Math/CS 103 Professor: Padraic Bartlett

Lecture 9: Cryptography

Week 9 *UCSB 2014*

In our earlier lectures, we've discussed at length a few concepts behind "hard" and "easy" problems, with a specific focus on P and NP. In this talk, we examine a particular application of "hard" problems: the field of cryptography!

1 Cryptography: Classical Approaches

Definition. An **encryption algorithm**, or **cipher**, is a method that allows us to turn normal, plainly-readable text into difficult-to-read **ciphertext**, via the use of a secret key. Formally, we write

$$enc(k, \text{plaintext}) = \text{ciphertext},$$
$$dec(k, \text{ciphertext}) = \text{plaintext},$$

where

- plaintext is a message we want to encrypt,
- k is a key,
- *enc* is a function that takes in a unencrypted message and a key, and outputs an encrypted version of that message, and
- *dec* is a function that takes in an encrypted message and a key, and outputs the unencrypted version of that message.

In this system, the functions enc, dec are assumed to be widely known. There are cryptographic systems that do not assume this, but they are widely considered to be "bad." There are a number of reasons for why this is believed to be true (see security through obscurity and Kerckhoffs's principle for a wider discussion of this philosophy;) one of the simpler arguments is that there are many more ways to determine what an algorithm does (intimidate/bribe any of a number of engineers, steal a copy of the code, pose as a legitimate user and buy the algorithm, etc.) than to steal a specific key (which can only be done by attacking the specific user of the key you want.)

Typically, in cryptography, we refer to the two communicating parties as Alice and Bob (A and B for short, but seriously everyone uses Alice and Bob,) and the hypothetical third-party eavesdropper as Eve (E.) Given any encryption system, we typically evaluate its strength by looking at it in the following three situations:

1. The attacker, Eve, has access to a number of cipher texts.
2. The attacker, Eve, has access to a number of cipher texts and their original plaintexts.

3. The attacker, Eve, has the ability to generate cipher texts for whatever plaintext inputs they choose.

The first situation is the most common one: it is typically assumed that the attacker Eve has access to most, if not all, of the encrypted traffic that Alice and Bob send back and forth to each other. The second is stronger, but not unreasonable to expect; in many situations (see: WWII and the Enigma machine for some great stories) the attacker will be able to "steal" some unencrypted messages via subterfuge, and it would be great if an encryption method could still work even with this occasionally happening. The third is stronger yet: it basically is a kind of system that can withstand most anything, as long as the keys remain hidden. Strong cryptographical systems work in all of these systems, and are what we want to find.

People have been creating encryption schemes for thousands of years; essentially, as long as there have been people with secrets to keep, there have been ways to keep things secret. We study several of these here:

Algorithm. The Caesar-shift cipher. The Caesar-shift cipher, whose first recorded use was by Julius Caesar to protect various military secrets, is the following encryption scheme. Given a plaintext message m and a key k, "encrypt" m by doing the following: one-by-one, take each character of m and circularly shift it over k places to the right in the alphabet. Caesar historically used this cipher with a shift of three: i.e. $A \mapsto D$, $B \mapsto E$, $\ldots W \mapsto Z$, $X \mapsto A$, $Y \mapsto B$, $Z \mapsto C$. The decryption scheme is similar: take your encrypted message and character-by-character, circularly shift each letter over k places to the left in the alphabet.

This cipher, with a key of 13, is known as "ROT13" and is frequently used on the Internet to hide spoilers; this cipher-key combo is particularly convenient, because its encryption and decryption functions are identical (shifting right by 13 and left by 13 are the same in a 26-character alphabet.)

Example. Take the message[1]

```
"Just the place for a Snark!" the Bellman cried,
As he landed his crew with care;
Supporting each man on the top of the tide
By a finger entwined in his hair.
```

If we applied a Caesar shift with key 4, we would get the message

```
"Nywx xli tpegi jsv e Wrevo!" xli Fippqer gvmih,
Ew li perhih lmw gvia amxl gevi;
Wyttsvxmrk iegl qer sr xli xst sj xli xmhi
Fc e jmrkiv irxamrih mr lmw lemv.
```

Weaknesses. This is a very weak cipher. In particular, it is easy to beat with brute-force approaches: for example, suppose we saw the text

