CS100, Spring 2000, Project 7 Decryption Write-Up: Cracking the Code

1 Overview

1.1 The Story Line

You work for Bureau 13, a top-secret government organization that protects the public from all kinds of nasty things. In the past week, Bureau 13 has intercepted an email message from a nefarious society called the Evil Group of People (EGOP) that's bent on destroying the world. Unfortunately, the email message seems garbled even though EGOP used ASCII. Unclear as to EGOP's intent, your supervisors, Yan and Dis, have assigned you the task of deciphering the message. You are the perfect choice because you took CS100 in college and learned how to decrypt messages. Having become a highly skilled programmer, you will hopefully save the world in completing Project 7.

1.2 How This Write-Up Differs From the Project 6 Write-Up

Sections 2 and 3 have a few small changes from the Project 6 Write-Up, e.g. Section 2.3 defines ***cracking*** more carefully and Section 3.6 is new. Sections 4 and onwards are new.

2 Background: Encryption and Decryption

2.1 Characters

Your bosses informed you that the information EGOP sent is ***text*** written in English. Each English letter is called a ***character***. In general, a language uses a set of characters to build words. Since sets have no implied order, we organize the character sets into ***alphabets***, which usually sort characters.

2.2 Hidden Information

Suppose part of EGOP's message contains the passage of text `"TUBSU SIF BUUBDL"`. Assuming EGOP uses English, this portion doesn't appear to contain meaningful text. However, you might notice a pattern to the letters — try to substitute a letter "one-lower" in the alphabet for each given letter. So, the letter `'B'` becomes `'A'`, the letter `'C'` becomes `'B'`, and so on. Eventually, you will discover that EGOP "hid" the message `"START THE ATTACK"`.

2.3 Encryption and Decryption

Zounds! EGOP hid a secret message! How did they do that?

- The process of hiding information is called ***encryption*** or ***encoding***.
- The process of "unhiding" information is called ***decryption*** or ***decoding***.

Both processes are required! Encryption provides privacy and protection for information, like messages proclaiming intentions to take over the world. But, what good is the ability to hide something if you can't find ("unhide") it, and what good is the ability to unhide if nothing can be hidden?

To hide or unhide information, something must change the information. The set of rules for performing this ***transformation*** is called a ***key***:

- Encryption transforms text using an ***encryption key***.
- Decryption transforms text using an ***decryption key***.

A ***cryptosystem*** consists of both the rules for encryption and the rules for decryption. That is, a cryptosystem is an encryption key together with the corresponding decryption key:

- To encrypt text, transform unencrypted text with an encryption key.
- To decrypt text, transform encrypted text with a decryption key.

You might sometimes hear or see the term ***cracking***, which means figuring out the unencrypted text from encrypted text without knowing the decryption key.[1] Note that it suffices to find the decryption key: if you can figure out the decryption key, then you can use it to decrypt the encrypted text and thereby extract the unencrypted text.

2.4 Substitution Ciphers

A ***cipher*** is a cryptosystem where the encoding and decoding is done character-by-character, as opposed to being done on words, phrases, or blocks of 1024 characters. From undercover agent Artemov, you know that EGOP failed to study any computer science and, fortunately, chose one of the simplest cryptosystems: a ***substitution cipher***. A substitution cipher, or ***mapping/permutation***, changes each character to another character. For example, in Section 2.2, EGOP's substitution cipher exchanges each letter of the English alphabet with the "next" letter. Note that `'Z'` "wraps around" to `'A'`. The notation `'A'`→`'B'` means, "`'A'` maps to `'B'`", which also means, "replace each `'A'` with `'B'`". You must follow two rules with a substitution cipher:

1. Use every character from a given character set.
2. Map every character to only one other character.

Rules 1 and 2 means a character uniquely maps to another character. For example, if `'A'`→`'B'` is part of the encryption key, then neither `'A'`→`'c'` nor `'c'`→`'B'` can be part of the encryption key.

2.5 Encryption Key

Sometimes we might say ***encipher*** or ***encode*** instead of *encrypt*. A cipher uses an ***encryption key*** to encode text. So, applying the encryption key to text encrypts, or "hides", that text. Represent an encryption key by writing two lines:

1. On the ***top line***, write the character set.
2. On the ***bottom line***, write the conversion for each character.

Figure 1 shows the encryption key for the famous substitution cipher called the ***Caesar cipher***, which Julius Caesar supposedly used. The Caesar cipher simultaneously renames each letter with the letter 3 positions "down" in the alphabet and "wraps around" back to `'a'` after the letter `'z'`.

```
abcdefghijklmnopqrstuvwxyz   top line: alphabet
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
defghijklmnopqrstuvwxyzabc   bottom line: encoding of each letter of the alphabet
```

Figure 1: Encryption Key for Caesar Cipher

An encryption key renames each character on the first row to the character just below on the second row. Using mapping notation, you may express the encryption key in Figure 1 as `'a'`→`'d'`, `'b'`→`'e'`, ..., `'w'`→`'z'`, `'x'`→`'a'`, `'y'`→`'b'`, and `'z'`→`'c'`, where each character is used exactly once on the top and exactly once on the bottom.

How do you encipher a message using an encryption key? Call the unencrypted message ***plaintext*** and the encrypted message ***ciphertext***. Figure 2 places the plaintext on the top line above the ciphertext on the bottom line. Observe that each plaintext letter is replaced by the appropriate character from the encryption key for the Caesar cipher. For example, `'w'`→`'z'`, `'e'`→`'h'`, the space maps to a space, `'a'`→`'d'`, `'p'`→`'s'`, and so forth. In both Figures 1 and 2, the bottom line is the encoded version of the top line. (But, see also Figures 6–8.) Note that the encryption key does not require the top line to be sorted. See Figure 3 for an equivalent representation of the Caesar cipher. Once more, each line uses each character from the character set exactly once.

[1] In many cryptosystems, the decryption key is "easy" to compute from the encryption key. Therefore, the encryption key must be hidden for the cryptosystem to be secure. However, there are some cryptosystems where it is (believed to be) infeasible to compute the decryption key from the encryption key, in which case it is OK for the encryption key to be public. So-called ***public key cryptosystems*** have this property, e.g. PGP/RSA.

```
we apologize for the inconvenience  top line: unencoded plaintext
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
zh dsrorjlch iru wkh lqfrqyhqlhqfh  bottom line: encoded ciphertext
```

Figure 2: Encryption Example Using the Caesar Cipher

```
abnocdgpwxyefqrstuvhijklmz  top line: alphabet, in random order
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
deqrfgjszabhituvwxyklmnopc  bottom line: encoding of each character
```

Figure 3: Equivalent Encryption Key for the Caesar Cipher

2.6 Decryption Key

To produce understandable decrypted text, you need to decrypt the encrypted text. Sometimes we might say ***decode*** or ***decipher*** instead of *decrypt*. To decrypt, we undo encryption, i.e. "run encryption in reverse" to transform ciphertext into plaintext, as shown Figure 4.

```
we apologize for the inconvenience  top line: unencoded plaintext
↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
zh dsrorjlch iru wkh lqfrqyhqlhqfh  bottom line: encoded ciphertext
```

Figure 4: Intermediate Decryption Example Using the Caesar Cipher

Observe that Figure 4 has flipped the arrows upside down to show that decryption transforms the bottom line into the top line. Now, compare the character mappings used in Figures 2 and 4:

- Figure 2: 'w'→'z', 'e'→'h', the space maps to a space, 'a'→'d', 'p'→'s', and so forth.
- Figure 4: 'w'←'z', 'e'←'h', the space maps to a space, 'a'←'d', 'p'←'s', and so forth.

The mappings in Figure 4 reverse the mappings in Figure 2, which corresponds to the idea of "running encryption in reverse". So, to form the ***decryption key*** for decoding ciphertext, reverse all the mappings in the encryption key. Figure 5 shows this process for the encryption key from Figure 1.

```
abcdefghijklmnopqrstuvwxyz  top line: alphabet
↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
defghijklmnopqrstuvwxyzabc  bottom line: encoding of each character
```

Figure 5: Intermediate Decryption Key for the Caesar Cipher

For conciseness and consistency omit arrows but understand that they implicitly point down. So, you must flip Figures 4 and 5 upside down, yielding Figures 6 and 7. Figure 8 summarizes the different types and arrangements of keys.

```
we apologize for the inconvenience  top line: unencoded plaintext
zh dsrorjlch iru wkh lqfrqyhqlhqfh  bottom line: encoded ciphertext
```

Figure 6: Final Decryption Example Using the Caesar Cipher

2.7 Inversion

What is an ***inverse***? The inverse of a process is its "opposite". For example, the opposite of *increment by 2* is *decrement by 2*. For a given operation, the inverse of a value is its opposite value. For example, for

`defghijklmnopqrstuvwxyzabc`	top line: encoded alphabet
`abcdefghijklmnopqrstuvwxyz`	top line: alphabet

Figure 7: Final Decryption Key for the Caesar Cipher

	General Key	Encryption Key	Decryption Key
Top Line	alphabet	unencoded alphabet	encoded alphabet
Bottom Line	transformed alphabet	encoded alphabet	unencoded alphabet

Figure 8: Summary of Keys

addition, the inverse of 2 is -2. For cryptosystems, encryption and decryption are inverse processes. For the operation of transforming text, encryption and decryption keys are inverse "values".

