 1 + 0.08 \times 2 + 0.01 \times 3 + 0.1 \times 3 = 1.12$$

With this encoding, on average, you save 0.88 bits per message.

3.4 A Real World Example

The example above with the elevator is fairly contrived, but exactly the same idea can be used for messages that are more like the ones you typically send. Let's look at messages that are just typed with normal text in English.

If you look at lots of English text, you will note that different letters have different probabilities of occurring. The letter "e" is by far the most common, and letters like "x", "j" and "z" are very rare. There are a lot of standard encodings for letters and probably the most common one today is the ASCII encoding which assigns 7 bits to each letter. This allows for $2^7 = 128$ letters, and that is plenty for the 26 upper-case and 26 lower-case letters, all the punctuation, the space character, tabs, control characters brackets, braces, and other miscellaneous characters.

The ASCII encoding is very simple to work with: every chunk of 7 bits represents the next character in a string, but in terms of packing English text, it is very inefficient since the actual characters have wildly different frequencies. In fact, the space character is the most common, followed by "e", and so on. It isn't hard to examine large chunks of English text to get a rough idea of the relative frequencies of the letters, and once we have those, we can find an encoding of the character set that is far more efficient in terms of the time it would take to transmit typical messages or the space that it would take to store them on a computer.

One question we would like to answer here is that given the set of relative frequencies of the letters, how do we find the most space-efficient encoding for them?

Rather than start with the full ASCII set that contains all 128 characters, as usual, it is best to begin with the simplest possible situations, and work up from there. Let's just look at tiny alphabets and see what we can do.

Note: In certain situations, we can do even more, and we will talk about those later, but for now, we'll assume that a bit sequence must be sent for each letter. We'll start with 2-character alphabets, then 3, and so on. To make the notation easier, suppose there are n characters and we call them $C_1, C_2, \ldots, C_n$, where C_1 is the most common character and they are listed in order of frequency, so C_n is the least common. Let's write p_1 for the probability that C_1 occurs, p_2 for the probability that C_2 occurs, and so on. If there are n characters, then:

$$p_1 + p_2 + \cdots + p_n = 1,$$

with

$$p_1 \geq p_2 \geq \cdots \geq p_n.$$

If we use b_1 bits to represent the first character, b_2 to represent the second and so on, then the average number of bits required to send a single character is:

$$p_1 b_1 + p_2 b_2 + \cdots + p_n b_n.$$

3.4.1 A 2-Character Alphabet

If there are only two characters, we can assign 0 to one and 1 to the other. The average number of bits sent for each character is $p_1(1) + p_2(1) = p_1 + p_2 = 1$.

for the next message.

3.4.2 A 3-Character Alphabet

We're going to need at least 2 bits for some of the characters since with just 0 and 1, only two characters can be distinguished. The easiest way would simply be to assign two bits to each, like 00, 10 and 11. The combination 01 would never occur. But this is clearly wasteful: once you see a leading 0, you know the next bit has to be zero in this encoding, so why bother to send it? A better encoding would be 0, 10 and 11. (Or equivalently, 1, 00 and 01 — there's nothing that makes a 0 better than a 1: both take the same amount of space and the same amount of time to transmit[4]).

Since we're trying to minimize the average number of bits sent, we should assign the 0 code to the most common letter, and the 10 and 11 codes to the other two. It doesn't matter how the last two are assigned; they both will require 2 bits.

Can you prove that this is the most efficient encoding? (Hint, work out the average bit count per letter under the different scenarios, keeping in mind that $p_1 \geq p_2 \geq p_3$.) The answer appears in Appendix A.0.1

3.4.3 A 4-Character Alphabet

Again, the easiest encoding would be to use all the 2-bit patterns to cover the four possibilities: 00, 01, 10 and 11, although in the elevator example, we've already seen that this may be a bad idea.

At this point, things begin to get tricky. To see why, suppose all four characters are equally likely: that each has probability 25% of occurring (in other words, $p_1 = p_2 = p_3 = p_4 = 0.25$). Then the assignment to the four possibilities works best, since the average message size is 2 bits, and if we used the encodings we did for the elevator: 0, 10, 110 and 111, we would do worse. The average number of bits would be:

$$0.25 \times 1 + 0.25 \times 2 + 0.25 \times 3 + 0.25 \times 3 = 2.25,$$

which is, on average, 0.25 bits more per character, on average.

When should you use each kind of encoding? Think about this for a while before reading on.

The answer is that if you add the two lowest probabilities together and obtain a number that is larger than the highest probability, then the best encoding is to use 2 bits for each; otherwise, assign them as 0, 10, 110, 111 for the highest probability to lowest probability character.

3.4.4 Alphabets With More Than 4 Characters

With 5 or 6 characters in the alphabet, it's still probably possible to work out the optimal binary encodings by brute force, but in the real world where alphabets are generally much larger, this method becomes far too tedious.

As an example, try to find as good an encoding as you can for the following set of characters with the given probabilities of occurrence. In other words, what would be an optimal encoding for the characters "A" through "H" below, and what would be the average number of bits per character in that encoding? Since there are 8 characters, we could clearly just assign to each one of the eight 3-bit encodings: 000, 001, 010, 011, 100, 101, 110, 111, and we would have an average of 3 bits per character. How much better than that can you do?

[4]Notice that this may not always be true: using Morse code as an example, it takes about three times as long to send a "dash" as a "dot".

i	C_i	p_i
1	A	0.28
2	B	0.27
3	C	0.23
4	D	0.09
5	E	0.04
6	F	0.04
7	G	0.03
8	H	0.02

Try to work on this for a while and see how well you can do. The answer appears in Appendix A.0.2

3.4.5 A Practical Example

Assuming you know how to construct an optimal encoding for any distribution of characters (which we will find later), how could we use this?

Suppose you have huge documents with millions or even hundreds of millions of characters. You would like to compress them so as to take up as little storage space as possible. One approach would be, for each document, to have the computer make a pass through and count the number of instances of each character, including spaces, punctuation, et cetera. Then it can generate, using rules we will discover later, an optimal Huffman encoding for that character distribution. Finally, use the first few hundred characters of the encoded document to list the particular encoding for that document, and this would be followed by the Huffman-encoding of the original text.

Doing this for each document has the advantage that if the documents have wildly-different distributions, each will be compressed optimally. For example, texts in Italian will have a lot more vowels than texts in English and would need a different encoding. If a document consisted of mostly mathematical tables, almost all the characters might be digits, so a Huffman encoding that gave digits priority would allow the document to be compressed far more efficiently, et cetera.

3.5 Huffman Encoding

It should be clear that whatever the best solution is, it ought to assign shorter codes to the more common characters and longer codes to the rarer characters. This is true since if there is a pair of characters C_1 and C_2 where C_1 is the most common but has a longer encoding than C_2, then you could just swap the encodings and have a more efficient final result.

We can think of the problem as a series of investments of bits. The longest strings of bits should go with the least common characters, so pick the two least-common characters and spend one bit on each (a 0 and a 1) to tell them apart. This will be the final bit in the code for each, with some unknown number of bits preceding them which are the same. But now every time you add one bit to that preceding part, the cost, in terms of number of total bits spent, will be the sum of the costs for the individual characters, since that extra bit has to be added to both codes. This sentence is a little vague, but it provides an insight into how an optimal encoding is derived and to why it is the best.

As an aid to calculation, notice that it is not necessary to use the true probabilities when you are trying to optimize an encoding; any multiple of all the probabilities will yield the same minimum. In other words, suppose there are five characters and you've done a count of the number of times each character occurs and you come up with the following counts:

A	123
B	82
C	76
D	40
E	11

To find the probabilities you need to add all the character counts: $123 + 82 + 76 + 40 + 11 = 332$ and then divide each of the numbers above by 332, so for example, the probability of obtaining an "A" is $123/332 = .3704\ldots$ — a bit more than 37% of the time. The same can be done for each of the numbers above, but notice that they all will be equal to 332 times the actual probability, so if you just use 123, 82, and so on, all your sums will be exactly the 332 times as large as what you'd get using the true probabilities.

So let's try to figure out logically an optimal encoding for the character set above. There are 5 characters, so 2 bits will not be sufficient; at least one of them will require 3 bits of data (and perhaps more). We can't allow four of them to have 2 bits and the other 3 bits, since the 3-bit encoding will have to start the same way as one of the 2-bit encodings and there will be no way to tell whether the 2-bit character ended, or whether we need to wait one more bit for the 3-bit character. So at least two of the characters will require at least 3 bits. As we stated above, the letters "D" and "E" should be the ones chosen to have the longest string of bits in their codes, so we can arbitrarily say that D has a code of $x \ldots x0$ and E a code of $x \ldots x1$ where the earlier bits indicated by the x's (we don't know how many yet) are the same.

But each time we add a bit to replace the x's above for the D and E characters, we add one bit to the representation of 51 characters (both the 40 D's and the 11 E's). In a sense, we can treat the combination of D and E as a single character with weight 51 together with the other three letters, and we'd have a chart that looks something like this:

A	123
B	82
C	76
(DE)	$40 + 11 = 51$

Now the least common "letters" are C and (DE), where (DE) is the combination of D and E. We should use a bit to distinguish between them, and we obtain the following chart:

(C(DE))	$76 + 51 = 127$
A	123
B	82

The C combined with (DE) now is the most-likely "letter", so it moves to the top of the chart. The least likely of those left are A and B, which, when combined, yield:

(AB)	$123 + 82 = 205$
(C(DE))	$76 + 51 = 127$

Finally, one more bit needs to be added to distinguish between the two combinations above, and it could be indicated as follows, continuing the pattern above:

((AB)(C(DE)))	$205 + 127 = 332$

Each pair of parentheses contains two "subcodes" each of which is either a letter of a grouped pair of similar items. We can unwrap the parentheses as follows. Assign a leading 0 to one of the subcodes and a leading 1 to the other. Then continue recursively.

Thus the first bit tells us whether to go with (AB) or (C(DE)); arbitrarily use 0 for (AB) and 1 for (C(DE)). If we see a 0, we only need one bit to distinguish between A and B, so 00 means A and 01 means B. If we see a leading 1. then the next bit tells us whether we have a C or a (DE), et cetera, so the codes for the last three should be 10 for C, 110 for D and 111 for E:

Character	Frequency	Code
A	123	00
B	82	01
C	76	10
D	40	110
E	11	111

This optimal encoding requires only 3 bits for the least-common character, but there are situations with only 5 characters that 4 bits would be required for an optimal encoding. Here's an example:

Character	Frequency	Code
A	16	0
B	8	10
C	4	110
D	2	1110
E	1	1111

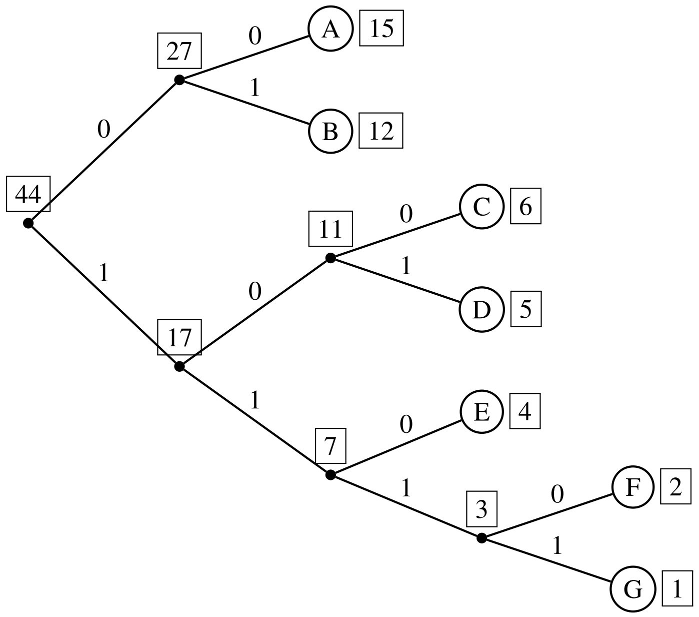
Follow the logic we used for the previous example to see that the structure for the encoding would be (A(B(C(DE)))).