[1]From **The Hunting of the Snark**, by Lewis Carroll. It's pretty great.

```
Ns ymj gjlnssnsl ymjwj bfx stymnsl, bmnhm jcuqtiji.
```

we could simply just go through values of k until we got something that looked like a promising translation:

```
Mr xli fikmrrmrk xlivi aew rsxlmrk, almgl ibtpshih.
Lq wkh ehjlqqlqj wkhuh zdv qrwklqj, zklfk hasorghg.
Kp vjg dgikppkpi vjgtg ycu pqvjkpi, yjkej gzrnqfgf.
Jo uif cfhjoojoh uifsf xbt opuijoh, xijdi fyqmpefe.
In the beginning there was nothing, which exploded.
```

Given that there are only 26 values of k to pick, this should be something we can do relatively quickly. Moreover, we could just do this to a small sample of text if it took us too long to translate everything, as there's usually only one shift that's going to make our text look readable.

Algorithm. Simple substition ciphers. One key issue with the algorithm above was that the range of possible key choices was far too small: we could simply adopt a brute-force approach and look at all possible outputs of our decryption function under different keys, and discover the original plaintext in this way.

A solution to this was the idea of a simple substition cipher, which is defined as follows: first, write down the alphabet. Then, write down some permutation ρ of that alphabet: i.e.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

This permutation ρ is the key for an encryption scheme defined as follows: take a plaintext message, and character-by-character replace each letter in the plaintext message with a character from the permutation. For example, if we used the permutation described above, we would have $A \mapsto E, B \mapsto J, C \mapsto W, D \mapsto D, \ldots$

This algorithm avoids the weakness we've noticed that Caesar-shift is weak to: where the Caesar shift only had 26 keys, this algorithm has as many keys as there are ways to permute the characters of the alphabet. If we count, we can see that there are 26! ways in which to do this: this is because in creating a permutation we are choosing one of our 26 characters for A to map to, any of the remaining 25 characters for B to map to, and so on/so forth until we have have one last character for Z to map to.

26! is a much larger search space than 26: it's roughly $4 \cdot 10^{26}$. By comparison, the fastest supercomputer on record (as of November, 2013, and as far as I know) can perform roughly $34 * 10^{15}$ calculations per second; if it could check whether one given permutation was a viable interpretation of our encrypted text per calculation, it would require about 373 years to test all possible calculations. So, at the least, brute-force is not always the *best* strategy to use. . .

Example. Under the permutation

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

the phrase

is transformed into the phrase

Weaknesses. While this algorithm is pretty much immune to brute-force attacks, it is still remarkably easy to crack, even by hand. We can do this by studying the underlying structure of the plaintext message sent — i.e. the structure of the English language itself! — and use this information to crack the algorithm.

To get an idea of how this would work, consider the following longer sample of ciphertext:

```
Ony rgon af ony tdznoyapk hgb dk qykyo ab gii kdjyk qu ony
dbycpdodyk af ony kyifdkn gbj ony outgbbu af yedi hyb.
Qiykkyj dk ny wna, db ony bghy af vngtdou gbj zaaj wdii,
knyrnytjk ony wygl ontapzn ony egiiyu af ony jgtlbykk, fat ny
dk otpiu ndk qtaonyt'k lyyryt gbj ony fdbjyt af iako vndijtyb.
Gbj D wdii kotdly jawb prab onyy wdon ztygo eybzygbvy gbj
fptdapk gbzyt onaky wna gooyhro oa radkab gbj jykotau Hu
qtaonytk.  Gbj uap wdii lbaw D gh ony Iatj wnyb D igu Hu
eybzygbvy prab uap.
```

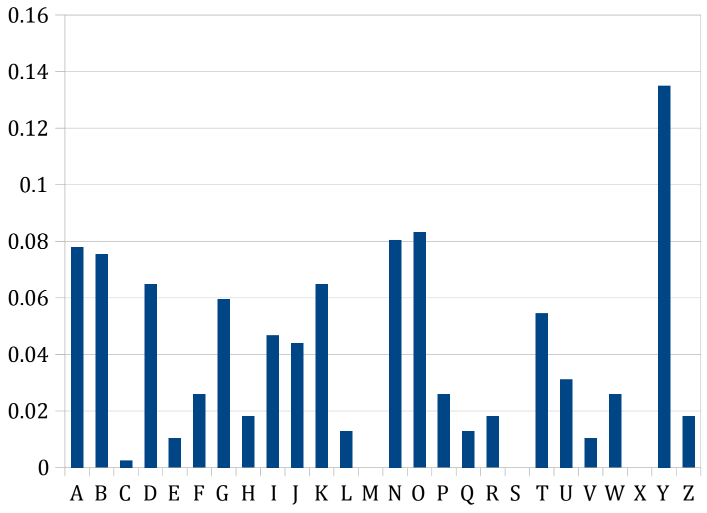
On its surface, this doesn't look much like English anymore. However, there are still bits of underlying structure that we can use to figure out what the cipher might be! For example, you could make the following observations:

- There is only one single-letter word that occurs in our passage above, the word "d." In English, there are only two one-letter words: "a" and "I." Consequently, it would seem rather likely that one of the letters A or I were mapped to D.
- Similarly, there is a character "k" that occurs as a single character after an apostrophe in our text sample; this likely eliminates every character except for "s", "t" or rarely "d." So we likely have that one of S or T maps to D.
- The three-letter word "ony" repeatedly occurs. If you do a word frequency analysis of the English language, you can see that the two most common three-letter words by a decent margin are "the" and "and"; so we could try seeing what happens if we assume one of these two words map to ONY.
- ...

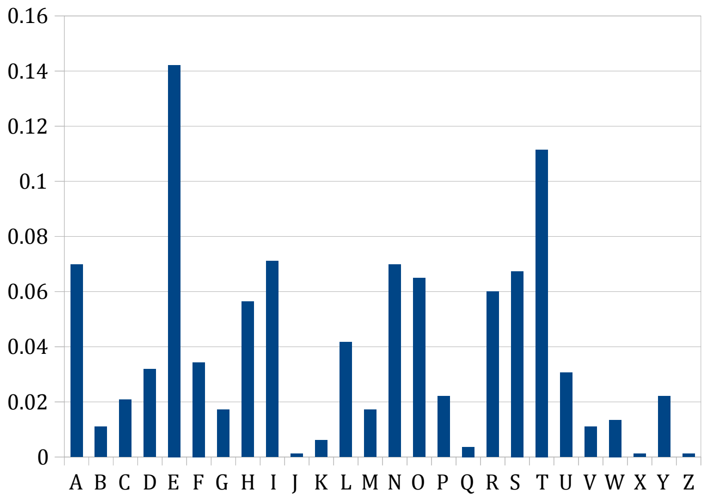
An objection you could make here is that we're deriving all of this structure from the "unencrypted" parts of our communication – i.e. we're figuring things out about our message by looking at the spaces and punctuation, which weren't encrypted!

In practice, people sometimes deal with this specific attack by simply omitting punctuation and spaces (or replacing all instances of a space with an infrequently-used letter like "q"). However, this doesn't stop us from a more fundamental method of attack — character frequency!

Specifically: notice that the characters in the above text occur with the following frequencies:



Now, notice that if you were to take a sufficiently large sample of works in English — say, a handful of Iain M. Banks novels, or the works of Raymond Chandler — you also have certain character frequencies! Intuitively, this makes sense: we're much more likely to see the letter "a" than the letter "q," for example. In general, English text tends to have the following character distributions:



This gives us the following observations:

- Given the above two graphs, we might assume that the most frequently-occurring character in our sample set, "y", corresponds to the most frequently-occurring character in English, "e."

- Consequently, if we looked through our text and picked out the most frequently-occurring three-letter cipherphrase, “ony”, we could assume that this matches up with frequently-occurring three-letter objects in English! Again, studying a large body of work reveals that the most frequently occurring trigrams in English, in order, are

  1. the
  2. and
  3. tha
  4. ent
  5. ing
  6. ion
  7. tio
  8. for
  9. nde
  10. has
  11. nce
  12. edt
  13. tis
  14. oft
  15. sth
  16. men

  Of these, “the” is the only one in the top 8 that ends in “e”: so it may be reasonable to assume that “the” and “ony” correspond to each other!