To form a decryption key, you swap the top and bottom lines of an encryption key. More formally, swapping the top and bottom lines is called ***inverting***:

- inverting an encryption key produces a decryption key
- inverting a decryption key produces an encryption key

Why the term *inverse*? If you transform plaintext using an encryption key, i.e. encrypt, you produce ciphertext. If you then transform the ciphertext using the decryption key, i.e., decrypt with the inverse of the encryption key, you produce the original plaintext. That is, the encryption key and decryption key "undo" each other and are inverses of each other. Mathematically speaking, for plaintext p,

$$encode(decode(p)) = encode(decode(p)) = p.$$

For instance, $'\texttt{a}' \rightarrow '\texttt{d}'$ followed by $'\texttt{d}' \rightarrow '\texttt{a}'$ yields the mapping $'\texttt{a}' \rightarrow '\texttt{d}' \rightarrow '\texttt{a}'$.

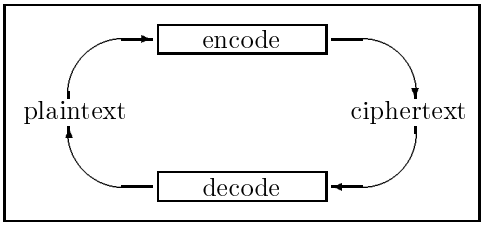


Figure 9: Encryption and Decryption are Inverse Processes

3 Foundation for Cracking: Frequency Analysis

To crack EGOP's message, you need to know how to compute a decryption key without knowing the encryption key — EGOP tries to keep theirs a secret. Before finding the decryption key, you need to review some basics of natural languages.

3.1 Natural Languages

In computer science, a ***language*** is any set of words over some alphabet. A ***natural language***, as opposed to a programming language, has native human speakers. Natural languages have lots of structure, such as grammar, semantics, and syntax, which assist decryption. To perform your task, consider only frequencies, as discussed below.

3.2 Frequencies

How might you crack a cryptosystem that uses character mappings? Consider how frequently certain characters appear in text. If you could spot repeated patterns in encoded text and then match them to known

patterns in "regular" text, you might be able to crack the cryptosystem! For example, the letter `'u'` almost always follows the letter `'q'`.[2] Natural languages have other patterns, too. A ***frequency*** is a measure of how often a pattern appears in a body of text. You may measure frequency of a pattern as either:

- a ***tally***: the number of times the pattern appears
- a ***fraction*** or ***percentage***: the ratio $\frac{\text{the number of times the pattern appears}}{\text{the total number of patterns}}$.

Recall that $x\% = \frac{x}{100}$ and observe that the fraction is always between $0\% = 0$ and $100\% = 1.0$.

There are published tables of frequencies of single letters and pairs of letters for different languages.[3] We refer to these frequencies as ***unigram*** and ***bigram*** frequencies:

- unigram = 1 letter, where *uni* = 1 and *gram* = letter
- bigram = 2 letter pair, where *bi* = 2 and *gram* = letter

Note that the order of letters matters, e.g. `"qu"` and `"uq"` are different! Higher-order frequencies, like trigrams, are also studied and would help our task, but for simplicity, do not consider them.

3.3 Example

For brevity, consider an invented language with the 4-letter alphabet {a,b,c,d}. Mark spaces between words with a hyphen (-) and assume no punctuation. Figure 10 shows a portion of text using this invented language. The following sections demonstrate how to collect and organize frequency data using ***frequency tables***.

```
-a-aaa-abba-abc-ac-ad-ada-add-baa-bad-cab-cb-cdc-dab-dad-dada-dc-
```

Figure 10: Example Text for Invented Language

3.4 Unigram Frequencies

You may collect unigram frequencies in tables using either tallies or percentages. From Figure 10, count the number of times `'-'`, `'a'`, `'b'`, `'c'`, and `'d'` each appear. Each character count produces a tally, tabulated in Figure 11. Compute each character's frequency as a ratio of the number of times that character appears and the total number of characters. You may tabulate the frequencies as ratios, as shown in Figure 12. These tables are called ***unigram tables***.

-	a	b	c	d
17	20	8	7	12

or

-	d	b	c	a
17	12	8	7	20

Figure 11: Unigram Frequencies (Tallies)

-	a	b	c	d
27	31	13	11	19

or

-	d	b	c	a
27	19	13	11	31

Figure 12: Unigram Frequencies (Percentages (%))

Although the tables might appear two dimensional, the numbers are in a single row. So, think of the set of unigram frequencies as a 1-D table. Why two tables for each table of unigram frequencies? You may choose to count characters in any order, so the tables in each pair are equivalent. However, note that the count and percent frequencies differ! To access a frequency for a particular character, use the notation $freq_j$, where j is any character from the character set, including `'-'`. For example, $freq_{\texttt{'a'}} = 31\%$ tells you that `'a'` occurs 31% of the time.

[2] Why not *always*? Because of typos, abbreviations, words borrowed from other languages, and people in advertising that like to mess around with spelling.

[3] See `http://www.cs.wright.edu/people/faculty/fdgarber/740/ascii/ascii2freq.html` for an example of frequencies in just words.

3.5 Bigram Frequencies

A bigram frequency measures how often a pair of letters occurs. For instance, take the ratio of the number of times 'c' comes before 'd' (1 time) with the total number of pairs (64 times). You will find that the pair "cd" appears 2% (1/64) of the time in the text shown in Figure 10. To collect all bigram frequencies, use a 2-D table called a ***bigram table***, as shown in Figures 13 and 14.

$j = $'d'

$i = $'c' →	-	a	b	c	d
-	0	8	2	3	4
a	6	3	4	1	6
b	3	3	1	1	0
c	4	1	1	0	1
d	4	5	0	2	1

or

$j = $'d'

$i = $'c' →	-	d	b	c	a
-	0	4	2	3	8
d	4	1	0	2	5
b	3	0	1	1	3
c	4	1	1	0	1
a	6	6	4	1	3

Figure 13: Bigram Frequencies (Tallies)

$j = $'d'

$i = $'c' →	-	a	b	c	d
-	0	13	3	5	6
a	9	5	6	2	9
b	5	5	2	2	0
c	6	2	2	0	2
d	6	8	0	3	2

or

$j = $'d'

$i = $'c' →	-	d	b	c	a
-	0	6	3	5	13
d	6	2	0	3	8
b	5	0	2	2	5
c	6	2	2	0	2
a	9	9	6	2	5

Figure 14: Bigram Frequencies (Percentages (%))

Using the bigram table, how do you store and access particular frequencies? Let the notation $freq_{i,j}$ indicate a frequency stored in the bigram table located at row i and column j:

- Row i indicates the first letter in a pair.
- Column j indicates the second letter in a pair.

In Figures 13 and 14, the i labels are to the left of the j labels, just like how they appear in words. So, you may express the pair i, j as, "character i before character j". More formally, determine the frequencies of pairs of characters with the following formula:

$$freq_{i,j} = \frac{\text{number of times the pair } i,j \text{ appears}}{\text{total number of pairs}}.$$

For instance, $freq_{\text{'c','d'}}$ refers to the frequency 2% located at row 'c' and column 'd' in Figure 14. Other examples include $freq_{\text{'a','-'}} = 9\%$, $freq_{\text{'-','a'}} = 13\%$, and $freq_{\text{'a','b'}} = 6\%$. Some observations you should note:

- $freq_{i,j}$ doesn't necessarily equal $freq_{j,i}$ because the pairs might occur a different number of times.
- $freq_{\text{'-','-'}}$ is 0 because the example has no double-spaces.
- By convention, we require the labels on the top be in the same order as the labels to the left.
- The bigram tables in Figures 13 and 14 are equivalent.
- Accounting for roundoff-error, adding up frequencies in each row or column in the bigram table yields the frequencies in the unigram table. For example, in Figure 13, the sum of column 'b' produces 2+4+1+1+0 = 8 and the sum of row 'b' produces 3+3+1+1+0 = 8, which matches the unigram tally of 'b' in Figure 11. Similarly, in Figure 14, the sum of column 'b' produces $3\% + 6\% + 2\% + \%2 = 13\%$ and the sum of row 'b' produces $5\% + 5\% + 2\% + 2\% \doteq 13\%$, which matches the unigram percentage of 'b' in Figure 12.

