2 Encryption Theory

We've seen a few different classical cryptosystems now, and we've seen their strengths and weaknesses. But how much better can we do? In this section we want to figure out what the best cryptosystem we *could* use looks like—and how we can think about encryption systematically.

2.1 Probability

A lot of our analysis of encryption systems and approaches to breaking them has relied on statistical arguments. We mostly couldn't *prove* that a Vigenère cipher had a key of length five, or that a pssage was a simple substitution cipher, or that a ciphertext `R` corresponds to a plaintext `e`. Instead we make guesses, and say they're more or less likely, and then try to weigh what is probably happening. And that benefits from a clear understanding of the theory of probability.

Definition 2.1. A *probability space* consists of three pieces:

1. A set Ω, called the *sample space*;
2. A set fi of subsets of Ω, called the *event space*;
3. A function $P :$ fi $\to [0, 1]$ that satisfies some rules will list off in a minute.

The underlying idea here is that Ω is the set of all the things that can happen. So if we're rolling a die, $\Omega = \{1, 2, 3, 4, 5, 6\}$. And we can ask questions like "what's the probability of rolling a 3?" But we can also ask questions like "what's the probability of rolling an even number?" So an event is a subset of the possible outcomes that we can measure and might care about.

Then the probability function P tells us how likely a given event is. So we can describe a set of outcomes, like "the die comes up odd" or "the die doesn't roll a 2", and P will output a number between zero and one, which we call the probability of that event.

In practice in this course, Ω will always be a finite set, and fi will be the power set 2^Ω of every possible subset of Ω. In a full course in probability or statistics you would allow Ω to be an infinite set, and then it generally turns out that you will have some subsets you can't measure the probability of. (If you're familiar with the Banach-Tarski paradox, this is related; if you want to know more, this is the domain of measure theory, which we cover in Math 6214.)

Example 2.2. Suppose we roll a die and flip a coin. Then our sample space

$$\Omega = \big\{(n, m) : n \in \{h, t\}, m \in \{1, 2, 3, 4, 5, 6\}\big\}.$$

We can define an event like "the coin comes up heads and the die roll is even", which would be the set

$$E = \{(h, 2), (h, 4), (h, 6)\} \subset \Omega.$$

In this case we'd have $P(E) = 1/4$.

Example 2.3. Suppose we roll two dice but don't care about which is which. Then our sample space will be

$$\Omega = \big\{(1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,2), (2,3), (2,4), (2,5), (2,6), \\ (3,3), (3,4), (3,5), (3,6), (4,4), (4,5), (4,6), (5,5), (5,6), (6,6)\big\}$$

There are 21 possible outcomes here, but some of them are more likely than others. $P(\{(1,1)\}) = 1/36$, but $P(\{(1,2)\}) = 2/36$.

So what properties does P need to satisfy for it to describe probability reasonably? We can start with a couple of simple ones:

1. For each $\omega \in \Omega$, $0 \leq P(\{\omega\}) \leq 1$
2. $\sum_{\omega \in \Omega} P(\{\omega\}) = 1$.

The first rule says that each outcome has a probability between can't happen, at 0%, and will definitely happen, at 100%. The second rule says that exactly one outcome will happen; the chance of *something* happening is exactly 1.

Once we start thinking about multi-outcome events, we can say more complex things. It's not true in general that $P(E \cup F) = P(E) + P(F)$; for instance, on a single die, $P(\{1, 2, 3\}) = 1/2$ and $P(\{2, 4, 6\}) = 1/2$, but

$$P(\{1, 2, 3\} \cup \{2, 4, 6\}) = P(\{1, 2, 3, 4, 6\}) = 5/6 \neq 1.$$

The problem here, though, is that one of the outcomes is getting counted twice; it appears in both events! When that doesn't happen, things work much more nicely.

Definition 2.4. We say that two events E and F are *disjoint* if $E \cap F = \varnothing$.

3. If E and F are disjoint, then $P(E \cup F) = P(E) + P(F)$.

We also want to be able to talk about when things don't happen. We write E^C for the *complement* of the event E, so that $E^C = \{\omega \in \Omega : \omega \notin E\}$. So if E is all the events where a thing happens, then E^C is all the events where it doesn't happen.

4. $P(E^C) = 1 - P(E)$.

2.1.1 Conditional Probability and Bayes's Theorem

We often have two events, and want to know what one tells us about the other. sometimes the answer is nothing: E is equally likely to happen whether F does or doesn't.

Definition 2.5. We say that two events E, F are *independent* if $P(E \cap F) = P(E)P(F)$.

Example 2.6. The classic example is rolling two dice: the probability of getting a 1 on the first die doesn't affect the probability of getting an odd number on the second die.

More formally: let Ω be the set of possible rolls of two dice, keeping track of which is which. Thus $\Omega = \{1,2,3,4,5,6\}^2 = \big\{(n,m) : n,m \in \{1,2,3,4,5,6\}\big\}$. We can define events $E = \{(n,m) in \Omega : n = 1\}$ the event where the first die rolls a 1, and $F = \big\{(n,m) \in \Omega : m \in \{1,3,5\}\big\}$ the event where the second die comes up odd. Then

$$\begin{aligned} P(E) &= 1/6 \\ P(F) &= 1/2 \\ E \cap F &= \{(1,1),(1,3),(1,5)\} \\ P(E \cap F) &= 1/12 = {}^1\!/_6 \cdot {}^1\!/_2. \end{aligned}$$

Thus the events E and F are independent.