3.6 Tree Formulation

Another way to visualize the Huffman encoding is as a tree. If we follow the same procedure as above for the following distribution of letters:

A	15
B	12
C	6
D	5
E	4
F	2
G	1

we obtain the encoding structure ((AB)((CD)(E(FG)))). This can be displayed as the binary tree below, where the letters are the final nodes, the numbers in boxes are the weights, both of the final and internal nodes and the 0's and 1's above the lines indicate the encoding. To find the code for any particular letter, just find the unique path to the letter from the root (numbered 44) and read off the numbers on the connecting edges.



For example, to get to D from the root we follow three edges labelled 1, 0 and 1, so the Huffman encoding for D in this example is 101. Here is the complete encoding:

A	15	00
B	12	01
C	6	100
D	5	101
E	4	110
F	2	1110
G	1	1111

3.7 Entropy and Huffman Encoding

Suppose we have two possible messages, A and B, and the probability of message A is $3/4 = 0.75$ and of B, $1/4 = 0.25$. We can calculate the Entropy of this situation as follows:

$$H = -(.75 \log_2 .75 + .25 \log_2 .25) \approx 0.811278$$

and earlier we stated that this is measured in bits.

Now if we encode A as 0 and B as 1, then we "spend" one bit per message, so in a sense, we're wasting about $1 - 0.811278 = .188722$ bits for each message. Is there some clever way that we could get by with fewer bits? If we're willing to group the messages, we can do better using a Huffman encoding.

We can group as many messages as we want, but let's just group them three at a time, so that we send sequences of bits that correspond to sets of three messages. Here is a table of all the possible 3-message groups, together with their probabilities and an optimal Huffman encoding (there is more than one optimal encoding) for the groups:

Group	Probability	Encoding
AAA	$27/64 = 0.421875$	1
AAB	$9/64 = 0.140625$	001
ABA	$9/64 = 0.140625$	010
BAA	$9/64 = 0.140625$	011
ABB	$3/64 = 0.046875$	00010
BAB	$3/64 = 0.046875$	00011
BBA	$3/64 = 0.046875$	00001
BBB	$1/64 = 0.015625$	00000

Over the long run, about $27/64$ of the time we'll send a 1-bit message, $(9+9+9)/64 = 27/64$ of the time, a 3-bit message, and $(3+3+3+1)/64 = 10/64$ of the time, a 5-bit message. The expected (average) message length (in bits) will thus be:

$$\frac{27}{64}\cdot 1 + \frac{27}{64}\cdot 3 + \frac{10}{64}\cdot 5 = \frac{158}{64}.$$

Since each of these 1-, 3- or 5-bit grouped messages corresponds to a group of three of the original messages, the average number of bits sent per original (A or B) message will be: $158/(3\cdot 64) = 158/192 \approx 0.822917$, which is very close to the 0.811278 bits predicted by the entropy calculation. Grouping more messages will yield even better approximations.

Exercise: Work out a Huffman encoding if we pack only two of the original messages above per group. Show that the bit efficiency is $27/32 = 0.84375$, which is already pretty close to the value predicted by the entropy.

4 ASCII Character Encoding

The acronym ASCII stands for "American Standard Code for Information Interchange" and is a method to represent characters using sequences of bits.

How many bits are required to cover a "reasonable" character set? Well, in English, there are 26 letters in the alphabet and we'd like both the upper-case and lower-case versions, for 52. We also need the numerals 0 through 9 for 10 more, and then a few more for punctuation, special characters like "#", "$", "%", the space character, et cetera. It's clear that there will be more than $64 = 2^6$, and it's also fairly clear that we won't need $128 = 2^7$.

Thus 6 bits doesn't supply enough different "messages" (where each "message" represents a single character) and 7 bits gives us more than enough. So the ASCII encoding assigns characters to each of 128 7-bit combinations as follows:

	000	001	010	011	100	101	110	111
0000	^@	^A	^B	^C	^D	^E	^F	^G
0001	^H	^I	^J	^K	^L	^M	^N	^O
0010	^P	^Q	^R	^S	^T	^U	^V	^W
0011	^X	^Y	^Z	^[	^\	^]	^^	^_
0100		!	”	#	$	%	&	’
0101	(	)	*	+	,	-	.	/
0110	0	1	2	3	4	5	6	7
0111	8	9	:	;	<	=	>	?
1000	@	A	B	C	D	E	F	G
1001	H	I	J	K	L	M	N	O
1010	P	Q	R	S	T	U	V	W
1011	X	Y	Z	[	\	]	^	_
1100	‘	a	b	c	d	e	f	g
1101	h	i	j	k	l	m	n	o
1110	p	q	r	s	t	u	v	w
1111	x	y	z	{	|	}	∼	DEL

To find the bit pattern for a character in the array above, concatenate the three bits in its column to the end of the four bits in its row. For example, the ASCII encoding for the numeral “5” is 0110101.

Exercise: Determine the characters spelled out by the following sequence of bits:

$$1010010\ 0110010\ 1000100\ 0110010$$

Exercise: Write your first name as a sequence of ASCII bit sequences.

The “control characters” like ^A, ^B and so on were originally commands that controlled functions on a teletype machine. Control-G would ring a bell, control-M was a carriage return, control-H was a backspace, control-I was a tab character, et cetera.

With computers it is also very useful to be able to consider the binary encodings as 7-bit binary numbers with values from 0 to 127. Here are a couple of examples:

- If you know that you’ve got a numeral, you can find the numerical value of that numeral by subtracting from its ASCII code the ASCII code for zero. For example, 0110101 (= 53, base-10) represents the character “5” and 0110000 (= 48, base-10) represents the ASCII “0”, and $53 - 48 = 5$.
- It is easy to check to see into what class a character with an ASCII code falls. For example, if the character has a code number between (and including) that of “A” and “Z” then it is an upper-case letter.
- If you have a character that you know is in upper-case, you can convert it to lower-case by adding the value of ’a’−’A’ (in ASCII) to it. That’s because each of the lower-case letters have ASCII values that are exactly 32 larger than the corresponding upper-case letter.

4.1 Extensions of ASCII

Since most modern computers organize their data into 8-bit bytes, standard ASCII characters, as described in the previous section, are represented by 8-bit patterns where the first bit is a 0 and the last 7 are the ASCII code. That leaves another 128 values (with a leading 1, or, in other words, corresponding the values from 128 to 255). There is no universal standard for this, so documents that are produced using the extended ASCII on a Macintosh computer may look wrong on a PC and vice-versa.

Since there is no standard, it’s a little dangerous to use one of these extensions.

Usually, the extended character set is used to represent things like accented characters, mathematical symbols, and other things.

4.2 Unicode

Most western alphabets, even those with many accented characters, require fewer than 256 characters to represent all the possibilities, but asian languages like Chinese or Japanese, have symbols that stand for syllables or words that number in the thousands.

The need for many more characters and the fact that the ASCII extensions are not standard led to the development of a more complete and extensible system of character encoding called Unicode. Unicode is more complex than ASCII in that different characters are represented by different numbers of bytes, up to 4. If every combination of 4 bytes were used, that would allow for up to $2^{32} = 4294967296$ different characters. The actual number is significantly less, however, since an effort was made to make ASCII a subset of Unicode so that old ASCII files would be interpreted correctly.

The way this works is that if a byte has a leading 0 bit, it is interpreted as a single ASCII character, but if the leading bit is a 1, then, depending on the situation, the next 1, 2 or 3 bytes are used to determine the character. The details are too complex to discuss here, but the idea is somewhat related to Huffman encoding that we discussed earlier in this document (see Section 3.5).

5 Error Detection and Correction

Up to now, we have considered the case where the recipient of a message receives the message from the sender with zero transmission errors. In the real world, this is unlikely to occur 100% of the time due to noise in the transmission lines, et cetera. What would be nice is to have some mechanism for the recipient to know if an error occurred, in which case he could request that the message be resent, or better still, that there be enough redundancy in the message that the recipient can correct the errors on his end. In this section, we'll consider those sorts of problems.

Of course there is no way to guarantee that a message is received without error since no matter how small the probability of an error is for each bit transmitted, it could be true that every one of them is wrong. For example, suppose your message is just a "yes" or "no": in other words, a single bit. Let's also say the the probability that there is an error is 1 in 1000, or $1/1000 = 0.001$. If you just send a single bit, then you'll succeed 999 times out of 1000 which is pretty good, but you decide to be even safer, and you send three copies of the bit, and the recipient will look at all three bits and will assume that your meaning is the bit that is most common in the received set.

We can just use the binomial theorem to find the probability that an error will be made. This will occur if two of the three or if all three bits are transmitted in error. If p is the probability of an error (0.001 for this example), then the probability that all three are wrong is p^3 and the probability that two of the three are wrong is $3p^2(1-p)$. (The 3 is because the correct bit can be in any of the three positions.) Anyway, the probability of an error is: $p^3 + 3p^2(1-p)$ and if $p = 0.001$, that amounts to 0.000002998: about 3 chances in a million. On the other hand, if there is any disagreement in the three arrival bits the recipient knows that some error occurred and can ask for a retransmission, accepting it only when all of the three bits that arrive are the same. In this case, there will only be an error when all three are wrong, and that will occur only one time in a billion.

The example above illustrates two different ideas: that of error detection and of error correction. The system above (for a single bit of information) allows error detection for up to two errors in the three bits and for error correction if only one of the three bits is in error.

If it is possible to request a retransmission whenever an error is detected, then in this case we can reduce the probability of an error to one in a billion, but sometimes there is no "retransmission" possible. Consider the situation where we want to store one bit of data in a computer chip. If we write three copies of it, and try to read it later, only to discover that at least one error has occurred, since we can't go back and ask for it to be rewritten, the best we can hope for is to do the "correction" of assuming that the value that arrived two times out of three is the correct one and as we have seen, this will give us an error about 3 times in a million (assuming that $p = 0.001$, which is *much* larger than typical memory errors in modern computer chips — errors on transmission lines are much higher).

5.1 Error Detection: Parity

One method that can easily be used to detect most errors is to add a "parity bit" to each collection of data bits. Typically, a ninth bit is added to each 8-bit character and when 8 bits of data is stored, the parity bit is set to either 0 or 1 in such a way that the total number of 1 bits is even (for a system with even parity) or odd (for a system with odd parity). Then, when the data is read, hardware checks that the 9 bits have the appropriate parity and if not, an error is detected. Obviously, if there are errors in 2 (or 4 or 6 or 8) of the bits, the error will go undetected, but if the probability of an error in at leasat one of the bits is small, say p, then the probability that an error will go undetected is about p^2, which is tiny. Of course, if there is an error, no correction is possible.

5.2 Error Detection: Checksums

Suppose that a message consists of a lot of data, perhaps in 8-bit bytes (basically, characters). Another sort of error check can be accomplished by simply considering each byte to be an 8-bit number (from 0 to 255), adding the values together ignoring any overflow, and transmitting that number (also just 8 bits) at the end as a checksum. The recipient adds the values of the received bytes in the same way and compares the checksum. If they are different, an error has occurred (but obviously there's no easy way to tell exactly what it was).

For an error to be missed using the checksum, the sum will have to be exactly the same as the original checksum, and, assuming the errors are randomly-distributed among the bits, the chances of an error being undetected are only 1 in $2^8 = 256$.

If desired, the checksum can be used in combination with parity bits on each byte which will reduce the probability of an error being missed to a much smaller value.