3.6 Example of Bigrams

The string "abccc" has bigrams "-a", "ab", "bc", "cc", "cc", and "c-". Where did the "-a" and "c-" come from? We insert spaces at the start and the end of each line of text to mark the start of the first word and end of the last word in each line of text. Figure 14 collects and organizes the tallies of unigrams and bigrams for "abccc". Observe that the unigram tallies are equal to the row and column sums of the

Bigram

	-	a	b	c
-	0	1	0	0
a	0	0	1	0
b	0	0	0	1
c	1	0	0	2

Unigram

-	a	b	c
1	1	1	3

Unigram

-	1
a	1
b	1
c	3

6 Total

Figure 15: Tallies for Example String "abccc"

bigram tallies. Further observe that the total number of unigrams is equal to the total number of bigrams. You might wonder why the number of spaces is 1. This is because we treat the inserted spaces at the front and end of each line of text as half-spaces or shared spaces; this also makes the bigram and unigram tables match.

3.7 Enciphering and Deciphering

What happens to the frequencies when plaintext is scrambled after applying an encryption key? Figure 17 shows the ciphertext generated by encrypting the plaintext from Figure 10 using the encryption key shown in Figure 16.

```
a b c d
c b d a
```

Figure 16: Example Encryption Key

```
-a-aaa-abba-abc-ac-ad-ada-add-baa-bad-cab-cb-cdc-dab-dad-dada-dc-   plaintext
-c-ccc-cbbc-cbd-cd-ca-cac-caa-bcc-bca-dcb-db-dad-acb-aca-acac-ad-   ciphertext
```

Figure 17: Encoded Text for Figure 10

What happened to the frequencies? Inspect the unigram and bigram tables of frequency tallies in Figures 18 and 19. As expected, the tables do change. The numbers appear to change, but do they really? For instance, yes, $freq_{'a'}$ changes from 20 to 12 in unigram table. However, notice that number 20 still appears, but now is $freq_{'c'}$. In retrospect, this is not surprising: the encryption key maps each 'a' to 'c'. So, the old frequency of 'a' is the new frequency of 'c'. Similarly, in the bigram table, $freq_{'d','a'}$ in the plaintext table moves to $freq_{'a','c'}$ in the ciphertext table because the encryption key simultaneously maps 'd' to 'a' and 'a' to 'c'. That is, enciphering "scrambles" frequencies by rearranging them. Decryption maps the ciphertext back to plaintext, restoring the frequencies back to their positions in the original table. Thus, decryption not only "unscrambles text", but also "unscrambles frequencies".

plaintext

-	a	b	c	d
17	20	8	7	12

ciphertext

-	a	b	c	d
17	12	8	20	7

Figure 18: Unigram Tallies

plaintext

	-	a	b	c	d
-	0	8	2	3	4
a	6	3	4	1	6
b	3	3	1	1	0
c	4	1	1	0	1
d	4	5	0	2	1

ciphertext

	-	a	b	c	d
-	0	4	2	8	3
a	4	1	0	5	2
b	3	0	1	3	1
c	6	6	4	3	1
d	4	1	1	1	0

Figure 19: Bigram Tallies

4 Using Frequencies for Decryption

4.1 Recap – Where are we, and where are we going?

The previous section ended with the following key observations:

- We say that the original frequencies of plaintext are ***unscrambled***.
- Enciphering rearranges, or ***scrambles***, frequencies.
- Deciphering restores the original arrangement of, or ***unscrambles***, frequencies.

We can restate the last observation as, if ciphertext is deciphered, then frequencies will be "unscrambled", i.e. frequencies will be restored to the original positions they had for plaintext. Three questions now arise:

1. Can we turn that if-then statement around? That is, does unscrambling ciphertext frequencies suffice to decipher ciphertext?
2. If so, given that we don't know the plaintext, how do we recognize/identify that a table of frequencies is unscrambled?
3. When we rearrange frequencies to unscramble them, what kinds of rearrangements are both legal and effective?

To answer these questions, we need to know more about frequencies of actual plaintexts and to study the effects of encryption on frequencies. To help you keep track of things, we will periodically summarize where we are and where we are going with a Roadmap.

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
→	Section 4.2	Q: Does "unscramble ciphertext frequencies" suffice to "decipher ciphertext"?
	Section 4.3	Q: How do we recognize that a table of frequencies is unscrambled?
	Section 5	Q: What are legal and effective ways to rearrange frequencies?

4.2 Decryption and Frequencies: How Might Frequencies Help?

Let A stand for *text is decrypted*. Let B stand for *frequencies are unscrambled*. By definition, unencrypted text has unscrambled frequencies. Therefore, if A is true, then B must be true. This can be suggestively written in formal, logical notation as $A \Rightarrow B$.

What we are now wondering about is whether we can turn things around: Is $A \Leftarrow B$ true? That is, if we somehow make B become true, then does that necessarily mean we made A become true?

Unfortunately, a little thought shows us that the answer is *no*! For example, consider an encryption key that does nothing except rename `'a'` to `'t'` and vice versa. The effect on unigram frequencies of applying this

key to text is to swap the frequencies of 'a' and 't'. Since plaintext "a theta" and ciphertext "t aheat" have exactly the same unigram frequencies, both have unscrambled unigram frequencies. (But note that the bigram frequencies are different.)

However, we should not give up hope because maybe, if we somehow make B become true —especially if we look at bigram frequencies— then although in theory we aren't *guaranteed* that we've made A become true, *perhaps* in practice we are *very likely* to have made A become true. This turns out to be the case, an *empirical* "fact" that you will verify in Project 7.

In order to apply this idea, given a table of frequencies, we have to be able to recognize that it is unscrambled. We consider how to do that in the next subsection.

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
→	Section 4.3	Q: How do we recognize that a table of frequencies is unscrambled?
	Section 5	Q: What are legal and effective ways to rearrange frequencies?

4.3 Recognizing Unscrambled Frequencies

To use the idea "unscramble frequencies to crack a cryptosystem", we need to study unscrambled frequencies (frequencies of plaintext) to look for recognizable structures/patterns that identify frequencies as being unscrambled.

So, let us look at plaintexts from a wide range of areas: If we look at a narrow range, then we might draw conclusions that apply to only a narrow range. So, let us consider the following three plaintexts:

- *Genesis* from the 1970 edition of *The New English Bible*, available at

  `http://etext.library.cornell.edu/cgi-bin/bie-idx?type=HTML&byte=116542360&rgn=book`

- An edition from 1597 of William Shakespeare's *Romeo and Juliet*, available at

  `http://etext.library.cornell.edu/cgi-bin/shakeEd-idx?type=HTML&byte=186196093&rgn=play`

- A slightly old version of the Announcements page for CS100, available at

  `http://courses.cs.cornell.edu/cs100/2000sp/oldannounce.txt`

Each of these documents contains a ***corpus*** or large plaintext. To compute frequency tables, we must deal with both upper- and lower-case letters, punctuation, and possibly accents. We defer the details of what we do to Section 4.3.1, and move immediately to Figure 20 for the unigram frequencies of these corpuses. Amazing!

Unigram Frequencies (%)

		-	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
Romeo and Juliet	(95K)	20	6	1	2	3	11	2	1	5	5	0	1	4	3	5	7	1	0	5	5	7	3	0	2	0	2	0
Genesis	(185K)	20	7	2	2	4	10	2	1	6	5	0	1	3	2	5	6	1	0	5	5	7	2	1	2	0	2	0
Announcements	(17K)	18	5	1	3	3	11	2	2	3	5	0	1	3	2	5	7	2	0	5	6	7	3	1	2	0	1	0

Figure 20: Unigram Frequencies for Three English Corpuses

These wildly different plaintexts all have very similar frequencies! (Although not shown for space reasons, the bigram frequencies are also very similar.) This might lead us to postulate that English has frequencies that are ***intrinsic*** or inherent to its very nature, and indeed this is the case: English has intrinsic frequencies that large plaintext is very likely to "closely" approximate. Therefore, a good approximation to intrinsic frequencies can be obtained from just about any corpus.

If you try other natural languages, you will discover that they, too, have intrinsic frequencies. This means that as long as we continue to use nothing specific to English we will develop an algorithm that works for natural languages besides English.

Recall that we studied plaintext frequencies to find a way to recognize "unscrambled frequencies". Since we have seen that frequencies of large plaintext are "close" to intrinsic frequencies, we can use that as our criterion: If frequencies are "close" to intrinsic frequencies, then we will consider/assume them to be "unscrambled". Thus, *unscramble frequencies* means *bring "close" to intrinsic frequencies*. This has three immediate consequences:

- We need to figure out how to define what "close" means precisely enough for use in a computer program.
- We need a way to compute intrinsic frequencies. But as we've already observed, we can simply use the frequencies from just about any large plaintext, since it is very likely to have frequencies that closely approximate intrinsic frequencies. A large plaintext used in this fashion is called ***training text*** because it "teaches" our program what the right answer is.
- We need medium to large size ciphertext so that its unscrambled frequencies are close to intrinsic frequencies: Small plaintext, like words or brief sentences, tends to not contain enough letter combinations to be representative of intrinsic frequencies.[4]

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
(→)	Section 4.3.1	Q: How do we compute frequencies for real texts, e.g. deal with punctuation?
	Section 4.4	Q: How do we measure "closeness" or ***distance***?
	Section 5	Q: What are legal and effective ways to rearrange frequencies?