- If we apply this observation — that it is likely that $T \mapsto O, H \mapsto N, E \mapsto Y$ — we get

```
The rgth af the tdzhteapk hgb dk qeket ab gii kdjek qu the
dbecpdtdek af the keifdkh gbj the tutgbbu af eedi heb.
Qiekkej dk he wha, db the bghe af vhgtdtu gbj zaaj wdii,
kherhetjk the wegl thtapzh the egiieu af the jgtlbekk, fat he
dk ttpiu hdk qtathet’k leeret gbj the fdbjet af iakt vhdijteb.
Gbj D wdii kttdle jawb prab thee wdth ztegt eebzegbve gbj
fptdapk gbzet thake wha gttehrt ta radkab gbj jekttau Hu
qtathetk.  Gbj uap wdii lbaw D gh the Iatj wheb D igu Hu
eebzygbvy prab uap.
```

  The symbols we believe to be “correct” are in red.

- We now return to our earlier observations about the text that used its structure: (1) that the apostrophe character “k” likely corresponds to one of “s” or “t”, and (2) that the only occurring single character “d” should correspond to one of the two English single-letter words, “I” or “a”. In particular, the letter “d” is always capitalized in our text, so “I” likely maps to “D”! Furthermore, if we’re assuming that “t” maps to “o”, then we cannot have it mapping to “k” as well: therefore, we likely have “s” mapping to “k”. We can further update our text here with these observations:

```
The rgth af the tizhteaps hgb is qeset ab gii sijes qu the
ibecpities af the seifish gbj the tutgbbu af eeii heb.
Qiessej is he wha, ib the bghe af vhgtitu gbj zaaj wiii,
sherhetjs the wegl thtapzh the egiieu af the jgtlbess, fat he
is ttpiu his qtathet’s leeret gbj the fibjet af iast vhiijteb.
Gbj I wiii sttile jawb prab thee with ztegt eebzegbve gbj
fptiaps gbzet thase wha gttehrt ta raisab gbj jesttau Hu
qtathets.  Gbj uap wiii lbaw I gh the Iatj wheb I igu Hu
eebzygbvy prab uap.
```

From here, we can simply look at the phrases with all but one character determined — "qeset," "sijes," "with" — and make more guesses at characters. I.e. the only common word word ending in "eset" with five characters is "beset," so we have b mapping to q; similarly, "si?es" is either sides, sires, sites, sixes or sizes. We can eliminate sites on the grounds that we are already using the character t, and sixes and sizes on the fact that the character j occurs often in our sample text, and it's unlikely that x or z would do so. So we're down to j correspond to either d or r: the high frequency of the three-letter phrase "gbj" would lead us to believe that this is not "r," as there are not many three-letter words ending in "r" that we'd expect to see this many times, while there definitely are such words (specifically, "and") that end in d and occur this frequently!

In turn this motivates us to guess that the rest of gbj actually corresponds to "and:" if we do this, we get

```
The rath af the tizhteaps han is beset an aii sides bu the
inecpities af the seifish and the tutannu af eeii hen.
Biessed is he wha, in the nahe af vhatitu and zaad wiii,
sherhetds the weal thtapzh the eaiieu af the datlness, fat he
is ttpiu his btathet's leeret and the findet af iast vhiidten.
And I wiii sttile dawn pran thee with zteat eenzeanve and
fptiaps anzet thase wha attehrt ta raisan and desttau Hu
btathets.  And uap wiii lnaw I ah the Iatd when I iau Hu
eenzyanvy pran uap.
```

Repeating this process one more time — look for words like "B?essed," "da??ness," "ine??ities," etc. that are mostly completed, compare to the English language, make a guess — eventually gives us that our permutation is

ABCDEFGHIJKLMNOPQRSTUVWXYZ

and that our corresponding text is the following passage:

```
The path of the righteous man is beset on all sides by the
inequities of the selfish and the tyranny of evil men.
Blessed is he who, in the name of charity and good will,
shepherds the weak through the valley of the darkness, for he
is truly his brother's keeper and the finder of lost children.
And I will strike down upon thee with great vengeance and
furious anger those who attempt to poison and destroy My
brothers.  And you will know I am the Lord when I lay My
vengeance upon you.
```

Cryptography, now featuring Samuel L. Jackson.

Some of the weaknesses in this algorithm can be fixed, as we illustrate below:

Algorithm. Vigenère ciphers. In a sense, the main weakness of the simple substitution cipher is that each plaintext letter always corresponded to the same encrypted symbol: this allowed us to use our knowledge of English to break the code. We can fix this, however, by using a Vigenère cipher!