Also, of course, there is no reason to restrict the checksum to just 8 bits: the transmitted bytes can be taken 2 or 4 at a time to have a checksum of 16 or 32 (or more if you want) bits. With a 32-bit checksum, random errors will go undetected only one time in $2^{32} = 4294967296$.

Just adding together the bytes or words of data is only one way to form a checksum. At times it's better to perform a more complex operation involving, say, using the logical XOR function to merge in new bytes/words, and possibly rotating the bits in the sum before each XOR operation.

5.3 Error Detection: Hash Functions

If there is no reason to protect the content of the message from others, then checksums are usually sufficient, but what if it is important to know that the message has been received unaltered. In other words, the recipient would like to know that the content of the message has not been modified by a third party. If some sort of simple checksum is used, then in theory an opponent could modify the message in such a way that the checksum remained the same, or as long as the message itself is being modified, why not also modify the checksum so that things look ok?

One solution is to have a more complex scheme for generating a different kind of checksum such that even knowing the checksum it is not feasible to construct a message having that checksum. But if the message and checksum are sent together, then of course the opponent can modify the message, build the new checksum, and send both a modified message and checksum. There is a way around this that we'll discuss later.

5.4 Error Correction Methods

Exercise: How many protection bits must be provided if there is a single bit of data so that any single-bit error can be corrected and any double-bit error will be detected? Note that the protection bits may also be in error. Our example at the beginning of this Section 5 where we simply sent three copies of each bit does provide single-bit correction, but does not provide for the detection of an error when two bits are wrong.

If you found a method to solve the exercise above with three additional protection bits then you found an optimal solution. This probably seems very inefficient: you need to send four times as much information as the message contains to protect it in this way, even through the message contains only a single bit of data. It turns out that for larger messages, it gets cheaper and cheaper. For example, if the message contains 500 bits of data, only 10 additional bits need to be sent to guarantee error correction with a single bit in error and error detection if exactly two bits are wrong.

Exercise: Find a way to protect (1-bit correction, 2-bit detection) for two bits of data that requires only 4 protection bits.

To figure out how to construct such a protection scheme, let's first consider a simpler problem: assume that the incoming data is either completely correct or has exactly one error and we need to be able to correct that error from the transmitted data. We'll worry about extending the method to cover detection of the two-error case later.

Suppose that there are k bits of data, or information, and that there are m bits of protection, so the total transmission will consist of $k + m$ bits. Since, for now, we are assuming that the transmission was either perfect or that there was exactly one bit in error, we have to detect $k + m + 1$ outcomes (any of the $k + m$ bits could be in error, or it could be error-free). By examining the m protection bits, there have to be at least $k + m + 1$ different patterns. With m bits, there are 2^m patterns, so to be able to correct any single-bit error, we must have:

$$2^m \geq m + k + 1.$$

It turns out that as long as m is large enough for this to be true, a suitable code can be arranged and the method basically amounts to having the m bits used for various parity checks on the total transmission.

Exercise: Before continuing, see if you can figure out how to construct such a scheme (1-bit error correction only) where $k = 4$ and $m = 3$ (which can work, since $8 = 2^3 \geq 4 + 3 + 1$). As a hint, let's label the bits in the total message as $1, 2, \ldots, k + m = 7$ and we will interpret the $m = 3$ protection bits as a 3-bit number from 0 to 7, where a 0 indicates no error, and otherwise the number corresponds to the bit that is in error.

Let's solve the exercise in the previous paragraph in a very nice way that will illustrate exactly how the same thing can be done for any number of bits. Let's call the data bits d_0, d_1, d_2 and d_3. The protection (or parity) bits will be called p_0, p_1 and p_2. Consider the following table:

	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
Data	p_0	p_1	d_3	p_2	d_2	d_1	d_0
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1
10	1	0	1	1	0	1	0
11	0	1	1	0	0	1	1
12	0	1	1	1	1	0	0
13	1	0	1	0	1	0	1
14	0	0	1	0	1	1	0
15	1	1	1	1	1	1	1

With the four bits of data, we can transmit a number between 0 and 15, inclusive. That number is in the "Data" column, and if you check the bits in the d_i columns, you'll see that the binary number $d_3d_2d_1d_0$ corresponds to the decimal number in the left-hand column.

The numbers at the tops of the columns represent the bit positions of the pi and d_i bits. That number is displayed both in decimal and in binary.

In the table, p_0 is chosen so that the parity of the bits in positions $1, 3, 5$ and 7 is even: in other words, the number of 1 bits is even. Check a few rows to see that this is the case. Similarly, p_1 is chosen so that the parity of the bits in positions $2, 3, 6$ and 7 have even parity. (The numbers $2, 3, 6$ and 7 are the bit positions that have a 1 in the middle position of their binary expression. Finally, bit p_2 is chosen so that the parity of the bits in positions $4, 5, 6$ and 7 are even (and these are the numbers with a 1 in their high-order binary expression).

Notice the advantage of the out-of-order bit labeling: if p_2 were the third bit, we'd need to know its value when computing p_0 and p_1. If we wanted to extend the scheme to more than 4 bits of useful data, we'd add p_3 in position 8, p_4 in position 16 and so on.

It's easy to build hardware that generates, for each of the 16 desired 4-bit patterns, the 7-bit transmission listed in the row corresponding to that data value, but what should happen on the receiving end?

Well, let s_0 be zero if and only if the parity of the set of bits $\{p_0, d_3, d_2, d_0\}$ is even. Similarly, s_1 is zero exactly when the parity of the set of bits $\{p_1, d_3, d_1, d_0\}$ is even, and s_2 is zero exactly when the parity of $\{p_2, d_2, d_1, d_0\}$ is even. Now consider the binary number $S = s_2s_1s_0$. It will represent a number from 0 through 7, inclusive. Clearly if it is zero, then all of the parity checks succeeded, and the transmission is error-free (at least according to our assumption that it is never the case that there are 2 or more errors). But the cool thing is this: if S is non-zero, then it is a number from 1 to 7 and that number corresponds to the bit that is in error!

Let's try an example. Suppose we desire to transmit the number (decimal) 12 so we send the following bit pattern: 0111100. But suppose data bit d_2 is flipped, so what arrives at the other end is 0111000. When we check the parities controlled by p_0, p_1 and p_2 we find that only the middle one is correct (even) so $S = 101$ (binary) $= 5$ decimal. That means position 5, or the d_2 bit is wrong.

Exercise: Check a few other one-bit errors when transmitting different numbers. See what happens if the error happens to be among the parity bits. What happens if there are 2 errors in the transmission? How about 3 or more?

If you answered the previous exercise, you found that no matter how many errors there are the algorithm will either decide that everything is ok, or it will correct a single bit so no matter what, the transmission will be accepted, possibly with a single correction, and the result will always be wrong. It would be great if we improved the method so that every single-bit error can be discovered and corrected, and if every two-bit error is at least detected so that we know the data is faulty.

This isn't hard to do with a small modification to the method above. Simply add one more parity bit (p_3 in our case) so that the parity of all 8 bits in the transmission is, say, even. As before, check the parity on the original 7 bits in the same way. There are only a few possibilities (this time assuming that at most 2 errors can occur):

1. 0 total errors: All four parity checks will succeed (in particular, $S = 0$), so no errors of any sort occurred.
2. 1 total error: The overall parity will be odd so if S is zero, then p_3 must be in error. If S is non-zero, then p_3 is correct and S points to the particular bit among the original 7 that needs to be corrected.
3. 2 total errors: The overall parity will be even but S will be non-zero, so we know there are (at least) 2 errors, and we can't recover.

Exercise: Using the formula $2^m \geq m+k+1$ make a list of the values of m corresponding to all values of k up to 1000. (A range of values of k will correspond to each m.) This, of course, will handle only single-bit error correction. One more bit is required if double-bit error detection is also required. If you are designing a computer with 64-bit words and you would like to add hardware so that each word has single-bit error correction and double-bit error detection, how many total bits will you need to store to have the 64 data bits?

6 Cryptography

The goal of cryptography is to make it possible for two people to communicate in such a way that other people cannot understand the messages. There is no end to the number of ways this can be done, but here we will be concerned with methods of altering the text in such a way that the recipient can undo the alteration and discover the original text.

The original text is usually called "cleartext" and the encoded or altered text is called "ciphertext". The conversion from cleartext to ciphertext is called "encoding" or "enciphering", and the opposite operation is called "decoding" or "deciphering". If you are trying to read a secret message that was not intended for you and you initially don't know the encoding method, it is called "cracking" the code. Sometimes the words "code" or "coding" refer to a system where there is a way to transform the message one word at a time instead of one letter at a time, but we will just use the words "code" and "cipher" interchangeably.

In general, the more ciphertext you have, the easier it is to crack the code. So if you are trying to keep your secrets it generally is a good idea to change the coding mechanism regularly. For example, if a coding scheme has a keyword (like the Vigenère cipher described below), if a different keyword is used every day, there may never be enough ciphertext to decode the message. But if you change the encoding every day, you need to have some method of getting the new keyword to the intended recipient in a secure way. The easiest way to crack a code is to steal the codebook!

There is a way around this that we'll discuss in the section on public key cryptography.

7 Simple Substitution Ciphers

In a simple substitution cipher, one character is substituted for another. Here is a simple example:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
R	Z	B	U	Q	K	F	C	P	Y	E	V	L	S	N	G	W	O	X	D	J	I	A	H	T	M

To encode some text, simply find each character in the text in the first line, and replace it by the character below it. For example, using the example above, if you encode the word "BIRDBRAIN", you get "ZPOUZORPS". To decode, reverse the process—for the first character in “ZPOUZORPS”, find “Z” in the lower line, look above it to get “B”—the first letter of “BIRDBRAIN”, et cetera.

If you have to decode a lot, it is easier if you invert the line above to get the table below. With this table it is much easier to decode since the letters in the encoded word are now in alphabetical order in the top line.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
W	C	H	T	K	G	P	X	V	U	F	M	Z	O	R	I	E	A	N	Y	D	L	Q	S	J	B

Exercise: Pick a partner. Each of you encode a short (4-words or fewer) message to the other using a cipher where each letter is shifted by 11 characters, wrapping around from Z to A. In other words, change “A” to “L” since “L” is 11 characters beyond “A”, and “B” to “M”, “C” to “N” and so on. The letter “T” will be represented by “E” since “Z” is 6 steps from “T” and then you wrap to “A”, “B”, “C”, “D” and finally “E” in 5 more steps. After you have encoded the messages exchange them with your partner and each of you decode the other’s work.

These simple substitution ciphers are fairly easy to “crack”—the problem is that in English (or any language), certain letters are far more likely to appear. In English, for example, the letter “E” is far more likely to appear than the letter “Z”. In fact, here is a list of the letters used in English arranged approximately in order of usage (“E” is the most used letter; “Z” is least). The approximate percentages for the first few letters in the list below are: E: 12.7%, T: 9.1%, A: 8.2%, O: 7.5%, and the percentages for the last few are approximately: J: 0.2%, Q: 0.1%, Z: 0.1%.