4.3.1 Assumptions and Conventions

Before we continue, note that when we gave Figure 20, we did not specify how we computed the data since text can have upper- and lower-case letters, accents (e.g. "coöperation"), punctuation, and other non-letters. Here are the assumptions and conventions we used:

- Map upper-case letters to lower-case letters.
- Substitute a blank space `'-'` for any character that is not a letter.
- Assume no accents.

4.4 Distance

We need a way of comparing two equal-sized tables for "similarity" or "closeness". Suppose you wish to compare the *Genesis* and *Romeo and Juliet* frequencies. As shown in Figure 21, a first step is to subtract the frequencies for each character. Since the sign (positive or negative) of the differences have nothing to do

	-	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
Romeo and Juliet	20	6	1	2	3	11	2	1	5	5	0	1	4	3	5	7	1	0	5	5	7	3	0	2	0	2	0
Genesis	20	7	2	2	4	10	2	1	6	5	0	1	3	2	5	6	1	0	5	5	7	2	1	2	0	2	0
Differences	0	1	0	0	1	-1	0	0	1	0	0	0	-1	0	0	-1	0	0	0	0	-1	-1	1	0	0	0	0
Magnitudes	0	1	0	0	1	1	0	0	1	0	0	0	1	0	0	-1	0	0	0	0	1	1	1	0	0	0	0

Figure 21: Unigram Differences and Magnitudes of Differences

with how big the differences are, take the absolute value of each difference to get its magnitude. The last

[4] Ouvroir de Litterature Potentielle ("Potential Literature Workshop") is a group about game-like methods of writing. Member Georges Perec wrote *La Disparition* without using the letter `'e'`. The English translation, *A Void*, by Gilbert Adair also has no `'e'`. Besides as a gimmick, Perec wrote without `'e'` for plot/thematic reasons. *Le Ton Beau De Marot: In Praise of the Music of Language*, by Douglas R. Hofstadter, discusses both the original and the translation.

row shows the magnitude of each difference. (The differences do not quite jibe with the original frequencies because of rounding.)

The list of magnitudes gives us an idea of closeness, but it is cumbersome to deal with so much data. With bigrams, the tediousness would be even worse. Therefore, we would like to combine the magnitudes into a single number that measures the ***distance*** between two equal-sized tables. There are many possibilities of what function to use for distance, usually all based on the magnitudes of the differences between corresponding elements. For example,[5]

- Maximum or L^∞ distance: maximum magnitude
- L^2 distance: square root of the sum of the squares of the magnitudes
- Manhattan or taxicab or L^1 distance: sum of the magnitudes

For simplicity, choose the last one, the L^1 distance d. Why? It's simple and, as you will discover, tends to work. To compute d, pick two unigram or bigram tables with frequencies you wish to compare. Let N be the size of the character set, including a space. The values may contain either percentages or tallies, but both tables must have the same type of frequency! Take each absolute value of the difference between each element from the pair of tables and add all differences together:

$$\text{unigram distance } d = \sum_j \left| freq_j^{\text{table1}} - freq_j^{\text{table2}} \right| \qquad \text{bigram distance } d = \sum_{i,j} \left| freq_{i,j}^{\text{table1}} - freq_{i,j}^{\text{table2}} \right|.$$

Your choice of table1 and table2 is irrelevant because of the absolute value. Index i ranges over all rows $1..N$; index j ranges over all columns $1..N$.

One question that we must answer is, what distances count as "close"?

4.4.1 A Note About Distance and Labels

One thing to note is that when we compute the distance between two tables, we ignore the labels: i, j range over rows and columns, not over labels.

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
(→)	Section 4.5	Q: What distances count as "close"?
	Section 5	Q: What are legal and effective ways to rearrange frequencies?

4.5 Closeness

To try to calibrate what distances count as close we can compute the distances between the large plaintexts we used to discover intrinsic frequencies. Figure 22 shows the unigram and bigram distances between the same three corpuses used for Figure 20. Figure 22 suggests that distances of "around" 20% or less count as "close" for unigram distances and distances of "around" 50% or less count as "close" for bigram distances, but "around" is still too vague. For example, should 70% count as close for bigram distances?[6] Therefore, we replace the imprecise goal *bring "close" to intrinsic frequencies* by the more specific goal *bring as close as possible to intrinsic frequencies.*

At this point, we should also perform a sanity check. We should check at least one example to see that frequencies for ciphertext are not "close" to intrinsic frequencies: If ciphertext frequencies are also "close" to intrinsic frequencies, then "closeness" is not a good criterion for recognizing unscrambled frequencies. We use the following as our sample ciphertext:

[5] These distances belong to the class of L^p (pronounced "ell-pee") norms: the pth root of the sum of the pth powers of the magnitudes. The limit as p goes to infinity is the L^∞ norm.

[6] When you do Project 7, you will discover that 70% does count as close. Although 70% does sound high, keep in mind that it is still relatively far away from the maximum possible distance of 200%.

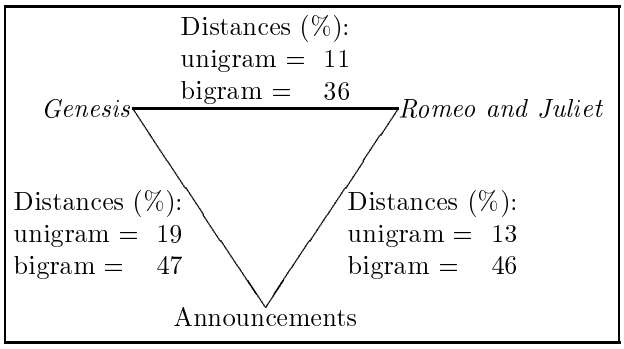


Figure 22: Unigram and Bigram Distances for Large Plaintexts

- Enciphered Announcements, the encryption of Announcements using the Caesar Cipher. Enciphered Announcements is available at

`http://courses.cs.cornell.edu/cs100/2000sp/cryptannounce.txt`

Figure 23 shows that the distances between our large plaintexts and Enciphered Announcements is indeed much larger than the distances between the plaintexts.

	Unigram Frequencies (%)																									Distance (%) from Enciphered Announcements	
	-	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	unigram	bigram
Romeo and Juliet	20	6	1	2	3	11	2	1	5	5	0	1	4	3	5	7	1	0	5	5	7	3	0	2	0	2	0	85	148
Genesis	20	7	2	2	4	10	2	1	6	5	0	1	3	2	5	6	1	0	5	5	7	2	1	2	0	2	0	82	150
Announcements	18	5	1	3	3	11	2	2	3	5	0	1	3	2	5	7	2	0	5	6	7	3	1	2	0	1	0	86	149
Enciphered Announcements	18	0	1	0	5	1	3	3	11	2	2	3	5	0	1	3	2	5	7	2	0	5	6	7	3	1	2	(0)	(0)

Figure 23: Unigram and Bigram Distances Between Ciphertext and Large Plaintexts

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
(→)	Section 5	Q: What are legal and effective ways to rearrange frequencies?

5 Attempts at Decryption

This section introduces progressively more sophisticated attempts at decryption. Although these attempts do not actually work —they produce incorrect results, are logically unsound, or run too slowly to be of any use— they are worth considering for the following three reasons:

- They rule out "obvious" approaches.
- They introduce techniques that we will use in our final attempt.
- They motivate (the need for) the more advanced techniques we end up adopting.

Before we look at specific attempts, let us consider ways to test them out:

- Try it out on ciphertext for which we already know the plaintext — that way we can see if the attempt worked.
- Try it out on plaintext. That is, what happens if we try to "decrypt" plaintext? It should be reasonably clear that any good decryption algorithm should leave plaintext unchanged, rather than scrambling it!

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
	Section 6.2	Q: What are legal and effective ways to rearrange frequencies?
(→)	Section 5.1	Q: Does sorting unigram frequencies work?

5.1 Sort Unigram Frequencies

Here is an "obvious" approach that you, like we did, might think of. Although the discussion in Section 4.2 already warns us it might not work, it is still worth trying. Form the encryption key as follows:

1. Sort unigrams on the top line by their frequency in training text.
2. Sort unigrams on the bottom line by their frequency in the ciphertext.