But if we set $G = \{(n,m) : n < 3\}$ things are different. We have

$$\begin{aligned} P(E) &= 1/6 \\ P(G) &= 1/3 \\ E \cap G &= E \\ P(E \cap G) &= 1/6 \neq {}^1\!/_6 \cdot {}^1\!/_3. \end{aligned}$$

Example 2.7. Now take Ω to be the space in example 2.3, where we roll two dice but forget the order. We can take E to be the event that our roll includes a die with a 1, and F the event where our roll includes a die with a 2. Thus

$$\begin{aligned}
E &= \{(1,1),(1,2),(1,3),(1,4),(1,5),(1,6)\} \\
F &= \{(1,2),(2,2),(2,3),(2,4),(2,5),(2,6)\} \\
P(E) &= 11/36 \\
P(F) &= 11/36 \\
E \cap F &= \{(1,2)\} \\
P(E \cap F) &= 2/36 \neq {}^{11}\!/_{36} \cdot {}^{11}\!/_{36}.
\end{aligned}$$

Thus the events E and F are not independent. Informally, knowing that one of the dice came up 2 does in fact make it less likely that one of the dice came up 1.

We can take that last informal idea and make it rigorous. We want to know, if a given event E happens, how does that affect the probability of another event F?

Definition 2.8. Let E, F be events. The *conditional probability* of F given E is

$$P(F|E) = \frac{P(F \cap E)}{P(E)}.$$

This measures the probability that F happens given that we know E has happened. In essence, the denominator counts the number of ways that E can happen; the numerator counts how many of those ways that F can also happen.

Example 2.9. Returning to example 2.7, we can ask the conditional probability of F given E. We saw that $P(E) = 11/36$ and $P(F) = 11/36$, but $P(E \cap F) = 2/36$. Then

$$P(F|E) = \frac{P(F \cap E)}{P(E)} = \frac{{}^{2}\!/_{36}}{{}^{11}\!/_{36}} = \frac{2}{11}.$$

Thus if we know that at least one die came up 1, there's only a ${}^{2}\!/_{11}$ chance that at least one die (the other die) came up 2.

Taking 2.8 of conditional probability and clearing denominators gives the pleasingly symmetric formula:

$$P(F|E)P(E) = P(F \cap E) = P(E \cap F) = P(E|F)P(F).$$

Rearranging this formula a little more gives the most famous theorem in probability theory:

Theorem 2.10 (Bayes)**.** *Let E, F be events. Then*

$$P(E|F) = \frac{P(F|E)P(E)}{P(F)}.$$

This theorem is used a lot to figure out how to interpret new evidence; if seeing E makes F more likely, this formula tells you how much more likely F is after seeing E.

Proposition 2.11. *Let E, F be events. Then*

1.
$$P(E) = P(E|F)P(F) + P(E|F^C)P(F^C).$$

2.
$$P(E|F) = \frac{P(F|E)P(E)}{P(F|E)P(E) + P(F|E^C)P(E^C)}.$$

Proof. 1.
$$\begin{aligned} P(E|F)P(F) + P(E|F^C)P(F^C) &= P(E \cap F) + P(E \cap F^C) \\ &= P\big((E \cap F) \cup (E \cap F^C)\big) \\ &= P(E). \end{aligned}$$

2. Substitute the formula from part (1) into Bayes's Rule from theorem 2.10. This essentially another way of stating Bayes's Theorem.

□

Example 2.12. A famous and sadly relevant application of Bayes's Theorem comes from disease testing. Suppose we have a test for covid that is pretty good, but not perfect. Suppose that if you have covid, there's an 80% chance the test will give a positive result (the sensitivity), and if you don't have covid there's a 80% chance the test will give a positive result.

If you get a negative result, how likely arey ou to be uninfected?

The naive answer is something like 20%, but that isn't true. It actually depends on how common covid is. If we assume 2% of people have covid, then we can compute

$$\begin{aligned} P(C) &= .02 \\ P(-|C) &= .2 \\ P(-|C^C) &= .8 \end{aligned}$$

so using Bayes's Rule, we can compute

$$\begin{aligned} P(C|-) &= \frac{P(-|C)P(C)}{P(-|C)P(C) + P(-|C^C)P(C^C)} \\ &= \frac{.2 \cdot .02}{.2 \cdot .02 + .8 \cdot .98} \\ &= \frac{.004}{.788} \approx .005. \end{aligned}$$

So if you get a negative test, there's a roughly .5% chance that you actualy have covid anyway. This is pretty good!

Conversely, if you get a positive test, how likely are you to actually have covid? We can compute

$$\begin{aligned} P(C) &= .02 \\ P(+|C) &= .8 \\ P(+|C^C) &= .2 \end{aligned}$$

so using Bayes's Rule, we can compute

$$\begin{aligned} P(C|+) &= \frac{P(+|C)P(C)}{P(+|C)P(C) + P(+|C^C)P(C^C)} \\ &= \frac{.8 \cdot .02}{.8 \cdot .02 + .2 \cdot .98} \\ &= \frac{.016}{.212} \approx .075. \end{aligned}$$

Thus even if you test positive, you only have a 7.5% chance of actually having covid. False positives aren't super likely, but so many more people don't have covid than do that there will be far more false positives than real positives.

(That doesn't mean this is a problem with real-world covid tests. Many of the tests are reporting $P(+|C) \approx .8$, as I suggested, but reporting $P(+|C^C) \approx 0$, which means there will be almost no false positives. This changes the first calculation relatively little, but the second one dramatically.)

2.1.2 Random Variables

We often don't just care about events, but about their consequences, something we can measure from them.

Definition 2.13. A *random variable* is a function $X : \Omega \to \mathbb{R}$.

Note that the notation here is confusing and annoying, but the idea isn't: we have a function from our set Ω that outputs real numbers. For reasons that are actually good, we're calling this function a "variable" and using the letter X to represent it.

Example 2.14. If Ω is the sample space of rolling two dice, we can define a random variable $X : \Omega \to \mathbb{R}$ to be the sum of the two dice rolled. So $X(1, 2) = 3$ and $X(4, 6) = 10$.