E T A O I N S H R D L U C M W F G Y P B V K X J Q Z

There is other information available about letter frequency as well. For example, here is a list of the most common initial letters in a word:

T A H W I S O M B C F

Here is a list of the most common terminal letters in a word:

E D S T N R Y O F G H

Here is a list of the most common digraphs:

TH HE IN ER AN RE ND HA ON OU ED

Finally, here is a list of the most common double letters:

LL EE SS OO TT RR PP FF NN CC MM

Following is a short passage encoded with a simple substitution cipher. This is typical of problems that appear in puzzle books or in daily newspapers. In fact, this one is probably even easier, since there is so much ciphertext:

```
  UJEJVZR QFEYGE, SV SO OFSU, JWIG FEESTGU FV VZG
UJJE JC FW FQFEVLGWV SW PZSIZ F NASVVGESWN QFEVR
PFO VFYSWN QAFIG.  FV QEGISOGAR VZG OFLG LJLGWV,
F XGFKVSCKA XKV TFIKJKO OZJPNSEA FEESTGU FV VZG
UJJE.
  CJE F LJLGWV, VZGEG PFO ZGOSVFVSJW JW XJVZ OSUGO,
FWU VZGW VZG OZJPNSEA OVGQQGU XFIY VJ LFYG PFR,
OFRSWN, "FNG XGCJEG XGFKVR."
  "WJV FV FAA!" OFSU UJEJVZR QFEYGE, OFSASWN
VZEJKNZ. "QGFEAO XGCJEG OPSWG!"
```

To try to crack this cipher, begin by counting the number of occurrences of each letter, and we obtain the following counts:

A	B	C	D	E	F	G	H	I	J	K	L	M
10	0	5	0	24	35	34	0	6	24	7	7	0
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
9	18	7	9	7	22	3	11	31	16	7	5	16

Since the text sample was relatively small, we can't be certain that the most common letter ("F" in the sample above) stands for the letter "E", but it's a pretty good bet that you'll find "E" among the letters "F", "G", and "V".

The structure of English gives plenty of other clues as well. For example, the word "F" appears twice in the text, so "F" must stand for "I" or "A". Since "F" is very common in the sample above, it is more likely to stand for "A", since "A" is much more common in English than "I". So the first guess you might make is that "F" stands for "A". Now the word "FV" appears in the text twice, and "V" is also very common. "AT" is a word in English, so perhaps "V" stands for "T" in the cipher. Now these are just guesses, but they are not bad guesses.

Making those substitutions gives us the following:

```
      T    A       T     A         A       AT T
  UJEJVZR QFEYGE, SV SO OFSU, JWIG FEESTGU FV VZG
        A  A A T   T          A    TT       A T
UJJE JC FW FQFEVLGWV SW PZSIZ F NASVVGESWN QFEVR
 A  TA       A     AT           T    A        T
PFO VFYSWN QAFIG.  FV QEGISOGAR VZG OFLG LJLGWV,
A   A T       T  A               A       AT T
F XGFKVSCKA XKV TFIKJKO OZJPNSEA FEESTGU FV VZG

UJJE.
      A      T  T      A      TAT         T
  CJE F LJLGWV, VZGEG PFO ZGOSVFVSJW JW XJVZ OSUGO,
A   T    T             T       A   T   A    A
FWU VZGW VZG OZJPNSEA OVGQQGU XFIY VJ LFYG PFR,
 A       A              T
OFRSWN, "FNG XGCJEG XGFKVR."
     T AT A      A       T    A       A
  "WJV FV FAA!" OFSU UJEJVZR QFEYGE, OFSASWN
T           A
VZEJKNZ. "QGFEAO XGCJEG OPSWG!"
```

Looking at the text above, there are a lot more clues. For one thing, in the first line is the word "SV", where the "V" may stand for "T". The only two words in English ending in "T" are "AT" and "IT", but we've already guessed that "F" stands for "A", so "S" is probably "I". Also, since we think we know what letters stand for "T" and "A", the other extremely common letter, "G", probably stands for "E". Finally, in the next-to-last line is the word "FAA"—a three letter word beginning with "A". In English, "A" must be "L", "D", or "S", but "at all" makes much more sense than "at add" or "at ass", so "A" is probably "L":

```
      T    A  E   IT I   AI      E A  I E  AT T E
  UJEJVZR QFEYGE, SV SO OFSU, JWIG FEESTGU FV VZG
        A  A A T E T I    I   A  LITTE I    A T
UJJE JC FW FQFEVLGWV SW PZSIZ F NASVVGESWN QFEVR
 A  TA I    LA E   AT   E I EL  T E  A E    E T
```

```
PFO VFYSWN QAFIG.  FV QEGISOGAR VZG OFLG LJLGWV,
A  EA TI  L   T  A           I L A  I E  AT T E
F XGFKVSCKA XKV TFIKJKO OZJPNSEA FEESTGU FV VZG

UJJE.
      A    E T  T E E  A   E ITATI        T   I E
  CJE F LJLGWV, VZGEG PFO ZGOSVFVSJW JW XJVZ OSUGO,
A   T E  T E      I L  TE  E   A   T   A E  A
FWU VZGW VZG OZJPNSEA OVGQQGU XFIY VJ LFYG PFR,
 A I     A E  E   E  EA T
OFRSWN, "FNG XGCJEG XGFKVR."
     T AT ALL    AI      T    A  E    AILI
  "WJV FV FAA!" OFSU UJEJVZR QFEYGE, OFSASWN
T          EA L   E   E   I E
VZEJKNZ. "QGFEAO XGCJEG OPSWG!"
```

From here, it’s easy to make progress. In the next-to-last line, “WJV FV FAA” is almost certainly “NOT AT ALL”, so “W” is “N” and “J” is “O”. Similarly, the word “VZG” is almost certainly “THE”, so “Z” is “H”. Thus we obtain:

```
   O OTH   A  E   IT I   AI   ON E A  I E  AT THE
  UJEJVZR QFEYGE, SV SO OFSU, JWIG FEESTGU FV VZG
 OO  O  AN A A T ENT IN  HI H A  LITTE IN   A T
UJJE JC FW FQFEVLGWV SW PZSIZ F NASVVGESWN QFEVR
 A  TA IN   LA E   AT   E I EL  THE  A E  O ENT
PFO VFYSWN QAFIG.  FV QEGISOGAR VZG OFLG LJLGWV,
A  EA TI  L   T  A  O    HO  I L A  I E  AT THE
F XGFKVSCKA XKV TFIKJKO OZJPNSEA FEESTGU FV VZG
 OO
UJJE.
   O  A  O ENT  THE E  A  HE ITATION ON  OTH  I E
  CJE F LJLGWV, VZGEG PFO ZGOSVFVSJW JW XJVZ OSUGO,
AN  THEN THE  HO  I L  TE  E   A   TO  A E  A
FWU VZGW VZG OZJPNSEA OVGQQGU XFIY VJ LFYG PFR,
 A IN    A E  E O E  EA T
OFRSWN, "FNG XGCJEG XGFKVR."
   NOT AT ALL    AI   O OTH   A  E    AILIN
  "WJV FV FAA!" OFSU UJEJVZR QFEYGE, OFSASWN
TH O  H    EA L   E O E   INE
VZEJKNZ. "QGFEAO XGCJEG OPSWG!"
```

From what we have above, “FWU” is clearly “AND”, so “U” codes for “D”, “ZGOSVFVSJW” is “HESITATION”, so “O” codes for “S”, “VZGEG” is either “THESE” or “THERE”, but “S” is used, so “E” codes for “R”:

```
  DOROTH   AR ER  IT IS SAID  ON E ARRI ED AT THE
  UJEJVZR QFEYGE, SV SO OFSU, JWIG FEESTGU FV VZG
DOOR O  AN A ART ENT IN  HI H A  LITTERIN   ART
UJJE JC FW FQFEVLGWV SW PZSIZ F NASVVGESWN QFEVR
 AS TA IN   LA E   AT  RE ISEL  THE SA E  O ENT
PFO VFYSWN QAFIG.  FV QEGISOGAR VZG OFLG LJLGWV,
A  EA TI  L   T  A  O   SHO  IRL ARRI ED AT THE
```

```
F XGFKVSCKA XKV TFIKJKO OZJPNSEA FEESTGU FV VZG
DOOR
UJJE.
   OR A  O ENT  THERE  AS HESITATION ON  OTH SIDES
  CJE F LJLGWV, VZGEG PFO ZGOSVFVSJW JW XJVZ OSUGO,
AND THEN THE SHO  IRL STE  ED  A   TO  A E  A
FWU VZGW VZG OZJPNSEA OVGQQGU XFIY VJ LFYG PFR,
SA IN    A E  E ORE  EA T
OFRSWN, "FNG XGCJEG XGFKVR."
   NOT AT ALL   SAID DOROTH   AR ER  SAILIN
  "WJV FV FAA!" OFSU UJEJVZR QFEYGE, OFSASWN
THRO  H    EARLS  E ORE S INE
VZEJKNZ. "QGFEAO XGCJEG OPSWG!"
```

From here, it's easy. Fill in the obvious letters for a couple of passes to obtain the final decryption:

```
  DOROTHY PARKER, IT IS SAID, ONCE ARRIVED AT THE
  UJEJVZR QFEYGE, SV SO OFSU, JWIG FEESTGU FV VZG
DOOR OF AN APARTMENT IN WHICH A GLITTERING PARTY
UJJE JC FW FQFEVLGWV SW PZSIZ F NASVVGESWN QFEVR
WAS TAKING PLACE.  AT PRECISELY THE SAME MOMENT,
PFO VFYSWN QAFIG.  FV QEGISOGAR VZG OFLG LJLGWV,
A BEAUTIFUL BUT VACUOUS SHOWGIRL ARRIVED AT THE
F XGFKVSCKA XKV TFIKJKO OZJPNSEA FEESTGU FV VZG
DOOR.
UJJE.
  FOR A MOMENT, THERE WAS HESITATION ON BOTH SIDES,
  CJE F LJLGWV, VZGEG PFO ZGOSVFVSJW JW XJVZ OSUGO,
AND THEN THE SHOWGIRL STEPPED BACK TO MAKE WAY,
FWU VZGW VZG OZJPNSEA OVGQQGU XFIY VJ LFYG PFR,
SAYING, "AGE BEFORE BEAUTY."
OFRSWN, "FNG XGCJEG XGFKVR."
  "NOT AT ALL!" SAID DOROTHY PARKER, SAILING
  "WJV FV FAA!" OFSU UJEJVZR QFEYGE, OFSASWN
THROUGH. "PEARLS BEFORE SWINE!"
VZEJKNZ. "QGFEAO XGCJEG OPSWG!"
```

7.1 Improving a Substitution Cipher

There are plenty of ways to improve on the simplest form of substitution cipher as presented above. This section lists some obvious (and not so obvious) things you can do.

Most important, get rid of the spaces, punctuation, et cetera, or at the very least, encode them with alternative symbols as well. Just an encoding of the spaces won't help much—there are vastly more spaces in any document than any other character, so the one that stands for a space will be obvious.

But you can fix that problem (and the frequency problem in general) by having duplicate ciphers. For example, don't restrict yourself to just 26 output symbols. (For example, imagine that your encoding will consist of integers between 0 and 999.) Then instead of having one number stand for "E", have a whole bunch of them stand for it so that the frequencies balance out a bit. Using the frequencies of letters in English we stated earlier (E: 12.7%, T: 9.1%, A: 8.2%, O: 7.5%, . . . , J: 0.2%, Q: 0.1%, Z: 0.1%), we'd have 127 different symbols for "E", 91 for "T", and 1 for "Q" and for “Z”. If we do this, the person trying to break the code will find that all the symbols will be approximately equally common, and it will be much harder to get started.

Remembering what we learned about entropy and information previously, since you want to in some sense make the uncertainty as high as possible, make the frequencies of the encoding letters as equal as possible.

But such a code is still subject to an attack on frequencies. For example, if the codebreaker has access to a lot of text, letter pair frequencies can be used. For example, in English, whenever there is a “Q” in the text, you can be virtually certain that it will be followed by a “U” (except if you’re talking about Iraq). Similarly “TH” and “EN” are very common, and you hardly ever see “BD” (bdelium). So even an augmented cipher as described in the previous paragraph can be attacked with a frequency analysis.

Of course there’s nothing to prevent the code maker from having encodings for letter pairs, triplets, common words, et cetera to make the task still more difficult. The disadvantage is that as you add more combinations, the ciphers become more and more difficult to decode. If you are using a computer program to encode and decode, however, this kind of added complexity isn’t a big deal.