Specifically: the key to a Vigenère cipher is a code word k that is some string of characters. To illustrate, suppose that the code word is "bah." To encrypt some message, like (for example) "Friendship is Magic!," we use the code word as a way to "shift" the letters of the codephrase as follows:

1. Write down your message. Below it, write a number of copies of the codeword so that each character in our message is matched up with a character from the codeword, as below:

```
Friendship is Magic!
bahbahbahb ah bahba
```

2. Then, take each character in the message, and "shift" it by the codeword character corresponding to it! In other words, take each codeword character, interpret it as a number (i.e. $a \mapsto 1, b \mapsto 2, \ldots$), and circularly shift the message character to the right that many places. For example, our message becomes

```
Hsqgoluiqr ja Obokd!
```

Example. We provide another example here, this one a bit more long-form. Consider the codeword "bode," and the message to be encoded[2]

```
The wasp and all his numerous family
I look upon as a major calamity.
He throws open his nest with prodigality,
But I distrust his waspitality.
```

In this setting, we would form the four lines

```
The wasp and all his numerous family
bod ebod ebo deb ode bodebode bodebo
I look upon as a major calamity.
d ebod ebod eb o debod ebodebod
He throws open his nest with prodigality,
eb odebod ebod ebo debo debo debodebodeb
But I distrust his waspitality.
ode b odebodeb ode bodebodebod
```

which results in the text

```
Vwi bcht fps eqn wmx pjqjtdyx hpqnnn
M qqdo zrdr fu p qfldv hcaerkic.
Mg ilwqlw trtr mkh rjui anvw twqsmlcamya,
Qyy K smxvgyxv wmx ypwukieqkic.
```

[2]**The Wasp**, a poem by Ogden Nash. He's great.

Weaknesses. This algorithm, given a sufficiently long codephrase, can avoid all of the issues that came up with our earlier work: given a codephrase that's like a few sentences long, then we would expect any given letter in our message to be shifted by many different values, and therefore that analyzing the character distributions will be useless.

And this is true! However, it is still weak to attacks that exploit the underlying structure of the English language. Specifically, there is a technique, called the **Kasiski test**, that we can use to break this code.

Roughly speaking, the Kasiski test is centered around the following observations:

- English has a lot of repeated sets of characters. We used this structure to great success in our earlier problem: we broke our earlier code largely by looking for repeated triples of characters and matching them up to known English characters.

- It is likely that our source message, being originally some large English text, has a number of often-repeated sequences. While it is possible that not all of those repeats will be preserved by the Vigenère cipher, (i.e. in our earlier text sample, the triple "him" occurred above the triple "ebo" the second time and "ode" the third time,) it is certainly likely that **some** triples will line up nicely with our code word, and therefore still occur as triples (for example, the first and second occurrences of "him" are matched with the same triple "ode.")

- Why do we care about these repeated phrases in our encrypted text? Well: if they correspond to repeated phrases in the original text (and aren't there just by accident,) then they tell us something about our codeword! In particular, they tell us that if our codeword sent those two repeated phrases to the same phrases, then both of those phrases were lined up over the "same" portion of our codeword!

- Therefore, there must be a whole number of copies of the codeword separating these two phrases, in order for them to line up! For example, using this observation on our text above with the repeated "his" phrase, if we count the number of letters between the first occurrence of "his" and the second occurrence of "his," we get 88. This tells us that it is likely that our codephrase is a divisor of 88; i.e. one of 2,4,8 or 11. Further analysis, by looking for more of these repeated sets, can eliminate other options, and tell us the precise length of the codeword we're studying.

- From there, we simply need to find the elements of the codeword! We can do this just like we did for the simple substitution cipher, basically. To be specific: break our cipher text into groups, each corresponding to the character of the code word that translated it. I.e. if we had the text from our earlier example and had deduced that the code word was length 4, we could then simply group our cipher text into four groups, each corresponding to the characters in the ciphertext that were shifted by a fixed codeletter.

  On each of these groups, we can then perform the character analysis we did before, and deduce one-by-one the codeword's characters. We omit a worked example here, but it is entirely within the reader's powers to solve one in the HW!

In the preamble to this lecture, we mentioned that the idea of “hard” problems was useful for creating cryptosystems. We haven’t illustrated how that works yet — all of the cryptosystems above don’t rely on “hard” problems! However, all of the cryptosystems above also had weaknesses of various kinds, that has made each of them mostly unused in the modern world. Next week, we’ll talk about how an NP-hard problem can be used to create a “better” encryption scheme, and what some of the encryption schemes used today are!