We can use random variables to define events: in particular, for any real number $x \in \mathbb{R}$, we have three events

$$\{\omega \in \Omega : X(\omega) \leq x\} \quad \{\omega \in \Omega : X(\omega) = x\} \quad \{\omega \in \Omega : X(\omega) \geq x\}.$$

Example 2.15. Continuing example 2.14, we can consider the event $E = \{\omega \in \Omega : X(\omega) \leq 4\}$, the set of all rolls that give a value ≤ 4. This event is

$$E = \{(1,1),(1,2),(2,1),(1,3),(2,2),(3,1)\}.$$

We can also look at the event $F = \{\omega \in \Omega : X(\omega) = 6$, which would be

$$F = \{(1,5),(2,4),(3,3),(4,2),(5,1)\}.$$

Given these events and a probability measure, we can define two functions:

Definition 2.16. Let $X : \Omega \to \mathbb{R}$ be a random varaible on a finite sample space. The *probability density function* of X is denoted $f_X(x)$ and defined by

$$f_X(x) = P(X = x).$$

The *cumulative distribution function* is denoted $F_X(x)$ and defined by

$$F_X(x) = P(X \leq x).$$

Example 2.17. Further continuing example 2.14, we can see that $f_X(1) = 0$ since it's not possible to roll 1. $f_X(2) = 1/36$, $f_X(5) = 4/36$, and $f_X(15) = 0$.

We can also compute $F_X(2) = 1/36$, $F_X(5) = \frac{1+2+3+4}{36} = 10/36$, and $F_X(15) = 1$.

In general, for finite sample spaces we mostly think about the probability density function (pdf), and for infinite or continuous sample spaces we think abou the cumulative distribution function (cdf). In fact, in a continuous sample space we define the pdf to be the derivative of the cdf. But in this class we'll mostly just want the pdf.

It's often helpful to do this procedure kind of backwards. We define the probability density function we want, and use that to produce a random variable and a probability distribution. Two common examples of this:

Example 2.18. The most common probability distribution we'll see in this course is the *uniform distribution*. A uniform distribution on the set S with n elements is given by a random variable X satisfying $f_X(j) = 1/n$ if $j \in S$. This gives us a probability function on the state space $\Omega = S$ defined by $P(\{i\}) = f_X(i)$.

Example 2.19. A *binomial distribution* is defined by the function

$$f_X(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

This models a situation where we make the same binary choice n times. If we do n "tests" which each have a probability of succeeding p, then the probability of getting exactly k successes is $f_X(k)$.

Finally, we often want to average our possible outcomes somehow.

Definition 2.20. Let X be a random variable that takes on the values $x_1, \dots, x_n$. Then the *expected value* or *mean* of X is

$$\begin{aligned}\mathbb{E}(X) &= \sum_{i=1}^{n} x_i \cdot f_X(x_i) \\ &= \sum_{i=1}^{n} x_i \cdot P(X = x_i) \\ &= \sum_{\omega \in \Omega} P(\{\omega\}) X(\omega).\end{aligned}$$

Remark 2.21. This is a case of us being stuck with terrible terminology. The "expected value" of a random variable is generally not a value you actually expect to get. In some cases, like rolling a six-sided die, the expected value is not actually a possible value.

Example 2.22. If we're rolling one die and want to calculate the expected value, we have

$$\mathbb{E}(X) = \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 4 + \frac{1}{6} \cdot 5 + \frac{1}{6} \cdot 6 = \frac{7}{2}.$$

2.2 Information Theory

With these probability ideas set out, we can apply them to encryption. The first thing we want to do is put down a formal definition of what we're talking about. (This is one of the core techniques of using math to reason about something, and generally the first step to a mathematical approach to anything.)

Definition 2.23. A (symmetric) encryption system is composed of:

- A set of possible messages $\mathcal{M}$

- A set of possible keys $\mathcal{K}$
- A set of possible ciphertexts $\mathcal{C}$
- An *encryption function* $e : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$
- A *decryption function* $d : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$

such that the decryption function is a partial inverse to the encryption function: that is

$$d(k, e(k, m)) = m \qquad\qquad e(k, d(k, c)) = c.$$

We often write $e_k(m) = e(k, m)$ and $d_k(c) = d(k, c)$. Thus for each $k \in \mathcal{K}$, $d_k = e_k^{-1}$. This implies that each e_k is one-to-one.

Of course, some encryption systems are terrible. To be good, we'd like our cryptosystem to have the following properties:

1. Given any $k \in \mathcal{K}, m \in \mathcal{M}$, it's easy to compute $e(k, m)$.
2. Given any $k \in \mathcal{K}, c \in \mathcal{C}$, it's easy to compute $d(k, c)$.
3. Given a set of ciphertexts $c_i \in \mathcal{C}$, it's difficult to compute $d_k(c_i)$ without knowing k.
4. Given a collection of pairs (m_i, c_i), it's difficult to decrypt a ciphertext whose plaintext is not already known. ("known-plaintext attack").

The first two principles make a cryptosystem practically usable; the third and fourth make it secure. The fourth property is by far the most difficult to achieve. You'll notice that all of the cryptosystems we've studied so far satisfy the first two properties, and several of them do at least okay on the third, none of them achieve the fourth at all.

(Recall that we didn't do an unknown-plaintext attack on the Hill cipher, but we did implement a known-plaintext attack).

These principles are particularly important because they provide security even if your adversary knows the cryptosystem you're using, but not the key. This insight is often called *Kerckhoffs's Principle*, after the nineteenth-century Dutch military cryptographer Auguste Kerckhoffs, who wrote that a cryptosystem "should not require secrecy, and it should not be a problem if it falls into enemy hands." This was later reformulated by Claude Shannon as *Shannon's maxim*: "The enemy knows the system."