Another option along the same lines is simply to add “nulls” to the cipher. Encodings that are just garbage and should be tossed out during the decoding. These can be used to balance frequencies in a nice way.

And of course you can add special items that mean things like “ignore the next item”, or “delete the previous item”. But in spite of all the suggestions above, if you have a sufficient amount of text encoded with a simple substitution cipher, it is only a matter of time before someone could break it.

A computer program can help with the attack in various ways. Such a program can know things like the following:

- The frequencies of all the letters in English
- The frequencies of pairs and triples of letters in English
- A list (with frequencies) of thousands of English words.
- There may even be other rules; for example, that if a noun begins with a consonant, it should be preceded by an “an” rather than an “a” which would be used in front of a noun beginning with a vowel. Or it could know the possible parts of speech of various words and frequencies of pairs. For example, in English, adjectives usually come before nouns (“white house” rather than “house white”). In other languages, different rules apply (“casa blanca” instead of “blanca casa”).
- With the ciphertext, the computer can also be told of any words/phrases that are expected to appear in the text.

With the data above, as the machine makes various guesses about the encoding of a character, it can instantly check the rest of the text to make sure that such a substitution makes sure that all or almost all of the rest of the sequences appear in valid English words.

Here’s an example of a straight substitution cipher that you can try to break:

```
  NOT NUA JMPETZ UTST ZENNELV EL NOT

JEGELV SAAW, UMENELV KAS NOTES

OAZNTZZ, UOA UMZ ZJEVONJX PTJMXTP.

NOT PMCVONTS AK NOT KMWEJX UMZ UENO

NOTW, AL NOT NOTASX NOMN ZOT UACJP
```

```
QTTY NOT GEZENASZ ADDCYETP PCSELV

NOT UMEN.

  NOT DOEJP UMZ YTSOMYZ ZEI XTMSZ

AJP, ZLCR LAZTP, KSTDQTJP, RCDQ

NAANOTP MLP RTZYTDNMDJTP.  ZOT

WMELNMELTP M PTTY ZEJTLDT MLP

NOT NUA JMPETZ YTTSTP PACRNKCJJX

MN OTS.

  KELMJJX, ALT AK NOTW WCNNTSTP

NA NOT ANOTS, "LAN GTSX Y-S-T-N-N-X,

E KTMS," DMSTKCJJX ZYTJJELV NOT QTX

UASP.

  UOTSTCYAL NOT DOEJP YEYTP CY,

"RCN MUKCJ Z-W-M-S-N!"
```

To save you some time, here are the character frequencies for the passage above:

A	B	C	D	E	F	G	H	I	J	K	L	M
25	0	14	10	28	0	3	0	1	23	10	18	25
N	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
44	31	24	4	5	22	60	14	7	7	11	13	23

The answer appears in Appendix A.0.3.

8 Permutations

Another obvious method to encode a message is to jumble the letters in some way so that the recipient can simply unjumble them. Here's a very simple way to do it that's not very secure, but it shows you the idea.

First, break the message up into chunks of 25 letters. If there is some left over at the end, add enough junk to extend the last chunk to be 25 letters, too. Each of the chunks is then encoded by itself. Here's an example of the encoding of a very short message: "WHO IS THAT CUTE GIRL?".

First off, it is not even 25 characters long, so let's extend it (I'll use the letter "X" for the extension, but it would be much better to use more innocent-looking letters. Here's the 25 character chunk: "WHOISTHATCUTEGIRL?XXXXXXX". (Obviously, you wouldn't use repeated "X" characters; you'd use common English letters, but we use "X" here to show the idea.)

Next, lay out the letters in a 5×5 square as follows (obviously it would be better to leave out the the punctuation):

W	H	O	I	S
T	H	A	T	C
U	T	E	G	I
R	L	?	X	X
X	X	X	X	X

Then just read out the text by columns instead of rows, so the encoding of the phrase becomes: "WTURXH-HTLXOAE?XITGXXSCIXX". To decode the message, simply write characters in the columns of a 5×5 grid and read the message out of the rows.

If this is the only thing that's been done, it is very easy to crack the encoding. For one thing, when you check the frequency, you notice that it is the same as in standard English, so it may just be a transposition code. But there is nothing to prevent you from doing a substitution cipher first, and then encoding as above. The nice thing about the above jumbling is that it will destroy any letter pair or letter triple frequencies.

9 Vigenère Cipher

A Vigenére Cipher is just a mixture of the 26 different ciphers shown in the table below:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

The 26 ciphers are called "A", "B", et cetera, and are listed in the first column. If you wish to look up the letter "G" in the cipher "C", go to the third row (labelled "C") and look up the letter under the "G". The result is "I". The encoding of the word "DILBERT" using the "F" cipher is "INQGJWY".

The Vigenère cipher uses different individual ciphers for each letter. The usual method is to agree upon a keyword with the person you wish to receive the message. Let's use "FROGLEGS" as our keyword for this example. Now suppose you wish to encode the phrase "ONCE UPON A MIDNIGHT DREARY".

Begin by writing "FROGLEGS" repeatedly over the phrase:

```
F R O G  L E G S  F  R O G L E G S F  R O G L E G
O N C E  U P O N  A  M I D N I G H T  D R E A R Y
```

To perform the encoding, we use the "F" cipher on the "O" of "ONCE", the "R" code on the "N", the "O" code on the "C", and so on, yielding:

TEQK FTUF F DWJYMMZY UFKLVE

To make sure you understand what is going on, decipher the following phrase using the Vigenère cipher with the keyword CRYPTOGRAPHY:

HFSGLQUIE PUB UVTTG MKRRH HEQ

Cracking a code like this is a bit more difficult than cracking a simple substitution code, but it is certainly possible. The longer the keyword, of course, the more difficult it is to crack the code. If the keyword were infinitely long, and random, it would be impossible to decode, since any decoding is as likely as any other. But when this sort of code is used, typically the keyword is a real English word or phrase so that it is not too hard to remember.

With a computer to aid you, if you know a cipher is of this sort and that the keyword is an English word (of which a pretty good list would consist of $100,000$ words or so), you can simply have the computer try all possible keywords, and for modern machines, it won't take long to check "only" $100,000$ cases.

A very long keyword of random letters is unbreakable, and here's why. Any coding can represent any text with some keyword. For example, suppose the message is "WEASEL". With an appropriate "keyword", it can represent anything that's 6 characters long. Suppose you want "WEASEL" to represent "TURNIP"—just use the table and figure out what the right keyword to do this would be. "W" maps to "T", so the first letter of the keyword is "X". "E" maps to "U", so the second letter of the keyword has to be "Q", and so on. Continuing in this way, we discover the keyword is "XQRVEE". Similarly, if the keyword were "HNAOKC", then "WEASEL" represents "DRAGON".

Clearly, if the key is allowed to be arbitrarily long and composed of arbitrary letters, then anything can stand for anything, and hence the code is completely secure. But the first time you reuse the key, you give the code breaker some information that can help. If the key is 1000 characters long, if you sent a message of a million characters, the key would have to be reused 1000 times, and it would not be at all secure.

This last situation can be partially achieved using the following trick: Begin with some keyword and after its letters are exhausted, begin using the letters from the message itself to produce a sort of "keyword" that is as long as necessary. Of course if your opponent knows you are doing this, cracking the code is no more difficult: just try all possible initial keywords and see what comes out in English. To illustrate the method, suppose that the original keyword is "COW" and the message to be transmitted is "ATTACK TOMORROW." Here's how to set up the encoding:

```
C O W A T T  A C K T O M O R
A T T A C K  T O M O R R O W
```

To decode it, of course, begin decoding as previously with the "COW" keyword, and when you're done, you have the first three characters of the message which can be used to decode the next three characters, et cetera.

Exercise: Show that the encoded text for the example above will be: "CHPAVD TQWHFDCN."

In the examples above we have left the space character as a space which makes life much easier for someone trying to crack the code. If we simply expand the table to include the space character in the scheme above, our opponent will have a more difficult problem.

10 Converting Text to Numbers

Some of the more mathematical methods of encryption that we'll discuss are best described in terms of transformations of integers into other integers. For the more interesting encodings, it really doesn't matter how you do this translation, but certain methods have slight advantages over others.

The most common method is probably the ASCII encoding that assigns a number to each character. The standard ASCII encoding specifies 128 different characters (including not only the standard upper and lower case letters and the digits, but all the punctuation, some control characters, and various other things.

Here is the standard ASCII encoding. The numbers on the sides and top are in octal (base 8) and need to be combined. For example, the character "S" is in row 12, column 3. It's ASCII octal number is thus 123. To convert to base 10, 123 (octal) $= 1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 = 64 + 16 + 3 = 83$ (decimal). 040 (octal) is the space character; 177 (octal) is the "delete" character.

	0	1	2	3	4	5	6	7
00	ˆ@	ˆA	ˆB	ˆC	ˆD	ˆE	ˆF	ˆG
01	ˆH	ˆI	ˆJ	ˆK	ˆL	ˆM	ˆN	ˆO
02	ˆP	ˆQ	ˆR	ˆS	ˆT	ˆU	ˆV	ˆW
03	ˆX	ˆY	ˆZ	ˆ[	ˆ\	ˆ]	ˆˆ	ˆ_
04		!	”	#	$	%	&	’
05	(	)	*	+	,	-	.	/
06	0	1	2	3	4	5	6	7
07	8	9	:	;	<	=	>	?
10	@	A	B	C	D	E	F	G
11	H	I	J	K	L	M	N	O
12	P	Q	R	S	T	U	V	W
13	X	Y	Z	[	\	]	ˆ	_
14	‘	a	b	c	d	e	f	g
15	h	i	j	k	l	m	n	o
16	p	q	r	s	t	u	v	w
17	x	y	z	{	|	}	∼	DEL

The ASCII assignments are for the numbers from 0 to 127, which require 7 bits of data. The standard character on a computer (one byte) is 8 bits of data, which can represent a number from 0 to 255. When ASCII encoding is used, each character is put in one byte, so effectively, one bit of each character is wasted.

But the nice thing about a numeric representation that packs into 7 or 8 bits is that the numeric representations can simply be concatenated to make representations of groups of letters. If you wish to encode two ASCII characters at once, simply place their binary representations next to each other, and you get a 16 bit number (between 0 and 65535). If you don't understand this, imagine that you are working in base 10, and you have a character set that encodes to a number between 00 and 99. Then if "A" happened to be 17 and "Z" were 42, then the character pair "AZ" would be the four-digit number 1742.

Binary representations work great on a computer, but since many people find base 10 much easier to work with than base 2, the numeric examples in this paper will use the following encoding:

	0	1	2	3	4	5	6	7	8	9
0	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX
1	SP	A	B	C	D	E	F	G	H	I
2	J	K	L	M	N	O	P	Q	R	S
3	T	U	V	W	X	Y	Z	a	b	c
4	d	e	f	g	h	i	j	k	l	m
5	n	o	p	q	r	s	t	u	v	w
6	x	y	z	.	,	:	;	’	“	‘
7	!	@	#	$	%	ˆ	&	*	-	+
8	(	)	[	]	{	}	?	/	<	>
9	0	1	2	3	4	5	6	7	8	9

The entries with XX are not used, and “SP” is the space character. Thus all characters code to between 10 and 99. “U” is 31; “5” is 95, and so on. If we want to encode four characters at a time, for example, we would encode the word “C3PO” as the 8-digit number 13932625.

11 Generating Long Keys

What you would like to have is a very long key to use in a scheme like the Vigenére encoding, but as we stated above, the problem with a long key is that you have to transmit that key to the persons who should be able to decode the messages in some secure manner.