Let's try this approach on Enciphered Announcements and also apply the sanity check mentioned earlier of trying to "decrypt" plaintext. Figure 24 shows the alphabet for each corpus sorted in decreasing unigram frequency: Pairing the two lines of letters from any two corpuses yields a key. Figure 25 shows the resulting

Unigram Frequencies (%)

Romeo and Juliet	-	e	t	o	a	i	h	r	n	s	l	u	d	m	w	y	f	c	g	b	p	k	v	x	q	j	z
	20	11	7	7	6	5	5	5	5	5	4	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0	0
Genesis	-	e	a	t	h	o	n	s	r	i	d	l	m	u	w	y	f	c	b	g	p	v	k	j	z	x	q
	20	10	7	7	6	6	5	5	5	5	4	3	2	2	2	2	2	2	2	1	1	1	1	0	0	0	0
Announcements	-	e	t	o	s	i	a	n	r	l	u	h	d	c	m	p	w	g	f	y	b	v	k	x	j	q	z
	18	11	7	7	6	5	5	5	5	3	3	3	3	3	2	2	2	2	2	1	1	1	1	0	0	0	0
Enciphered Announcements	-	h	w	r	v	l	d	q	u	o	x	k	g	f	p	s	z	j	i	b	e	y	n	a	m	t	c
	18	11	7	7	6	5	5	5	5	3	3	3	3	3	2	2	2	2	2	1	1	1	1	0	0	0	0

Figure 24: Sorting Unigram Frequencies

distances between pairs of corpuses after their unigram frequencies have been sorted. Although it might look like we are partly successful, in fact these figures show us a disaster!

At first, it looks like we are partly successful if we look at only the sorted unigram frequencies for Announcements and Enciphered Announcements. The unigram and bigram distances are 0, which is as close as you can get. Furthermore, you can read off the encryption key (top line `"etosian..."`, bottom line `"hwrvld..."`) from the labels! That is, we have discovered the following important potential fact:

- If the frequency tables for plaintext and ciphertext match, then the labels from the frequency tables form the top and bottom lines of the encryption key.

However, the fact that pairing ciphertext and its plaintext works tells us very little: In general, we do not have the plaintext decryption of the ciphertext. Therefore, to evaluate our attempted approach, we must look at other pairings of our texts. These other pairings constitute the sanity check from Section 4.5 of trying to "decrypt" plaintext.

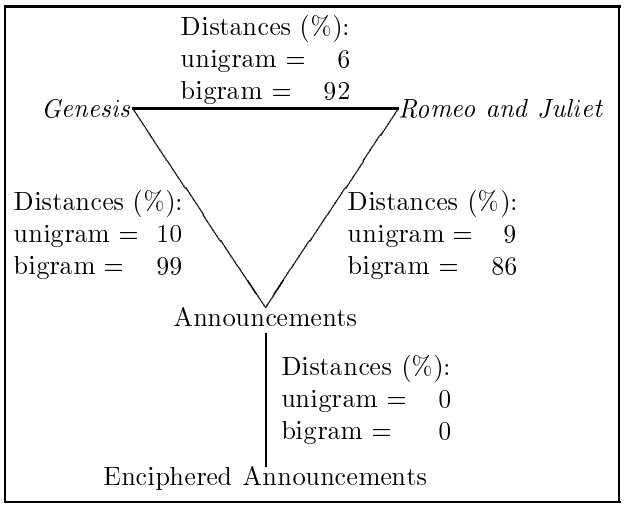


Figure 25: Distances after Sorting Unigram Frequencies

Observe that for all other pairs of plaintexts, letters are not properly matched up, e.g. 't in *Romeo and Juliet* is matched in *Genesis* to 'a instead of 't. Also observe that although the distance between unigram frequencies (percentages) went down as a result of sorting —indeed, sorting makes unigram tables as close as possible— we got a bogus encryption/decryption key. That is, since we "decrypted" plaintext, the key should identical top and bottom lines since no letters are renamed. However, the key we get by reading off the labels has top line `etoai...` and bottom line `eatho...`: attempting to "decrypt" *Genesis* with this key would yield gibberish. These results pretty conclusively show that using unigram frequencies will not work.[7]

However, observe that it "almost" works. Although the two lines of labels should are not identical as they should be, i.e. no letters should be renamed,the two lines are not "too different". That is, not every letter is matched with itself, but letters are not "too far" from where they should be, e.g. 'a in *Romeo and Juliet* is "two positions" away from the 'a in *Genesis*. An experienced human could probably take the "almost-solution" and play around with it to get to the correct solution by thinking of higher-order patterns. For example, vowels and consonants tend to alternate and we're pretty sure that 'u will follow 'q. But, higher-order frequencies reflect higher-order patterns, and bigrams are the next higher-order pattern after unigrams! And indeed, if you look at the L^1 distance between bigram frequencies (percentages), you'll see that our attempted approach made these distances go up, indicating that we are moving farther away from a solution. Thus, it makes sense to base our next attempted approach on bigram frequencies.

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
	Section 5	Q: What are legal and effective ways to rearrange frequencies?
	Section 5.1	Sorting unigram frequencies does not work, but "almost" does (Hope) When the frequency table for ciphertext matches the table for training text, read the encryption key off of the labels
(→)	Section 5.2	Q: Does sorting bigram frequencies work?

[7]Actually, since different letters probably do have at least slightly different intrinsic unigram frequencies, if we have "large enough" training and ciphertexts, then their frequencies will converge to identical, intrinsic frequencies, in which case sorting unigram frequencies would work. The problem is, our experiments show that "large enough" appears to be on the order of multiple books or more.

5.1.1 Sorting implies Swapping

In our attempted approach above, we said to "sort letters according to their unigram frequencies". How would we write a program to do that? We need to sort frequencies and keep track of the corresponding letters. Thus, we would use two parallel arrays: letters on top, frequencies on bottom. We would sort the frequencies, e.g. using selection sort, and every time we swap two frequencies, then we would swap the two corresponding letters.

Swapping will turn out to be extremely important throughout our attempts.

5.2 Sort Bigram Frequencies?

Sorting unigram frequencies almost worked, and we have reason to believe using bigram frequencies would work better. Therefore, a natural first reaction is to consider the following:

1. Sort bigrams in the top line by their frequency in training text.
2. Sort bigrams in the bottom line by their frequency in ciphertext.
3. Somehow read off the encryption/decryption key.

Unfortunately, "sort bigrams by frequency and read off the key" is problematic. For example, what should we do in the situation pictured in Figure 26? Matching `"th"` with `"ab"` suggests `'h'`→`'b'` (and `'t'`→`'a'`) is

	most frequent bigram	second most frequent bigram
training text	`"th"`	`"he"`
ciphertext	`"ab"`	`"cd"`

Figure 26: Attempt to Sort Bigrams by Frequency and Read Off the Key

part of the encryption key, whereas matching `"he"` with `"cd"` suggests `'h'`→c is part of the encryption key, which is a contradiction: A key must map each letter to exactly one letter. A key is not allowed to map `'h'` to both `'b'` and `'c'`.

One intuition as to why this approach is problematic is that with sorting unigram frequencies, swapping two frequencies corresponds to swapping two letters. However, with sorting bigram frequencies, swapping two frequencies does *not* correspond to swapping two letters: Swapping two letters affects a whole *set* of associated frequencies.

In Section 6.3 we will carefully investigate how swapping two letters affects the table of bigram frequencies. However, for now, we will suppress the details, except to point out that that there are details that must be considered, and rule out "sort bigram frequencies" as a simple approach.

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
	Section 5	Q: What are legal and effective ways to rearrange frequencies?
	Section 5.1	Sorting unigram frequencies does not work, but "almost" does. (Hope) When the frequency table for ciphertext matches the table for training text, read the encryption key off of the labels
	Section 5.2	"Sort bigram frequencies" is problematic.
(→)	Section 5.3	Q: Why not try all decryption keys?

5.3 Exhaustive Search

Our goal can be formulated as follows: Find a decryption key that when applied to ciphertext unscrambles bigram frequencies. This is a search, and one fundamental search algorithm is to try all possibilities:

- Enumerate (generate) *all* ***candidate*** (possible or potential) decryption keys.
- Apply each key to ciphertext.
- Compute the distance between the resulting frequencies and intrinsic frequencies.
- Keep track of the "best" key, i.e. the key that brings us closest to intrinsic frequencies.
- Use the "best" key as the decryption key.

We can enumerate all keys by progressively listing more and more keys. This algorithm is shown in Figure 27. (If you wish to see a justification for this algorithm to enumerate all keys, see Section 6.2.)

- Start with any single key on the list, e.g. with the sorted alphabet as both its top and bottom lines.
- For each key on the list, include all keys obtainable by *swapping* —we told you swapping would be important!— two letters in the bottom line.
- Repeat until the list stops changing.

Figure 27: Algorithm to Enumerate All Keys via Swaps

For instance, suppose you have a three element character set 'a', 'l', 'n'. Figure 28 shows all possible decryption keys for this character set — only the bottom lines are shown and a↔n means "swap 'a' with 'n'".