This principle is practically important because, while it's difficult to come up with an entirely new cryptosystem; it's relatively easy to generate a new key. We can change keys on a regular basis, and abandon the use of old keys that have been compromised; it's much more difficult to do the same thing with an entire cryptosystem. Similarly, keys are much smaller than entire systems, so it's easier to communicate them and keep them secret.

So what are the requirements for a cryptosystem to be secure in this manner? When is the key enough to provide security?

2.2.1 Perfect Secrecy

Definition 2.24. A cryptosystem has *perfect secrecy* if knowing the ciphertext conveys no information about the plaintext, even given knowledge of the cryptosystem.

Mathematically, if $\mathcal{M}$ is the set of possible messages, and $\mathcal{C}$ is the set of possible ciphertexts, a system has perfect secrecy if $P(m|c) = P(m)$ for all $m \in \mathcal{M}$ and $c \in \mathcal{C}$.

Here $P(m)$ is the probability of the m message being m, and $P(m|c)$ is the probability of the message being m given that you know the ciphertext is c. Thus a system has perfect secrecy if knowing the ciphertext gives you no information about the message.

Recall that Bayes's theorem says $P(a|b)P(b) = P(b|a)P(a)$. Thus a message has perfect secrecy if and only if $P(c|m) = P(c)$ for all $m \in \mathcal{M}, c \in \mathcal{C}$. Thus we can also say a cryptosystem has perfect secrecy if knowing the *message* gives no information about the *ciphertext*.

Example 2.25. Suppose we have a cryptosystem with two keys k_1, k_2; three messages m_1, m_2, m_3; and three ciphertexts c_1, c_2, c_3. Assume that $P(m_1) = P(m_2) = 1/4$ and $P(m_3) = 1/2$. Further, suppose we have an encryption function given by the following table:

	m_1	m_2	m_3
k_1	c_2	c_1	c_3
k_2	c_1	c_3	c_2

Let's assume the keys are used with equal probability. Then we can compute the probability that the ciphertext is c_2:

$$P(k_1)P(m_1) + P(k_2)P(m_3) = \frac{1}{2} \cdot \frac{1}{4} + \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{8}.$$

However, this system does not have perfect secrecy, since $P(c_1|m_3) = 0$, or alternatively, since $P(m_2|c_2) = 0$.

Because the encryption function is one-to-one, we know that $\#\mathcal{C} \geq \#\mathcal{M}$ for any encryption system. (Every system we've studied has had $\#\mathcal{C} = \#\mathcal{M}$, since the plaintext is a string of letters, and the ciphertext is an equal-length string of letters. But it's easy enough to *not* do this).

A system with perfect system has some other constraints.

Proposition 2.26. *If a cryptosystem has perfect secrecy, then $\#\mathcal{K} \geq \#\mathcal{M}$.*

Proof. Fix some specific ciphertext $c \in \mathcal{C}$ with $P(c) > 0$. Recall that perfect secrecy means that $P(c|m) = P(c)$ for every $m \in \mathcal{M}$, so this means that $P(c|m) > 0$ for every $m \in \mathcal{M}$. (In other words, if the ciphertext gives no information about the message, this means that any possible ciphertext has to be possibly linked to any possible message).

Thus there is at least one key k such that $e(k, m) = c$. Further, by injectivity, this key has to be different for each message: if $e(k, m_1) = c$ and $e(k, m_2) = c$ then $m_1 = m_2$. Thus there is at least one key for each message, and thus $\#\mathcal{K} \geq \#\mathcal{M}$. □

Example 2.27. The Caesar cipher does not have perfect secrecy for messages of more than one letter, since there are only 26 possible keys, and more than 26 possible messages.

The Vigenère and autokey ciphers do not have perfect secrecy for messages longer than the keylength.

We've shown that for a cryptosystem with perfect secrecy $\#\mathcal{C} \geq \#\mathcal{M}$ and now $\#\mathcal{K} \geq \#\mathcal{M}$. The most convenient possible world is when all three of these things are the same.

Theorem 2.28 (Shannon)**.** *Suppose a cryptosystem satisfies $\#\mathcal{K} = \#\mathcal{M} = \#\mathcal{C}$. Then the system has perfect secrecy if and only if:*

1. *Each key $k \in \mathcal{K}$ is used with equal probability; and*
2. *For each $m \in \mathcal{M}$ and $c \in \mathcal{C}$ there is exactly one $k \in \mathcal{K}$ with $e(k, m) = c$.*

Proof. Suppose the cryptosystem has perfect secrecy. Let $S_{m,c} = \{k \in \mathcal{K} : e_k(m) = c\}$. To prove (2) we just need to show that $S_{m,c}$ contains exactly one element for each m and c.

By injectivity, we know that $S_{m_1,c} \cap S_{m_2,c} = \varnothing$ if $m_1 \neq m_2$—otherwise there would be some key that encrypts both m_1 and m_2 to the same ciphertext c.

Further, $S_{m,c}$ is non-empty for every pair m, c. Since the cryptosystem has perfect security, knowing the ciphertext gives no information about the plaintext—so knowing the ciphertext is c can't rule out the fact that the plaintext is m. Thus every ciphertext must be reachable by every plaintext; so for each pair m, c, there is a key k such that $e_k(m) = c$.

Now fix some specific ciphertext $c \in \mathcal{C}$. We know that for each m there is at least one $k \in S_{m,c}$. But $\#\mathcal{M} = \#\mathcal{K}$ by hypothesis, so there must be exactly one k for each m. Thus $S_{m,c}$ has exactly one element for each m.