The more “random” the key is, the more secure it is. To make a really random key, you could imagine using something like a timer connected to a Geiger counter and placing it close to some radioactive source so you could use the intervals between atomic decays to generate numbers that are truly random.

But if you’re willing just to have a sequence that looks very random, there are many techniques for generating such sequences of so-called “pseudo-random numbers”. Volume 2 of Donald Knuth’s classic book, “The Art of Computer Programming—Seminumerical Algorithms”, contains a treasure-trove of information about how to generate pseudo-random sequences. We will look at just a couple of methods here to give you an idea of the possibilities.

A very easy way to do this (but not a very good one) is based on the fact that if p is a prime number, and n is any number with $0 < n < p$, then the sequence of numbers $kn (\text{mod } p)$ cycle through all the numbers between 1 and $p-1$ without missing any of them, and in an order that appears somewhat random as k goes through $1, 2, 3, \dots, p-1$.

For example, if $p = 13$ and $n = 5$:

$$\begin{array}{ccc}
5 \cdot 1 (\text{mod } 13) = 5 & 5 \cdot 2 (\text{mod } 13) = 10 & 5 \cdot 3 (\text{mod } 13) = 2 \\
5 \cdot 4 (\text{mod } 13) = 7 & 5 \cdot 5 (\text{mod } 13) = 12 & 5 \cdot 6 (\text{mod } 13) = 4 \\
5 \cdot 7 (\text{mod } 13) = 9 & 5 \cdot 8 (\text{mod } 13) = 1 & 5 \cdot 9 (\text{mod } 13) = 6 \\
5 \cdot 10 (\text{mod } 13) = 11 & 5 \cdot 11 (\text{mod } 13) = 3 & 5 \cdot 12 (\text{mod } 13) = 8
\end{array}$$

The numbers cycle through the series: $5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, 8$. This sequence has the vague appearence of being random. If you choose a larger prime, the sequence will be longer.

Using such a sequence, let’s encrypt a message using the code described in the previous section. The message will be:

PASSWORD: ELEPHANT

Including the space, the numeric codes for these letters are:

26, 11, 29, 29, 33, 25, 28, 14, 65, 10, 15, 22, 15, 26, 18, 11, 24, 30

Now we’ll use a combination of the method above to generate a sequence of pseudo-random numbers to use as a sort of Vigenére key. We’ll let $p = 16139$ and $n = 4352$. As k goes from 1 to 18 (there are 18 characters in the message above), $kn (\text{mod } 16139)$ goes through the following 18 numbers:

4352, 8704, 13056, 1269, 5621, 9973, 14325, 2538, 6890, 11242, 15594, 3807, 8159, 12511, 724, 5076, 9428, 13780

We will just encode our numbers by adding these pseudo-random numbers to the numbers in our sequence and taking the result modulo 100. For example, the first letter, 26, is converted to (26+4352) $(\bmod\ 100)$ = 78. The encoded sequence becomes:

78, 15, 85, 98, 54, 98, 53, 52, 55, 52, 09, 29, 74, 37, 42, 87, 52, 10

Assuming that the person who wishes to decode the message knows the values of p and n, he can generate the exact same sequence of keys, and can undo the encryption above. For example, to decrypt the first character, he generates the first number in the key sequence, 4352, subtracts that from 78 (giving -4274), and taking that result modulo 100 to get 26.

Now notice that the only information that has to be passed to the recipient as the key is the pair of numbers p and n—there is no need to transmit a long sequence of keys.

11.1 Better Pseudo-Random Sequences

The sequence above is too simple, but with just a tiny modification, it can be made much more interesting. The following sequence can work quite well.

Choose a (usually large) prime number p and two integers a and c. In addition, choose a "seed", or starting number for the random sequence which we will call X_0 such that $0 <= X_0 < p$. Here is the formula for X_i, if $i > 0$:

$$X_i = (aX_{i-1} + c)(\bmod\ p).$$

So, for example, if $p = 16139$, $a = 91$, and $c = 541$, and we begin with $X_0 = 11111$, we generate the following sequence:

$$\begin{aligned}
X_0 &= 11111 \\
X_1 &= (91 \cdot 11111 + 541)(\bmod\ 16139) = 11024 \\
X_2 &= (91 \cdot 11024 + 541)(\bmod\ 16139) = 3107 \\
X_3 &= (91 \cdot 3107 + 541)(\bmod\ 16139) = 8915 \\
X_4 &= (91 \cdot 8915 + 541)(\bmod\ 16139) = 4856 \\
X_5 &= (91 \cdot 4856 + 541)(\bmod\ 16139) = 6684
\end{aligned}$$

which you can continue as long as you want. The first pseudo-random numbers in this sequence are: 11111, 11024, 3107, 8915, 4856, 6684. Of course the sequence has to cycle in p or fewer steps.

As an exercise, try to decode the following message based on $p = 16139$, $a = 91$, and $c = 541$, but with the starting value $X_0 = 0$. We will use the random sequence in the same way we did above—we converted our message to numbers between 10 and 99 using the table in Section 10, then we added successive keys and took the result modulo 100. Here is the resulting encoded message:

23, 52, 85, 91, 15, 06, 53, 61, 30, 72, 23

To get you started, the first number in the sequence is zero, so $X_0 = 0$, so the first number decodes as $(23 - 0)(\bmod\ 100) = 23$, so the first letter in the message is "M". Then

$$X_1 = (91 \cdot X_0 + 541)(\bmod\ 16139) = 541(\bmod\ 16139) = 541,$$

so the second letter is $(52 - 541)(\bmod\ 100) = 11$, so the second letter is "A". Continue in the same way.

12 XOR and Binary

Since data that's stored in computers and transmitted over wires is usually encoded as binary numbers, we can think of the data in a message as just a long sequence of 0's and 1's. If we could somehow have a totally-random sequence of 0's and 1's that is available both to the transmitter and recipient and would only be used once, then we would have a totally secure method to transmit messages, as follows.

Generate a binary message, and align it bit for bit with the random binary sequence. Perform the binary operation XOR on every pair and transmit that to the recipient. To decode, the recipient does exactly the same thing: she aligns the encoded message bit for bit with the random sequence and performs XOR on the pairs to produce the original message.

XOR stands for "exclusive or" and works as follows: If the incoming bits are the same (either 00 or 11) the result is 0. If they are different (01 or 10) the result is 1. It is easy to prove that for every possible selection of bits A and B that $(A \text{ XOR B}) \text{ XOR B} = \text{A}$. There are only four possibilities to check.

Exercise: Show that XOR behaves as described above.

Without a copy of the random sequence, any sequence of bits could be translated with equal likelihood into any other, so this method is totally secure, assuming you have a totally secure method get a copy of the sequence to both parties, and to assure that the sequence is totally random. If you generated it using one of the pseudo-random methods described above, and your opponent guesses that you are using this method, then it is not secure at all.

13 The German "Enigma" Code

During the second world war, the German military used a special encrytion machine called "Enigma" to encode its messages. Basically, the Enigma machine generated, given a "seed", a sequence of numbers that appeared sufficiently random to make them very difficult to crack, even if the internal details of the machine were known.

The allies had captured some of the Enigma machines, so they did know the internal workings, but much of the foundations of modern computer science were developed in an attempt to crack the German messages (usually with a fair amount of success).

It's beyond the scope of this paper to describe the internals of the Enigma machine and encoding technique, but here are a couple of references. See "The Code Book", by Simon Singh, or take a look at the web site:

`http://www.gl.umbc.edu/~lmazia1/Enigma/enigma.html`

14 Public Key Cryptography

One of the biggest practical problems in cryptography is the distribution of keys. Suppose you live in the United States and want to pass information secretly to your friend in Europe. If you truly want to keep the information secret, you need to agree on some sort of key that you and he can use to encode/decode messages. But you don't want to keep using the same key, or you will make it easier and easier for others to crack your cipher. Or suppose you want to change the key every day and get the new keys to every one of your submarines. If you print a list of keys, one for each day, and have it on all the submarines, if one ever gets captured, then all of the lists on all the other submarines are useless. And if you don't know that it was captured, you're in even bigger trouble.

But it's also a pain to get keys to your friend. If you mail them, they might be stolen. If you send them cryptographically, and someone has broken your code, that person will also have the next key. If you have to go to Europe regularly to hand-deliver the next key, that is also expensive. If you hire some courier to deliver the new key, you have to trust the courier, et cetera.

14.1 Trap-Door Ciphers

But imagine the following situation. Suppose you have a special method of encoding and decoding that is "one way" in a sense. Imagine that the encoding is easy to do, but decoding is very difficult. Then anyone in the world can encode a message, but only one person can decode it. Such methods exist, and they are called "one way ciphers" or "trap door ciphers".

Here's how they work. For each cipher, there is a key for encoding and a different key for decoding. If you know the key for decoding, it is very easy to make the key for encoding, but it is almost impossible to do the opposite—to start with the encoding key and work out the decoding key.

So to communicate with your friend in Europe, each of you has a trap door cipher. You make up a decoding key D_a and generate the corresponding encoding key E_a. Your friend does exactly the same thing, but he makes up a decoding key D_b and generates the corresponding encoding key E_b. You tell him E_a (but not D_a) and he tells you E_b (but not D_b). Then you can send him messages by encoding using E_b (which only he can decode) and vice-versa—he encodes messages to you using E_a (which only you can decode, since you're the only person with access to D_a).

Now if you want to change to a new key, it is no big problem. Just make up new pairs and exchange the encoding keys. If the encoding keys are stolen, it's not a big deal. The person who steals them can only encode messages—they can't decode them. In fact, the encoding keys (sometimes called "public keys") could just be published in a well-known location. It's like saying, "If you want to send me a private message, encode it using this key, and I will be the only person in the world who can read it." But be sure to keep the decoding key (the "private key") secret.

14.2 Certification

There is, of couse, a problem with the scheme above. Since the public keys are really public, anyone can "forge" a message to you. So your enemy can pretend to be your friend and send you a message just like your friend can—they both have access to the public key. Your enemy's information can completely mislead you. So how can you be certain that a message that says it is from your friend is really from your friend?

Here is one way to do it, assuming that you both have the public and private keys E_a, E_b, D_a, and D_b as discussed in the previous section. Suppose I wish to send my friend a message that only he can read, but in such a way that he is certain that the message is from me. Here's how to do it.

I will take my name, and pretend that it is an encoded message, and decode it using D_a. I am the only person who can do this, since I am the only person who knows D_a. Then I include that text in the real message I wish to send, and I encode the whole mess using E_b, which only my friend knows how to decode.

When he receives it, he will decode it using D_b, and he will have a message with an additional piece of what looks to him like junk characters. The junk characters are what I got by "decoding" my name. So he simply encodes the junk using my public key E_a and makes certain that it is my name. Since I am the only one who knows how to make text that will encode to my name, he knows the message is from me.

You can encode any text for certification, and in fact, you should probably change it with each message, but it's easy to do. Your message to your friend would look like this:

"Attack at dawn. Here is my decoding of 'ABCDEFG': 'JDLEODK'."

To assure privacy, for each message, change the "ABCDEFG" and the corresponding "JDLEODK".

15 Diffie-Hellman-Merkle Key Exchange

Sometimes this is just called "Diffie-Hellman" but Martin Hellman suggested that Ralph Merkle deserved a lot of the credit for this scheme. Details on the math used in this section can be found in Appendix B.

Here is the basic idea: Suppose Alice and Bob want to obtain the same (very large) key so that they can encode messages to each other, but they have to communicate over a transmission line that their opponents can read, so one can't simply send the large key to the other. How do they do this?

In a practical situation, *huge* numbers will be used, with hundreds of decimal digits. In the example, of course, we'll use smaller numbers to illustrate the method.