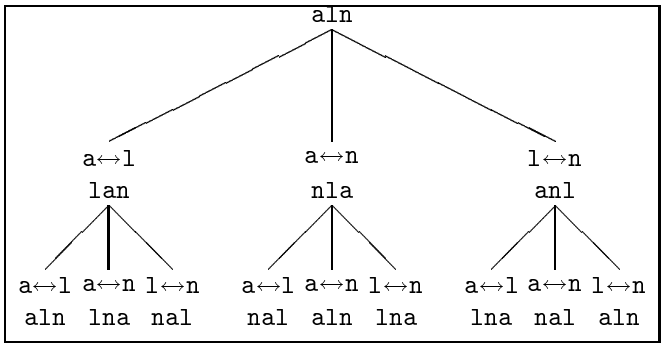


Figure 28: Using the Algorithm in Figure 27 to List of All Keys of aln

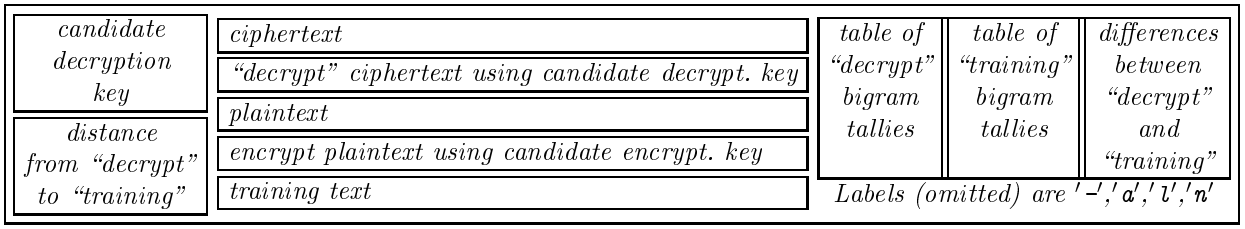
Suppose you need to decrypt a ciphertext that uses this three-letter character set and a '-' for spaces. Apply all candidate decryption keys from Figure 28 to the ciphertext.[8] Each time you apply a different key to the ciphertext, you generate something you hope is plaintext. Collect the frequencies from the hoped-for-plaintext. Choose as the decryption key and the plaintext the key and text with frequencies closest to intrinsic frequencies.[9] Figure 29 shows this exhaustive search, and note that indeed the minimum distance 15 corresponds to the decrypted ciphertext.

For "small" alphabets, we can reasonably perform an exhaustive search. However, what if the language has a relatively "large" character set? Consider English, which has 26 letters, not counting punctuation. There are $26! \doteq 4 \times 10^{26}$ decryption keys for a 26-letter alphabet! Machines these days cannot perform more than about a billion (10^9) operations per second, so even if they could look at a billion ciphers per second, it would still take 4×10^{17} seconds, or over 10^{10} years! So, you will have to find a better approach to crack EGOP's message since you have until May 4, 2000!

Therefore, we must use an *incomplete* search that looks at only a (small) subset of the keys. We answer the questions "Which keys?" and "How do you pick them?" in the next section.

[8] Conceptually, we have said to compute the table of bigram frequencies for each key by transforming ciphertext and then counting. We will see in Section 6.3 how we instead rearrange the frequencies in the original bigram table, thereby allowing us to omit the step of transforming ciphertext.

[9] A human could visually inspect the hoped-for-plaintext and simply choose the one that is readable. However, without more complex algorithms that involve pattern recognition, a computer program has "trouble" visually inspecting text. Therefore, we stick to our idea of bringing the frequencies as close as possible to intrinsic frequencies.



Decrypt. Key			Decrypt	Training	Differences
aln	cipher	n na nanl naa nl nll nllna anln			
aln	decrypt	n na nanl naa nl nll nllna anln	0 1 0 7	0 6 1 1	0 5 1 6
	plain	a al alan all an ann annal lana	3 1 0 2	5 2 3 5	2 1 3 3
Distance	encrypt	a al alan all an ann annal lana	3 0 2 2	0 4 1 0	3 4 1 2
39	training	aaa ala alan allan ana anna la nan	2 4 5 0	3 3 0 1	1 1 5 1

Decrypt. Key			Decrypt	Training	Differences
aln	cipher	n na nanl naa nl nll nllna anln			
anl	decrypt	l la laln laa ln lnn lnnla alnl	0 1 7 0	0 6 1 1	0 5 6 1
	plain	a al alan all an ann annal lana	3 1 2 0	5 2 3 5	2 1 1 5
Distance	encrypt	a an anal ann al all allan nala	2 4 0 5	0 4 1 0	2 0 1 5
35	training	aaa ala alan allan ana anna la nan	3 0 2 2	3 3 0 1	0 3 2 1

Decrypt. Key			Decrypt	Training	Differences
aln	cipher	n na nanl naa nl nll nllna anln			
lan	decrypt	n nl nlna nll na naa naanl lnan	0 0 1 7	0 6 1 1	0 6 0 6
	plain	a al alan all an ann annal lana	3 2 0 2	5 2 3 5	2 0 3 3
Distance	encrypt	l la laln laa ln lnn lnnla alnl	3 0 1 2	0 4 1 0	3 4 0 2
37	training	aaa ala alan allan ana anna la nan	2 5 4 0	3 3 0 1	1 2 4 1

Decrypt. Key			Decrypt	Training	Differences
aln	cipher	n na nanl naa nl nll nllna anln			
nal	decrypt	l ln lnla lnn la laa laaln nlal	0 0 7 1	0 6 1 1	0 6 6 0
	plain	a al alan all an ann annal lana	3 2 2 0	5 2 3 5	2 0 1 5
Distance	encrypt	l ln lnla lnn la laa laaln nlal	2 5 0 4	0 4 1 0	2 1 1 4
33	training	aaa ala alan allan ana anna la nan	3 0 2 1	3 3 0 1	0 3 2 0

Decrypt. Key			Decrypt	Training	Differences
aln	cipher	n na nanl naa nl nll nllna anln			
lna	decrypt	a al alan all an ann annal lana	0 7 1 0	0 6 1 1	0 1 0 1
	plain	a al alan all an ann annal lana	2 0 4 5	5 2 3 5	3 2 1 0
Distance	encrypt	n na nanl naa nl nll nllna anln	3 2 1 0	0 4 1 0	3 2 0 0
15 *	training	aaa ala alan allan ana anna la nan	3 2 0 2	3 3 0 1	0 1 0 1

Decrypt. Key			Decrypt	Training	Differences
aln	cipher	n na nanl naa nl nll nllna anln			
nla	decrypt	a an anal ann al all allan nala	0 7 0 1	0 6 1 1	0 1 1 0
	plain	a al alan all an ann annal lana	2 0 5 4	5 2 3 5	3 2 2 1
Distance	encrypt	n nl nlna nll na naa naanl lnan	3 2 2 0	0 4 1 0	3 2 1 0
17	training	aaa ala alan allan ana anna la nan	3 2 0 1	3 3 0 1	0 1 0 0

Figure 29: Distances for Each Candidate Decryption Key: Minimum 15 is Marked with "∗"

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
	Section 5.1	Sorting unigram frequencies does not work, but "almost" does. (Hope) When the frequency table for ciphertext matches the table for training text, read the encryption key off of the labels
	Section 5.2	"Sort bigram frequencies" is problematic.
	Section 5.3	***Exhaustive*** search is too slow; therefore, need an incomplete search.
(→)	Section 6.2	Q: How do we perform an effective, *incomplete* search?

6 Improving Decryption Methods with Swaps

Recall that it is infeasible to exhaustively search over all the keys, so we must look at a small subset of all the possible keys, i.e. perform an incomplete search. To motivate our incomplete search algorithm, let us take another look at the exhaustive search from Figure 29 in the previous section.

6.1 Introduction to "Hill-Climbing"

Figure 30 summarizes the distances of bigram tables for each key for Figure 29, reorganized as follows. The corners of the "hexagon" indicate each key with its corresponding bigram distance nearby. Two keys/corners are connected with a line if each key is obtainable from the other key by swapping two letters.