Now we just want to prove (1), that each key is used with equal probability. But for any k, c we can choose $m = d_k(c)$ and then because our encryption system has perfect secrecy, we can compute :

$$P(m) = P(m|c) = \frac{P(m,c)}{P(c)} = \frac{P(m,k)}{P(c)} = \frac{P(m)P(k)}{P(c)}$$

and canceling gives $P(k) = P(c)$. Since this is true for every k and every c, we must have $P(k)$ and $P(c)$ both constant; and in fact, $P(k) = P(c) = \frac{1}{\#\mathcal{C}}$.

Conversely, suppose our cryptosystem satisfies these conditions. Then given any $m \in \mathcal{M}, c \in \mathcal{C}$, there is exactly one $k \in \mathcal{K}$ with $e_k(m) = c$.

Fix some ciphertext c. Then each plaintext corresponds to exactly one key; and each key has the same probability; so each plaintext occurs with exactly the same probability. But this is the definition of perfect secrecy. □

Example 2.29 (The one-time pad)**.** A one-time pad is a cryptosystem of the following form:

The message is a string of N letters. The key is a randomly generated string of N letters. The ciphertext is obtained by adding each letter in the plaintext to the corresponding letter of the key (mod 26). This is essentially a Vigenère cipher, with a key length equal to the message length.

It's clear that the one-time pad satisfied property (2) of theorem 2.28. As long as the keys are generated uniformly at random, it also satisfies property (1), and this cryptosystem has perfect secrecy.

Thus a properly-implemented one-time pad is mathematically perfectly secure. However, it is rarely used becuase it is quite cumbersome, and we have many much less cumbersome systems that are "good enough".

Further, the proper implementation can be difficult; if your process for generating the key is *not* perfectly uniformly random, then the cryptosystem is not perfectly secure, and it is possible to break it with enough information.

Remark 2.30. The one-time pad is cumbersome, because they key has to be as large as the message. But this is true for any cryptosystem with perfect secrecy, because we showed that $\#\mathcal{K} \geq \#\mathcal{M}$. So any system with perfect secrecy is necessarily awkward, and we rarely use them.

2.2.2 Entropy

Suppose a cipher doens't have perfect secrecy. How much information do we actually need to break it? Well, first we need to specify what we mean by "information".

Definition 2.31. Let X be a random variable that takes on finitely many possible values $x_1, \ldots, x_n$ with probabilities $p_1, \ldots, p_n$. Then the *entropy* of X is given by

$$H(X) = H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

(adopting the convention that if $p = 0$ then $p \log_2 p = 0$).

Proposition 2.32 (Shannon)**.** *1. H is continuous in each variable.*

2. If X_n is a random variable uniformly distributed over n possibilities, then $H(X_n)$ is monotonically increasing as a function of n.

3. If X can be broken down into consecutive subchoices, then $H(X)$ is a weighted sum of H for the successive choices.

Further, any function with these three properties is a constant multiple of H.

Entropy measures the amount of information we get from a choice or evaluation of a random variable.

Example 2.33. Supose X is a "random" variable that returns x_1 with probability 1. Then

$$H(X) = -1 \log_2(1) = 0$$

because seeing the actual outcome gives no additional information over knowing the distribution.

Example 2.34. Supose X is a uniform distribution over a set of size n. Then

$$H(X) = -\sum_{i=1}^{n} \frac{1}{n} \log_2 \frac{1}{n} = \sum_{i=1}^{n} \frac{\log_2(n)}{n} = \log_2(n).$$

This makes sense—choosing uniformly from n things gives you about $\log_2(n)$ bits of information.

In particular, if X is a uniform distribution over the English alphabet, the entropy is $\log_2(26) \approx 4.7$.

Fact 2.35. *Let X be a random variable with n possible outcomes. Then*

1. *$H(X) \leq \log_2(n)$.*

2. *$H(X) = \log_2(n)$ if and only if the distribution is uniform.*

Thus entropy is maximized when the choice is maximally uncertain.

We said the entropy of a uniform distribution over the English alphabet is about 4.7 bits. But in actual English, letters aren't chosen at random! We can only really find the entropy of written English experimentally, by testing large bodies of English.

If we simply look at the probability distribution over letters from a frequency chart, we get $H \approx 4.132$ per letter. The fact that this number is less than 4.7 reflects the fact that not all letters are equally common.

However, English also doesn't consist of random sequences of letters. Some bigrams are more common that others. Taking bigram frequencies into account gives an entropy estimate of approximately 3.56 per letter. Of course then we need to consider trigrams, and quatragrams, and pentagrams, and in fact the entire infinite sequence; we experimentally estimate that English has an entropy of about 1.5 bits per letter.

Thus written English is highly redundant: about 70% redundant. We can also say that English has a *redundancy* of about 3.2 bits per letter. This doesn't mean we can randomly remove 70% of letters and still expect a readable message; it does mean that with a clever algorithm, we can *compress* a message to 30% of its original bit count.

This sounds wasteful, but is a really useful property of language: if we needed to track the difference between xkkyrosl and xkkyorsl carefully, reading would be quite difficult.

This also explains a semi-famous meme:

> Aoccdrnig to rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit a porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

(The claim as stated isn't quite true; it's possible to scramble words enough to make reading difficult, especially if you move the first and last letters. But this does show the redundancy in written English, since we can still understand a pretty badly scrambled message.)

2.2.3 Unicity distance

So what does entropy tell us about the practical ability to decrypt a message? How can we use this redundancy to understand the security of a cryptosystem?

We can view a cryptosystem as using the key to *remove* information from the plaintext to give us the ciphertext. But the key can only remove as much information as it contains; if the key doesn't remove all the redundancy, there's enough information in principle to recover the message.

Definition 2.36. The *unicity distance* for a given language and cipher is the length of an original ciphertext necessary to, on average, have enough information to break the cipher.

Example 2.37. CWU as a Caesar cipher can be "GAY" or "VPN". Without more context there's no way of telling. In fact the Caesar cipher has unicity distance of 2; most messages over 2 characters are breakable, so this is an exception.