Suppose the desired key has about 100 decimal digits. Alice chooses two numbers, a prime number p (with about 100 digits) and a number g which is a primitive root $(\bmod p)$. In other words, g is a number such that all the numbers $g^1, g^2, g^3, \ldots, g^{p-1}$ are all different, modulo p (and of course this means that $g^{p-1} = 1 \bmod p$. Alice sends both of these numbers over the insecure transmission line, so that at the end, both she, Bob, and any opponents know the values of p and g.

The prime p should be huge, with hundreds of digits, but g can be small, even 2, if 2 is a primitive root modulo p.

Now, both Alice and Bob select their own secret integers (usually large) which they do not transmit to each other. Suppose Alice's number is a and Bob's is b. Alice calculates $A = g^a (\bmod p)$ and Bob calculates $B = g^b (\bmod p)$. Each sends their number to the other person, so everyone (including their opponents) know p, g, A and B.

Finally, Alice computes $S = B^a (\bmod p)$ and Bob calculates $T = A^b (\bmod p)$. It turns out that $S = T$ since:

$$S = B^a (\bmod p) = g^{ba} (\bmod p) = g^{ab} (\bmod p) = A^b = T.$$

Now that they both have a common key, they can communicate using that.

Since, under multiplication modulo p every number from 1 to $p-1$ has an inverse, encoding the number representing the message is simple. Suppose the message is $m < p$. To transmit the message secretly, simply calculate the product $mS (\bmod p)$ and send that. The recipient simply multiplies this by $S^{-1} (\bmod p)$ and the result is m.

It is not hard (for Bob and Alice) to calculate S^{-1}. Bob, for example, knows $S = g^{ab}, B = g^b, a, p$ and g. Let $G = p - 1$. We know that every number x between 1 and $p-1$ satisfies $x^G = 1 (\bmod p)$. To obtain the inverse of S, Bob does the following (and Alice can make a similar calculation):

$$(g^b)^{G-a} = g^{bG-ba} = (g^G)^b g^{-ab} = 1^b g^{-ab} = g^{-ab} = S^{-1}.$$

Let's look at an example (using, of course, tiny numbers compared to what would be used in a real-world situation). Suppose Alice chooses $p = 29$ and $g = 8$. (This means that their secret key will be a number between 1 and 28: not very secure, since an opponent will only have to try 28 possibilities to crack their code.)

The number 8 is a primitive root of root $((\bmod 29))$ since here are the values of 8^i for i ranging from 1 to 28 and they are all different:

$$\{8, 6, 19, 7, 27, 13, 17, 20, 15, 4, 3, 24, 18, 28, 21, 23, 10, 22, 2,\ 16, 12, 9, 14, 25, 26, 5, 11, 1\}$$

Now suppose that Alice's secret number $a = 5$ and Bob's is $b = 17$:

$$A = g^a (\bmod 29) = 8^5 (\bmod 29) = 32768 (\bmod 29) = 27,$$

and

$$B = g^b (\bmod 29) = 8^1 7 (\bmod 29) = 2251799813685248 (\bmod 29) = 10.$$

Now both Bob and Alice can compute the same secret number:

$$S = B^a (\bmod 29) = 10^5 (\bmod 29) = 100000 (\bmod 29) = 8,$$

and

$$T = A^b (\bmod 29) = 27^1 7 (\bmod 29) = 2153693963075557766310747 (\bmod 29) = 8.$$

That's because:

$$\begin{aligned} S &= T = 8^{5\cdot 17} (\bmod 29) = 8^{85} (\bmod 29) \\ &= 57896044618658097711785492504343953926634992332820282019728792003956564819968 (\bmod 29) \\ &= 8. \end{aligned}$$

Notice that even with the tiny prime number 29 we need to work with fairly huge numbers. Imagine what this would look like with p having hundred(s) of digits.

15.1 Group Communication

One nice feature of this scheme is that it is easy to extend the idea if you want to have three or more people communicating. Obviously, you could set up separate channels for each pair of communicators, but with more than three it would get unwieldy, and even with three, to send a message to the two others would require two different encodings.

But if Alice, Bob and Charlie want to communicate, they simply agree on a p and g, as before, and each selects their own secret number, a, b and c, respectively. Then g^a, g^b and g^c (all modulo-p) are passed around, then g^{ab}, g^{bc} and g^{ca} and finally, everyone can construct their own identical copy of g^{abc}, which is then used for mutual communication. If a fourth person, Debra, also wants to get into the act, a similar method can be used to construct g^{abcd}, et cetera.

Exercise: For the three-person scheme, how is the inverse of g^{abc} constructed by each of the three people?

16 RSA Encryption

OK, in the previous section we described what is meant by a trap-door cipher, but how do you make one? One commonly-used cipher of this form is called "RSA Encryption", where "RSA" are the initials of the three creators: "Rivest, Shamir, and Adleman". It is based on the following idea:

It is very simply to multiply numbers together, especially with computers. But it can be very difficult to factor numbers. For example, if I ask you to multiply together 34537 and 99991, it is a simple matter to punch those numbers into a calculator and 3453389167. But the reverse problem is much harder.

Suppose I give you the number 1459160519. I'll even tell you that I got it by multiplying together two integers. Can you tell me what they are? This is a very difficult problem. A computer can factor that number fairly quickly, but (although there are some tricks) it basically does it by trying most of the possible combinations. For any size number, the computer has to check something that is of the order of the size of the square-root of the number to be factored. In this case, that square-root is roughly 38000.

Now it doesn't take a computer long to try out 38000 possibilities, but what if the number to be factored is not ten digits, but rather 400 digits? The square-root of a number with 400 digits is a number with about 200 digits. The lifetime of the universe is approximately 10^{18} seconds – an 18 digit number. Assuming a computer could test one million factorizations per second, in the lifetime of the universe it could check 10^{24} possibilities. But for a 400 digit product, there are on the order of 10^{200} possibilties. This means the computer would have to run for 10^{176} times the life of the universe to factor the large number.

It is, however, not too hard to check to see if a number is prime—in other words to check to see that it cannot be factored. If it is not prime, it may be very difficult to factor, but if it is prime, it is not hard to show it is prime.

So RSA encryption works like this. I will find two huge prime numbers, p and q that have 100 or maybe 200 digits each. I will keep those two numbers secret (they are my private key), and I will multiply them together to make a number $N = pq$. That number N is basically my public key. It is relatively easy for me to get N; I just need to multiply my two numbers. But if you know N, it is basically impossible for you to find p and q. To get them, you need to factor N, which seems to be an incredibly difficult problem.

But exactly how is N used to encode a message, and how are p and q used to decode it? Below is presented a complete example, but I will use tiny prime numbers so it is easy to follow the arithmetic. In a real RSA encryption system, keep in mind that the prime numbers are huge.

In the following example, suppose that person A wants to make a public key, and that person B wants to use that key to send A a message. In this example, we will suppose that the message A sends to B is just a number. We assume that A and B have agreed on a method to encode text as numbers as was discussed in Section 10. Here are the steps:

1. Person A selects two prime numbers. We will use $p = 23$ and $q = 41$ for this example, but keep in mind that the real numbers person A should use should be *much* larger.

2. Person A multiplies p and q together to get $pq = (23)(41) = 943$. 943 is the "public key", which he tells to person B (and to the rest of the world, if he wishes).

3. Person A also chooses another number e which must be relatively prime to $(p - 1)(q - 1)$. In this case, $(p - 1)(q - 1) = (22)(40) = 880$, so $e = 7$ is fine. e is also part of the public key, so B also is told the value of e.

4. Now B knows enough to encode a message to A. Suppose, for this example, that the message is the number $M = 35$.

5. B calculates the value of $C = M^e (\text{mod } N) = 35^7 (\text{mod } 943)$.

6. $35^7 = 64339296875$ and $64339296875 (\text{mod } 943) = 545$. The number 545 is the encoding that B sends to A.

7. Now A wants to decode 545. To do so, he needs to find a number d such that $ed = 1(\text{mod } (p-1)(q-1))$, or in this case, such that $7d = 1(\text{mod } 880)$. A solution is $d = 503$, since $7 * 503 = 3521 = 4(880)+1 = 1(\text{mod } 880)$.

8. To find the decoding, A must calculate $C^d (\text{mod } N) = 545^{503} (\text{mod } 943)$. This looks like it will be a horrible calculation, and at first it seems like it is, but notice that $503 = 256 + 128 + 64 + 32 + 16 + 4 + 2 + 1$ (this is just the binary expansion of 503). So this means that

$$545^{503} = 545^{256+128+64+32+16+4+2+1} = 545^{256} 545^{128} \cdots 545^1.$$

   But since we only care about the result $(\text{mod } 943)$, we can calculate all the partial results in that modulus, and by repeated squaring of 545, we can get all the exponents that are powers of 2. For example, $545^2 (\text{mod } 943) = 545 \cdot 545 = 297025 (\text{mod } 943) = 923$. Then square again: $545^4 (\text{mod } 943) = (545^2)^2 (\text{mod } 943) = 923 \cdot 923 = 851929 (\text{mod } 943) = 400$, and so on. We obtain the following table:

$$\begin{aligned}
545^1 (\text{mod } 943) &= 545 \\
545^2 (\text{mod } 943) &= 923 \\
545^4 (\text{mod } 943) &= 400 \\
545^8 (\text{mod } 943) &= 633 \\
545^{16} (\text{mod } 943) &= 857 \\
545^{32} (\text{mod } 943) &= 795 \\
545^{64} (\text{mod } 943) &= 215 \\
545^{128} (\text{mod } 943) &= 18 \\
545^{256} (\text{mod } 943) &= 324
\end{aligned}$$

   So the result we want is:

$$545^{503} (\text{mod } 943) = 324 \cdot 18 \cdot 215 \cdot 795 \cdot 857 \cdot 400 \cdot 923 \cdot 545 (\text{mod } 943) = 35.$$

Using this tedious (but simple for a computer) calculation, A can decode B's message and obtain the original message $N = 35$.

16.1 RSA Exercise

OK, now to see if you understand the RSA decryption algorithm, suppose you are person A, and you have chosen as your two primes $p = 97$ and $q = 173$, and you have chosen $e = 5$. Thus you told B that $N = 16781$ (which is just pq) and you told him that $e = 5$.

He encodes a message (a number) for you and tells you that the encoding is 5347. Can you figure out the original message?

The answer appears in Appendix A.0.4.

A Solutions

A.0.1 Optimal 3-character encoding

We want to show that if $p_1 \geq p_2 \geq p_3$, then assigning the code 0 to C_1 and the codes 10 and 11 to C_2 and C_3 (in either order) generates the most efficient encoding; namely, that:

$$p_1(1) + p_2(2) + p_3(2)$$

is the smallest possible value. In other words, if we assign 2 bits to C_1, the results will always be worse. Here are the average bit lengths corresponding to the other ways we could make that assignment:

$$p_2(1) + p_1(2) + p_3(2)$$
$$p_3(1) + p_1(2) + p_2(2)$$

We just have to show that:

$$p_1(1) + p_2(2) + p_3(2) \geq p_2(1) + p_1(2) + p_3(2)$$

and

$$p_1(1) + p_2(2) + p_3(2) \geq p_3(1) + p_1(2) + p_2(2).$$

Both are similar; let's just do the first. The following inequalities are all equivalent; we just add or subtract the same thing from both sides to move from one to the next:

$$\begin{aligned} p_1(1) + p_2(2) + p_3(2) &\geq p_2(1) + p_1(2) + p_3(2) \\ p_1(1) + p_2(2) &\geq p_2(1) + p_1(2) \\ p_2(2-1) &\geq p_1(2-1) \\ p_2 &\geq p_1 \end{aligned}$$

Since the last line is true and they're all equivalent, we've shown the first inequality to be true. The second is similar.