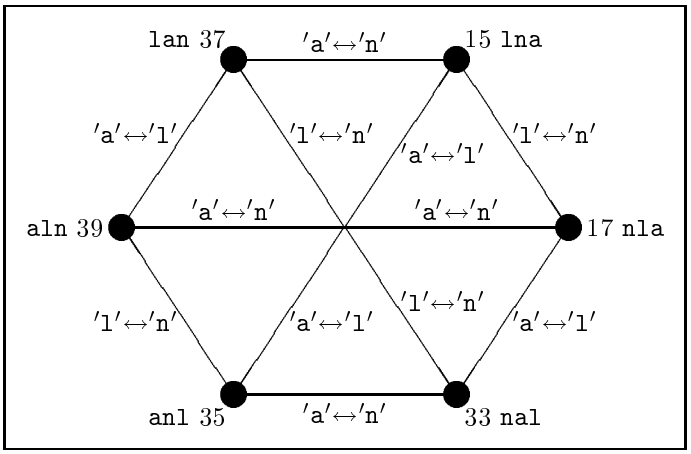


Figure 30: Reformulation of Keys and Distances from Figure 29

Rather than inspecting each key, here is a modifed approach. Consider trying to reach the best key `"lna"` from the starting key `"aln"` by traveling along a "path" in Figure 30. One "path" is from `"aln"` to `"nla"` to `"lna"`, which has the property that from each key we go to its best "neighboring" key:

- start at `"aln"`, which has distance 39.
  - from `"aln"`, swap `'a'` and `'l'` to get to neighboring key `"lan"`, which has distance 37.
  - from `"aln"`, swap `'a'` and `'n'` to get to neighboring key `"nla"`, which has distance 17.
  - from `"aln"`, swap `'l'` and `'n'` to get to neighboring key `"anl"`, which has distance 35.
- go the best neighbor `"nla"`, which has distance 17.
  - from `"nla"`, swap `'a'` and `'l'` to get to neighboring key `"nal"`, which has distance 33.

  - from `"nla"`, swap `'a'` and `'n'` to get to neighboring key `"aln"`, which has distance 39.
  - from `"nla"`, swap `'l'` and `'n'` to get to neighboring key `"lna"`, which has distance 15.

- go to the best neighbor `"lna"`, which has distance 15.

  - from `"lna"`, swap `'a'` and `'l'` to get to neighboring key `"anl"`, which has distance 35.
  - from `"lna"`, swap `'a'` and `'n'` to get to neighboring key `"lan"`, which has distance 37.
  - from `"lna"`, swap `'l'` and `'n'` to get to neighboring key `"nla"`, which has distance 17.

- all the neighbors are worse, so stay at `"lna"` and stop looking.

This approach of going to the best neighbor is ***hill-climbing*** and is fleshed out in Section 7.1. In the meantime, note the following:

- We do not have to skip keys we've already seen. (Doing so takes time and space and doesn't always save time in the long run.)
- We do not have to store all the keys or even all the neighbors: Compute the keys as needed and "throw away" the rest.
- You can get from "anywhere" to "anywhere". That is, you can get from every key in Figure 30 to every other key in Figure 30 by a path of 0 or more swaps.

Swaps keep coming up! This suggests we should take a closer look at swaps.

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
	Section 5.1	Sorting unigram frequencies does not work, but "almost" does. (Hope) When the frequency table for ciphertext matches the table for training text, read the encryption key off of the labels
	Section 5.2	"Sort bigram frequencies" is problematic.
	Section 5.3	***Exhaustive*** search is too slow; therefore, need an incomplete search.
	Section 6.2	Q: How do we perform an effective, *incomplete* search?
	Section 6.1	(Idea) Hill-Climbing: pick the best "neighbor"
(→)	Section 6.2	Why swaps are so important, e.g. used to generate "neighbors"

6.2 Why Swaps Are so Important

We've tried to motivate the importance of swaps since Section 5.1, where our first decryption attempt involved sorting unigram frequencies: Most sorting algorithms are based on swaps. We saw swaps again when considering enumerating all keys for an exhaustive search, and yet again in our tentative exploration of hill-climbing. Why are swaps so important, e.g. why are they used for sorting? First of all, *rearrangement* or *permutation* is a fundamental operation in all of these areas:

- Sorting = rearranging elements in a list in order.
- Bottom line of a key = a rearrangement of the top line.
- Generating all keys = generating all rearrangements of the top line for use in the bottom line.

Now that we see that permutations are important, we also have the following fundamental fact:

- "Transpositions, i.e. swaps, generate all permutations".

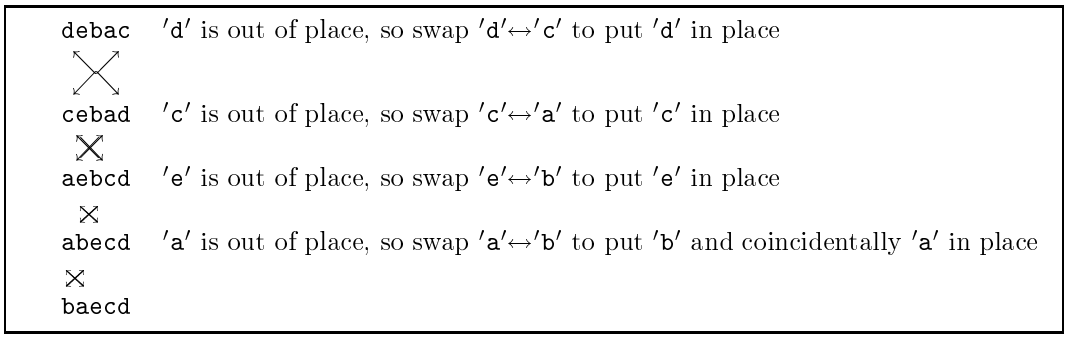


Figure 31: Using Swaps to get from "debac" to "baecd"

That is, every permutation can be modified into every other permutation by a "path" or sequence of 0 or more swaps, e.g. no matter what key we start with, we can reach the decryption key using a sequence of 0 or more swaps.

The justification for this fundamental fact is simply that we can sort using swaps: As long as an item is out of order, swap it into place; repeat until done. (Selection sort does these swaps in a specific order, e.g. right-to-left.) For example, Figure 31 shows one way to get from "debac" to "baecd" using swaps. So we know that we can always get from one permutation to another:

- Swaps can be used for sorting, because no matter what order the numbers are in, it is possible to swap them into sorted order.
- Our algorithm for generating all keys (permutations) will work because swaps generate all permutations.
- Our hill-climbing algorithm is not inherently prevented from reaching the decryption key: No matter what key we start with, there is a path of 0 or more swaps to the decryption key.[10]

Now that we understand why swaps are important, let us investigate them further: What effect do swaps have on bigram tables?

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies
		Approximate ***intrinsic*** frequencies with ***training text***
		Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
	Section 5.1	Sorting unigram frequencies does not work, but "almost" does.
		(Hope) When the frequency table for ciphertext matches the table for training text, read the encryption key off of the labels
	Section 5.2	"Sort bigram frequencies" is problematic.
	Section 5.3	***Exhaustive*** search is too slow; therefore, need an incomplete search.
	Section 6.2	Q: How do we perform an effective, *incomplete* search?
	Section 6.1	(Idea) Hill-Climbing: pick the best "neighbor"
	Section 6.2	Why swaps are so important, e.g. used to generate "neighbors"
(→)	Section 6.3	The effect of swaps on bigram tables

6.3 The Effects of Swaps on Bigram Tables

Let us look at the effect of swapping two letters in a key on bigram frequency tables. In Figure 33 we take some plaintext through a series of Stages. Figure 32 shows the plaintext, summarizes the Stages, and includes the encryption keys that could be used to transform the plaintext into the text at each Stage.

[10] However, it is true that hill-climbing can get "stuck" at a ***local optimum***, where all the "local" neighbors are farther away from training frequencies, so we will not reach the ***global optimum***, the solution that is closest overall to training frequencies.

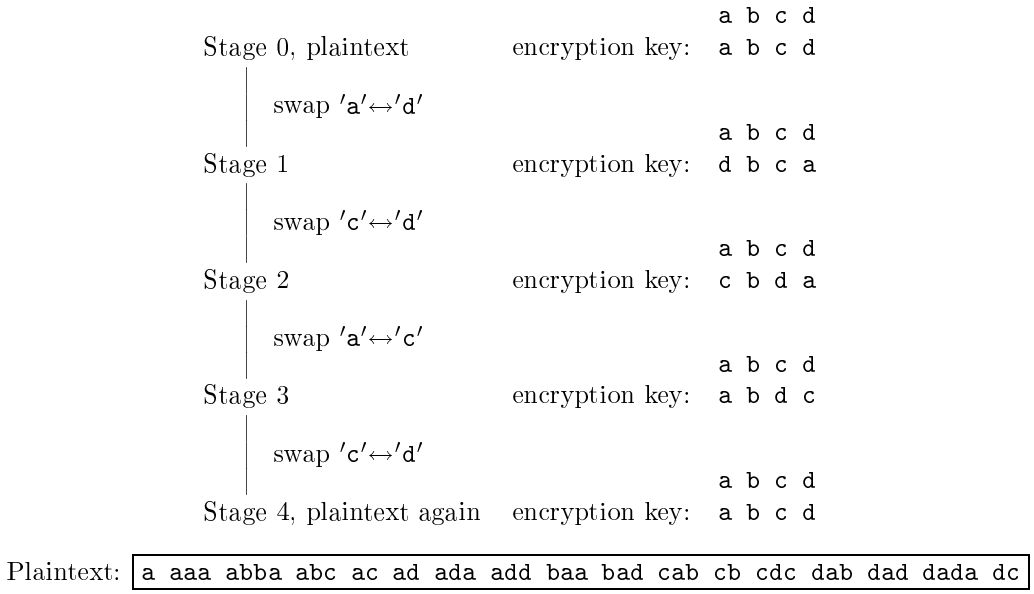


Figure 32: The Pipeline of Stages that Plaintext Goes Through

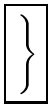
Figure 33 has a lot of information. Let us look at it a bit at a time.