Example 2.38. Suppose ABCDE is the ciphertext from a simple substitution cipher. It can be any word with no repeated letters; it could be "water" or "slope" or "maths" or a number of other things.

Example 2.39. Suppose the ciphertext is still "ABCDE", but this time we think the text was enciphered with a Vigenère cipher with a keyword of 5 letters. In this case the plaintext can be literally anything.

So how do we find the unicity distance? How do we know how much ciphertext we need?

Proposition 2.40. *The unicity distance of a language and encryption scheme is the number of bits in the key devided by the redundancy of the language.*

Proof Sketch. A message of length n and redundancy r bits has a total redundancy of nr bits. Thus if a key has more than nr bits it can remove all the information, and if it has fewer it cannot. $\square$

Example 2.41. A Caesar cipher has 25 possible keys, which is 4.64 bits. $4.64/3.2 \approx 1.45$, so you need at least 1.45 characters to decrypt a message enciphered with a Caesar cipher.

A simple substitution cipher has 26! possible keys, which is about 2^{88}. Thus there are 88 bits worth of keys. $88/3.2 \approx 27.5$ so you need at least 28 letters to decrypt a message enciphered with a simple substitution cipher.

A Vigenere cipher with an N-letter keyword has 26^N possible keys, for $\log_2(26^N) = N\log_2(26) \approx N \cdot 4.7$ bits. Thus the unicity distance is $N \cdot 4.7/3.2 \approx N \cdot 1.47$.

Remark 2.42. Remember that these are average minimums. You don't want to promise anyone you can break a simple substitution cipher with thirty characters of ciphertext, even though it's more likely to be possible than not.

Example 2.43. Earlier we mentioned one-time pads. By definition, a one-time pad has a key length equal to the message length. Suppose we have an N-letter message. Then like the Vigenère cipher there are 26^N possible keys, worth $\approx 4.7N$ bits. The unicity distance is $4.7N/3.2 \approx 1.47N$ letters.

But since the message is of length $N < 1.47N$, it is below the unicity distance, and we can't decrypt it. This is exactly what you'd expect, since a one-time pad has perfect secrecy and thus can't be decrypted at all.

We can think of a one-time pad as putting n bits of information in the key, and then using that key to transmit n bits of information. The key can completely conceal all the information in the message, since the key contains as much information as the message. But this means the key isn't any easier to communicate than the message itself is.

2.3 Inconvenience: the Cryptographer's Friend

2.3.1 Diffusion and Confusion

An informal approach to understanding how hard a cipher is to crack comes from Claude Shannon, who gave two properties a good cryptographic method should have:

Definition 2.44. An encryption method has good *diffusion* if changing one character of the plaintext changes several characters of the ciphertext, and vice versa.

Definition 2.45. An encryption method has good *confusion* if each part of the ciphertext depends on many parts of the key. This makes it hard to figure out the key from the ciphertext.

Thus diffusion means that changes in the plaintext are spread out through the ciphertext; while confusion means that changes in the key are spread out through the ciphertext. Each of these properties makes frequency analysis harder, since they increase the extent to which letters do not pair up one-to-one.

Simple substitution, Vigenère, and autokey ciphers have essentially no diffusion or confusion; each plaintext letter corresponds to one ciphertext letter, in a way mediated by one letter of the keyword. This makes them very susceptible to frequency analysis, as we saw in previous lectures.

The Hill cipher had better diffusion and confusion. Each letter of the ciphertext depends on an entire block of the plaintext, and an entire row of the key matrix. (It would be better if each letter of the ciphertext depended on the entire matrix).

This means that we had to do frequency analysis on n-grams instead of on individual letters, which is much harder. (We mostly looked at cases where $n = 2$, but a realistic implementation would want much larger blocks). It also means that guessing "some" of the ciphertext doesn't actually give us any of the entries in the key.

Diffusion and confusion do have one major downside: error propagation. A small error in the ciphertext will make the decrypted plaintext drastically different, and possibly unreadable. (You may have noticed this in the homework, if you ever made an arithmetic error—the rest of the decryption would come out as gibberish). But of course this is exactly the property that makes it hard to decrypt—that guessing it halfway doesn't give you enough information to finish the cryptanalysis.

2.3.2 Complexity and big-O notation

In one sense, a Vigenère cipher isn't any more secure than a Caesar cipher: we can break either of them. And yet it's clear that the Vigenère cipher is much more secure. What do we mean by that?

Fundamentally, we care not just about whether it's possible to break an encryption scheme, but how much work it will take. If I can, mathematically, break your cryptosystem, but only with a hundred years of work, that's very much like not being able to break it at all. So we want to measure how many steps of computation it takes to break a cipher.

At the same time, if it takes you a hundred years do actually encrypt a message, that's also very like not having an encryption system at all. We can make most cryptosystems take longer by making the key larger, or by requiring more cumbersome computations everywhere, including in the encryption. In a good cipher, we want encryption and decryption to be fast if you have the key, but difficult if you don't.

So we really want to measure the relative difficulty of using and of breaking the cipher. And we want to see how these change as we make the key longer or shorter. To do that we need to talk about big-O notation.

Definition 2.46. Let $f(x)$ and $g(x)$ be positive functions of x. We say that f is big-O of g, and write $f(x) = O(g(x))$, if there are positive constants c, C such that $f(x) \leq cg(x)$ for all $x \geq C$.

This represents the idea that when x is very big, $f(x)$ will not be much bigger than $g(x)$.. Generally we'll use this when f is a relatively complicated function, and we can replace it with a simpler function g.

Proposition 2.47. *If* $\lim_{x \to \infty} \frac{f(x)}{g(x)}$ *exists and is finite, then* $f(x) = O(g(x))$.