A.0.2 Optimal 8-character encoding

Here is one optimal encoding:

i	C_i	p_i	encoding	b_i
1	A	0.28	00	2
2	B	0.27	01	2
3	C	0.23	10	2
4	D	0.09	110	3
5	E	0.04	11100	5
6	F	0.04	11101	5
7	G	0.03	11110	5
8	H	0.02	11111	5

and the average number of bits per character is:

$$p_1 \cdot b_1 + \cdots + p_8 \cdot b_8$$

$$= 0.28 \cdot 2 + 0.27 \cdot 2 + \cdots + 0.02 \cdot 5 = 2.48.$$

Of course any of the encodings with the same number of bits could be swapped around. You just need to find one where A, B and C are encoded in 2 bits, D in 3 bits, and E, F, G and H in 5 bits.

A.0.3 Cipher Solution

```
  THE TWO LADIES WERE SITTING IN THE
LIVING ROOM WAITING FOR THEIR
HOSTESS, WHO WAS SLIGHTLY DELAYED.
THE DAUGHTER OF THE FAMILY WAS WITH
THEM, ON THE THEORY THAT SHE WOULD
KEEP THE VISITORS OCCUPIED DURING THE WAIT.
  THE CHILD WAS PERHAPS SIX YEARS
OLD, SNUB NOSED, FRECKELD, BUCK
TOOTHED AND BESPECTACLED.  SHE
MAINTAINED A DEEP SILENCE AND
THE TWO LADIES PEERED DOUBTFULLY
AT HER.
  FINALLY, ONE OF THEM MUTTERED
TO THE OTHER, "NOT VERY P-R-E-T-T-Y,
I FEAR," CAREFULLY SPELLING THE KEY
WORD.
  WHEREUPON THE CHILD PIPED UP,
"BUT AWFUL S-M-A-R-T!"
```

A.0.4 RSA Solution

The answer for the RSA exercise is 16657

B Math for Cryptography

Here is some pure mathematics that is used in other parts of this article.

B.1 Prime and Composite Numbers

A positive integer is called “prime” if it cannot be written as the product of two smaller positive integers. Another way to say this is that a prime number is an integer greater than 1 that can only be written as the product of two positive integers if one of them is 1 and the other is the number. Thus 7 is prime and 6 is not, since $6 = 2 \cdot 3$. The first definition is a bit nicer since we don’t have to specifically exclude the number 1 (which can only be written as $1 \cdot 1$.

Here is a list of the first few prime numbers:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59.$$

A composite number is a number that can be written as the product of 2 or more prime numbers. (The number 1 is neither prime nor composite; the technical name for it is a “unit.”)

Here are some properties of prime and composite numbers, all stated without proof:

- There are an infinite number of prime numbers. In other words, prime numbers of arbitrary size exist, so if we need, say, “a prime number that is at least 100 digits long, we can find one.
- Every integer greater than 1 can be factored uniquely into prime numbers, where by “uniquely” we mean that every factorization has the same number of prime factors of each size. In other words, the ordering of the factors

doesn't matter, so although $12 = 2 \cdot 3 \cdot 2$ and $12 = 3 \cdot 2 \cdot 2$ appear to be different, they are the same, since both factorings have exactly two copies of the prime 2 and one copy of the prime 3. This fact is often called, "The fundamental theorem of arithmetic."

- A straight-forward but often inefficient way to check if a number is prime or not and if it is not, to determine its factors is to try to divide it by the integers from 2 up to the square-root of the number. (In fact, you only need to do these test divisions by prime numbers up to the square root, but sometimes you don't have a list of all such primes.) So to verify that, say, 101 is prime, we just try to divide it by $2, 3, 5$ and 7 and when none of them divides it evenly, we can conclude that it is prime. We can stop at 7, since the next prime, 11, is larger than the square-root of 101.
- There are mathematical methods that are computationally efficient that can tell you if a (usually large) number is prime or not, but if a large number happens to be composite, it seems to be very difficult to determine its factors. Although there are some tricks, the best-known methods currently known amount essentially to trying all the possibilities. This means that to find the factors of a 200-digit integer, we'd basically need to test-divide by all the primes with 100 or fewer digits, and that is a *lot* of numbers and even with computers that could do a trillion tests per second would require much more than the lifetime of the universe to complete. Since it is relatively easy to find 100-digit primes, a very difficult (and effectively impossible, at least now) problem can be created by finding two 100-digit primes, multiplying them together, and then asking someone to factor the resulting 200-digit composite number.

B.2 Modular Arithmetic

Modular arithmetic is sometimes called "clock arithmetic" since you can think of it as behaving something like a clock: each time you add and hour, you go to the next larger hour except when you get to 12, the next hour takes you back to 1. Similarly, subtracting an hour works like normal arithmetic until you get down to 1 and the next step down takes you to 12. There are only 12 essentially-different "hour numbers".

When we work in modular arithmetic we usually use clocks that have a 0 instead of a 12 (so the hours go from 0 to 11), and in fact we also like to work with clocks that have more or fewer numbers than 12.

It turns out that we can do much more than just add or subtract 1 in modular arithmetic. We can add and subtract all the numbers and we can also multiply, or even take powers of numbers and the usual rules of arithmetic continue to work. We have to be careful with division, however: it usually doesn't work.

Here's a more formal definition of modular arithmetic:

If n is a positive integer we can divide all the the integers into n classes, where all the numbers in each class differ by an integer multiple of n. If two integers are in the same class, we say that they are equal "modulo n." Mathematically, we write it as follows:

$$7 \equiv 15 (\mathrm{mod}\ 4).$$

(This is true, since by adding two copies of 4 to 7 we obtain 15.)

Often it is useful to name a particular integer in each class, and for that purpose the set: $\{0, 1, 2, \dots, n-1\}$ works. You can see that this works, since all of the numbers in this set differ by less than n, and once we get to n, it is n more than 0. Then $n+1$ is n more than 1, $n+2$ is n more than 2, et cetera.

Although in this article we'll be primarily interested in non-negative integers, it is perfectly reasonable to have equations like this:

$$\begin{aligned} -7 &\equiv 1 (\mathrm{mod}\ 8) \\ -17 &\equiv -25 (\mathrm{mod}\ 4) \end{aligned}$$

Once you have one number in an equivalence class you can find all the others by adding multiples (possibly negative) of n to it.

If you've got an integer m and you want to find out to which of the integers $\{0, 1, \dots, n-1\}$ it corresponds, modulo n just divide m by n and look at the remainder. If the remainder is positive, you're done; otherwise, add n to it. For example, $m = 39$ and $n = 7$. We see that 7 divides into 39 5 times with a remainder of 4, so $39 \equiv 4 (\mathrm{mod}\ 4)$. If $n = 11$ and $m = -122$ and you do the division, you obtain a quotient of 11 and a remainder of -1. Since the remainder is negative, add $n = 11$ to obtain 10 and we have: $-122 \equiv 10 (\mathrm{mod}\ 11)$

The following properties are all easy to prove. Assuming that $a \equiv b (\mathrm{mod}\ n)$ and $c \equiv d (\mathrm{mod}\ n)$ and that a, b, c, d are integers and k is a positive integer, then:

- $a + c \equiv b + d (\mathrm{mod}\ n)$
- $a - c \equiv b - d (\mathrm{mod}\ n)$
- $ac \equiv bd (\mathrm{mod}\ n)$
- $a^k \equiv b^k (\mathrm{mod}\ n)$

B.3 Primitive Roots $(\mathrm{mod}\ p)$

The concept of primitive roots applies more generally, but here we will only consider them modulo p, a prime number. We know that multiplication makes sense, modulo p, so we can construct a multiplication table, just as we would for normal integer arithmetic. Let's look at such a table where $p = 7$:

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Obviously, multiplying by zero yields zero, but if we ignore the first row and column, we have a very interesting multiplication table. The 1 behaves just like 1 in normal arithmetic: multiplying by 1 leaves the result unchanged.

In the rational numbers or the real numbers, every number has an inverse. The inverse of a number is the number you can multiply it by to obtain 1. The inverse of 3 is $1/3$; the inverse of π is $1/\pi$, et cetera. But in the integers, no number has an inverse except for 1. But in the example above, multiplication modulo 7, every non-zero number has an inverse. When it exists, the inverse of k is often written k^{-1}, so in the case of multiplication modulo 7, we have the following:

$$1^{-1} = 1;\ 2^{-1} = 4;\ 3^{-1} = 5;\ 4^{-1} = 2;\ 5^{-1} = 3;\ 6^{-1} = 6.$$

Since this is true, we can "undo" any multiplication: if we multiply a number n by m, we can recover n by multiplying again by m^{-1}. For example (using modulo 7), $3 \cdot 6 = 4$ and we can recover the 3 by multiplying by the inverse of 6; namely, 6: $4 \cdot 6 = 3$.

There is nothing special about the prime number 7: if p is any prime number, then in the multiplication table of the non-zero elements, every one of them will have an inverse.

Here is the interesting thing: let's look at successive powers of the non-zero elements:

$$\begin{array}{llllll}
1^1 = 1 & 1^2 = 1 & 1^3 = 1 & 1^4 = 1 & 1^5 = 1 & 1^6 = 1 \\
2^1 = 2 & 2^2 = 4 & 2^3 = 1 & 2^4 = 2 & 2^5 = 4 & 2^6 = 1 \\
3^1 = 3 & 3^2 = 2 & 3^3 = 6 & 3^4 = 4 & 3^5 = 5 & 3^6 = 1 \\
4^1 = 4 & 4^2 = 2 & 4^3 = 1 & 4^4 = 4 & 4^5 = 2 & 4^6 = 1 \\
5^1 = 5 & 5^2 = 4 & 5^3 = 6 & 5^4 = 2 & 5^5 = 3 & 5^6 = 1 \\
6^1 = 6 & 6^2 = 1 & 6^3 = 6 & 6^4 = 1 & 6^5 = 6 & 6^6 = 1
\end{array}$$

Notice that for certain numbers (3 and 5) in this case, the successive powers of these numbers generate all of the non-zero numbers. Numbers like this are called "primitive roots modulo p." Without proof we will say that for every prime number p there will exist primitive roots modulo p. This fact will provide the basis for the Diffie-Hellman-Merkle key exchange. See Section 15.

For every m, $m^{p-1} \equiv 1(\bmod\, p)$.

Another thing to notice is that for any non-zero element m, the smallest power k such that $m^k \equiv 1(\bmod\, p)$ has to divide $p - 1$ since after you get to 1, you cycle over and over until the end, where the result also has to be 1.

This means that if you can find a large p and can factor $p - 1$, then to test to see if a number n is a primitive root of p, you just have to look at powers of n, modulo p, which are divisors of $p - 1$. For example, if p is the following 101-digit prime number:

$$\begin{array}{l}
10000000000000000000000000000000000000000000000000 - \\
000000000000000000000000000000000000000000000002773
\end{array}$$

you will find that the factors are:

$$\begin{array}{l}
2, 6091, 4466507, 1967769462179, 2671394919769, 12469119346289, \\
761231721333902413970663, \text{and } 184169800969105958674950057 7
\end{array}$$

Thus, only those powers of a possible primitive root need to be tested. In fact, for this example, 2 is a primitive root, and could thus be used in the Diffie-Hellman-Merkle key exchange.

So here is why these primitive roots g are so valuable in cryptography. For every number x such that $1 \leq x < p$ there exists a number a such that $g^a \equiv x(\bmod\, p)$. Also, for every such x, there is another number, x^{-1} in the same range such that $x \cdot x^{-1} \equiv 1(\bmod\, p)$. If $g^a \equiv x(\bmod\, p)$ it is (relatively) easy to compute x from a, but it seems to be *very* difficult to compute a from x. The difficult problem is called the "discrete logarithm problem" since that's what we're basically trying to do: find the exponent.