Example 2.48. If $f(x)$ is bounded for all $x \geq C$, then we write $f(x) = O(1)$.

It's easy to see that $x = O(x^2)$. It's also the case that $7x^2 + 5x + 3 = O(x^2)$.

$x^n = O(2^x)$ for any natural number n.

When we study algorithms, we use big-O notation to describe the running time of algorithms. In particular, we want to compare the number of steps an algorithm takes to the number of *bits* in the input, since computers operate in bits. (Importantly, this means that if our algorithm takes a number n as input, we don't want to think about the size of the number n; instead we want to think about the number of bits it takes to represent n, which is approximately $\log_2(n)$).

If an algorithm takes $f(k)$ steps when given k bits as input, and $f(kn) = O(g(k))$, then we say that the algorithm has a running time of $O(g(k))$.

If an algorithm is $O(k^\ell)$ for some constant $\ell > 0$, we say that the algorithm runs in *polynomial time.* (If $\ell = 1$ then we say the algorithm runs in *linear time*; if $\ell = 2$ we say it runs in *quadratic time*, and so on). We consider polynomial-time algorithms to be "fast".

If an algorithm is $O(2^{ck})$ for some $c > 0$, we say the algorithm runs in *exponential time.* Exponential time algorithms are considered "slow". Thus in the ideal cryptosystem, encryption and decryption knowing the key will be polynomial-time computations, but decryption without the key will be an exponential-time computation.

There is a third important category of *subexponential time* algorithms. These algorithms are $O(2^{\epsilon k})$ for every $\epsilon > 0$. Thus they're faster than exponential, but not necessarily as fast as polynomial algorithms. For instance, an algorithm that runs in $O(2^{\sqrt{k}})$ time is subexponential, but significantly slower than polynomial.

Example 2.49. If you have the key, encrypting or decrypting a message with a Caesar cipher is $O(n)$: you need to do one addition for each letter (or bit) of the message.

(It may be useful to note that it's $O(n)$ regardless of whether n is the number of letters or the number of bits. These are different but only by a constant factor: if n is the number of letters then the number of bits is approximately $4.7n$.)

Breaking a Caesar cipher, by finding the key from a ciphertext, is actually $O(1)$. Regardless of how long the message is, we only need to try 26 possible keys, on a small portion of the message.

Example 2.50. The Vigenère cipher is $O(n)$ to encrypt or decrypt if you have the key: it's still one addition for each letter.

Breaking the Vigenère cipher with the Kasiski test, by looking for all repeated trigrams and counting offsets, takes $O(n^3)$ done in the naive way, since looking for repeated trigrams requires a lot of comparison; it is $O(n^2)$ if you organize the work cleverly.

Breaking the Vigenère cipher by computing indices of coincidence for substrings is also $O(n)$. You only need to tally each letter once, and then you do some standarized additions and comparisons.

Example 2.51. Enciphering or deciphering a Hill cipher message involves a matrix multipliation. Multiplying an $n \times n$ matrix by a $n \times 1$ vector requires n^2 distinct integer multiplications and so is $O(n^2)$.

Breaking the Hill cipher with a ciphertext-only attack is harder to analyze. Brute force would have complexity $O(n^3 26^{n^2})$, and a I found a paper with an algorithm of complexity $O(n 13^n)$.

This is exponential, and therefore actually really good complexity! You can improve this with some frequency analysis tools, but I couldn't find a good estimate how efficient that is. The main reason the Hill cipher is insecure is the vulnerability to a known-plaintext attack. Modern cryptosystems are difficult to find keys for even if you have multiple plaintext-ciphertext pairs.

2.3.3 One-way functions and $\mathcal{P} \neq \mathcal{NP}$

One of the major goals of encryption is to create a *one-way function*. This is an invertible function f such that $f(x)$ is easy to compute, but $f^{-1}(x)$ is difficult to compute.

In particular, if we have a one-way function $f : \{0,1\}^n \to (\mathbb{Z}/26\mathbb{Z})^{\mathbb{N}}$ that outputs a sequence of numbers mod 26, we can get a secure cipher. We can exchange an integer, and then use the same integer to generate an arbitrary-length keystream. From an information-theoretic perspective, the entropy of the keystream is no greater than the entropy of the key; if you know our key has only ten bits of information, there are only 1024 possible keystreams and you can check them all. And finding ten bits worth of keystream will be enough to identify the key.

But if we have a practically one-way function, then that doesn't help. They keystream might have enough information to determine the key uniquely, but figuring out the key from the keystream could be computationally too difficult. This is the theory that modern keystream ciphers use: some complicated, hard-to-invert function that generates a large and unpredictable keystream from a relatively short key.

Unfortunately, it's not entirely clear that one-way functions are even possible in theory! We'd like a function that can be computed in polynomial, but only inverted in exponential time. And this runs into one of the most famous and important problems in modern mathematics.

We say an algorithm is in the class $\mathcal{P}$ if it runs in polynomial time. But we say it is in $\mathcal{NP}$, for "nondeterministic polynomial", if an answer can be *checked* in polynomial time. (Formally, a problem is in $\mathcal{NP}$ if there is an algorithm using random components that runs in polynomial time if it gets maximally lucky. So if you can check an answer in polynomial time, you can use the algorithm "guess an answer and check if it's correct", and since you're perfectly lucky you would guess the correct answer first.)

If computing a function is in $\mathcal{P}$ then inverting it is always in $\mathcal{NP}$: if I tell you that z is $f^{-1}(y)$, you can check my answer in polynomial time by computing $f(z)$ and seeing if you get y.

It is widely believed that there are problems that are in $\mathcal{NP}$ but not in $\mathcal{P}$—problems where the answers are easy to check, but not easy to find. But this isn't known for sure. In 1999, the Clay Mathematics Institute listed seven important problems, called the Millennium Problems, and attached a \$ 1 million dollar prize to each of them. One of these problems is to prove (or disprove) that $\mathcal{P} \neq \mathcal{NP}$.

There is a large class of problems known as $\mathcal{NP}$-complete problems. This class includes the Traveling Salesman problem, the Knapsack problem, the Subset Sum problem, and the 3-satisfiability problem. If any one of these problems has a polynomial-time solution, then all of them do, and all $\mathcal{NP}$ problems are in $\mathcal{P}$. We'll revisit some of these problems towards the end of the cours.

Almost all reasonable cryptographic algorithms involve algorithms in $\mathcal{NP}$, since we want to be able to encrypt and decrypt quickly with the key. Thus we can essentially never prove that a practically usable cryptosystem is secure without proving that $\mathcal{P} \neq \mathcal{NP}$ as a lemma.

Instead we have to satisfy ourselves with proving *relative* security: show that decryption is at least as hard as a problem that we think is hard and don't yet know how to solve. If we show that decryption is $\mathcal{NP}$ complete, that doesn't mean it's definitely secure. But it does mean that it's as good as we can reasonably do.

2.4 Coding Theory

We have one last topic we want to note before moving onto the third section of the course and studying modern encryption schemes. This is the idea of *encoding* of messages.

We'd like to use mathematical tools and techniques for encryption. As you may have noticed, it's easier to do math with numbers than it is with sentences. So we'd like a way to turn our sentences into numbers.

We saw a first approach to this in section 1.1.2, where we interpreted every letter as a number modulo 26. This allowed us to see a text as a sequence of letters mod 26, an approach we used effectively for the Hill cipher in section 1.4.3.

But this approach has a lot of limitations. The most obvious might be the inability to encode spaces in our message; we also can't encode punctuation or extra letters for other languages. We can solve those problems by working in a larger modulus; a lot of sources suggested working mod 29, which has some mathematical advantages we'll see soon.

But we'd like to do something more systematic than these ad hoc adjustments. And this approach also has a second downside: the need to work with *sequences* of numbers, rather than one large number. Both of these problems can be solved by working in binary.

Seven binary digits gives 128 possible numbers. In the 1960s, the American Standards Institute came up with an encoding system for basic English text, called the American Standard Code for Information Interchange (ASCII), as seen in figure 2.1.

Thus one character can be represented by a number from 0 to 127, which can be represented by a string of seven (or eight) bits. Then a sequence of characters can be represented as an extended sequence of bits, which can then be reinterpreted as a number.

Example 2.52. In section 1.2.5 we looked at the binary string

01010000 01001111 01001011 01000101 00100000 00110101 00111001 00110100 00110101 00111000 00101100 00110110 00110010

which corresponds to the text `POKE 59458,62`. We can view this entire string as a single 104-digit binary number, which translates to 6362793312790922647425965110834 in decimal.

We can now do whatever mathematical operations we want to that number, and at the end of the process convert it back to a binary string and thus a sequence of ASCII characters.

000	␀	(nul)	016	►	(dle)	032	␣	048	0	064	@	080	P	096	‘	112	p
001	☺	(soh)	017	◄	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	☻	(stx)	018	↕	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	♥	(etx)	019	‼	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	♦	(eot)	020	¶	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	♣	(enq)	021	§	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	♠	(ack)	022	▬	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	•	(bel)	023	↨	(etb)	039	’	055	7	071	G	087	W	103	g	119	w
008	◘	(bs)	024	↑	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009		(tab)	025	↓	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	◙	(lf)	026		(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	♂	(vt)	027	←	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012		(np)	028	∟	(fs)	044	,	060	<	076	L	092	\	108	l	124	|
013	♪	(cr)	029	↔	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	♫	(so)	030	▲	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	☼	(si)	031	▼	(us)	047	/	063	?	079	O	095	_	111	o	127	⌂

Figure 2.1: A table of ASCII encodings

Example 2.53. Let's encode the message `Stop`. Our table tells us that we have

$$\texttt{S} \to 83 = 64 + 16 + 2 + 1 \to 01010011$$
$$\texttt{t} \to 116 = 64 + 32 + 16 + 4 \to 01110100$$
$$\texttt{o} \to 111 = 64 + 32 + 8 + 4 + 2 + 1 \to 01101111$$
$$\texttt{p} \to 112 = 64 + 32 + 16 \to 01110000.$$

So we get the binary string `01010011 01110100 01101111 01110000` which works out to

$$112 + 111 \cdot 256 + 116 \cdot 256^2 + 83 \cdot 256^3 = 1400139632.$$

Depending on your goals, there are better systems for encoding text. Modern computers use unicode, which allocate two or three bytes to each character to allow the printing of thousands of glyphs and emoji (the smiley face symbol, for instance, corresponds to the hex value `01F60A` and thus the binary string `00000001 11110110 00001010`).

Remark 2.54. There are also improvements to encoding schema we can use if we want to prioritize some other goals. Error-correcting codes such as Hamming codes are robust to transmission mechanisms that have a chance to flip bits, at the cost of requiring more space to transmit a given message. Compression codes such as Shannon coding work in the other direction: they compress messages to make them as short as possible, but allowing small errors to have large effects. You can even combine both ideas to get messages that are both shorter and more robust to errors than the original. We may revisit these ideas towards the end of the course, but we'll leave them aside for now.

Now that we can turn strings into large numbers, we could use an affine cipher or a Hill cipher or something to encrypt them. (If we work on eight-byte blocks, then each block can take on values of up to $2^{64} \approx 2 \cdot 10^{19}$, so we can work modulo 2^{64}.) I think (though I'm not sure) that even an affine cipher would be resistant to a ciphertext-only attack if the modulus is large enough.

But as we'll see in the next section, we can do much better.