At each Stage, we show the text so far and three pairs of unigram and bigram frequencies. These form three separate columns (left, middle, right) of frequency tables. For brevity, we use tallies instead of percentages. Verify the observation from Section 3.5 that adding up the tallies in each row or each column in the bigram table yields the tallies in the unigram table. (This property is also true for percentages.)

The left and middle columns contain the same frequency information —the unigram and bigram frequencies of the text so far— but organize it slightly differently; we will explain the right column in Section 6.4. A little terminology at this point will be helpful. Let us call the frequencies, the numbers themselves, the ***contents*** of a table, as distinguished from the the character ***labels***.

- The left column leaves the contents of the unigram and bigram tables unchanged, but swaps labels. This directly reflects two letters getting renamed to each other.
- The middle column leaves the labels unchanged, but rearranges the contents of the unigram and bigram tables to reflect the new frequencies.

Take a careful look to see that the left and middle columns *do* contain the same frequency information, namely the unigram and bigram tallies for the text so far. For example, in Stage 1, `"-d"` is shown to occur 8 times in both the left and middle bigram tables and also in the ciphertext.

Now take a closer look at *how* the contents of the tables in the middle column are rearranged. On the far right between each stage, we have a large brace  labeled with the following information: the two letters that were swapped, and a picture of the changes to bigram table in the middle column — entries that are changed are drawn as `xs`. Observe that when two letters are swapped, the effect on the tables in the middle column are to swap the corresponding rows and columns. This makes sense: If two letters are swapped (renamed to each other), then their corresponding frequencies get swapped, e.g. if `'a'` and `'d'` are swapped, then the frequencies of `"ab"` and `"db"` are swapped (rows `'a'` and `'d'` are swapped), as are the frequencies of `"ba"` and `"bd"` (columns `'a'` and `'d'` are swapped).

This tells us how to quickly (re)compute bigram frequencies when we swap two letters in text. Instead of recounting all the bigram frequencies by scanning the text, we merely take the old bigram table and swap the corresponding two rows and swap the corresponding two columns!

So, we have learned the effects of swaps on bigram frequency tables.

However, there is still more information to be learned from Figure 33.

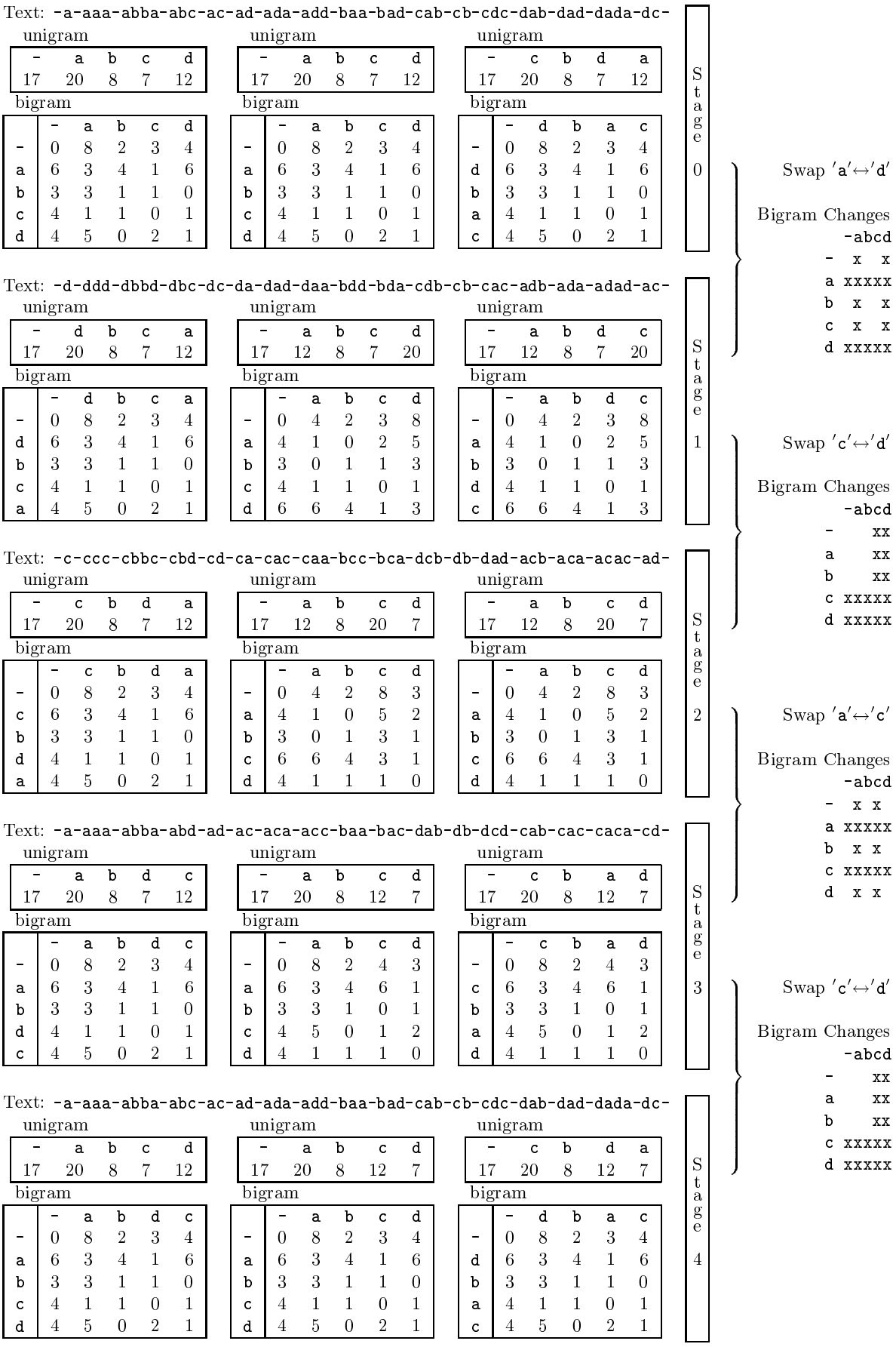


Figure 33: Effects of Swaps on Bigram Frequencies

6.4 Continuing to Investigate Figure 33, e.g. The Mysterious Right Column

Here are some further observations about Figure 33.

- As suggested in Section 5.1, we can read the bottom line of canonical (sorted top-line) encryption keys shown in Figure 32 off of the labels of the tables in the left column of Figure 33. Why? Well, if we unscramble frequencies, then the ciphertext labels should match up with the plaintext labels.
- Again, as in Section 5.1, in this example, we were matching frequencies for ciphertext to frequencies for plaintext, which is an unusual situation: Usually, we do not know the plaintext — that is why we need to crack the cryptosystem! Thus, normally we try to match ciphertext frequencies as best we can with training text frequencies.
- Aside: Here is a sanity check.

  Remember, the frequencies at each Stage are those of the ciphertext obtained by applying the encryption key to the plaintext. Note that there is nothing in that description about the path taken, i.e. which swaps were used and what order they were performed. Thus, the frequencies should depend on the key but not the path taken to reach the key. We see that at least in Figure 33 that this is indeed the case.

  Consider the ciphertext at Stage 2:

  ```
  c ccc cbbc cbd cd ca cac caa bcc bca dcb db dad acb aca acac ad
  ```

  Observe that Figure 33 shows two ways to reach the ciphertext from plaintext:

  $$\text{Stage } 0 \rightarrow \text{Stage } 1 \rightarrow \text{Stage } 2 \text{ and Stage } 2 \leftarrow \text{Stage } 3 \leftarrow \text{Stage } 4.$$

  Figure 33 also shows two ways to reach plaintext from the ciphertext:

  $$\text{Stage } 0 \leftarrow \text{Stage } 1 \leftarrow \text{Stage } 2 \text{ and Stage } 2 \rightarrow \text{Stage } 3 \rightarrow \text{Stage } 4.$$

  In all cases above, the path taken to reach Stage 2 or Stage 0/4 does not affect the resulting frequencies.
- At Stage 2, we can easily compute (count frequencies!) the middle table from the ciphertext but not the left table — figuring out the order of the labels is equivalent to decrypting!
- At Stage 2, the left table (which matches the table for plaintext) is (assumed to be) close to training text.
- The contents of the middle table are not close to the original table in Stage 0. Therefore, the middle table is not close to training text.

Thus, using bigram frequencies still looks hopeful, i.e. it looks like maybe ciphertext frequencies appear to (usually) be far from intrinsic frequencies.

Let us elaborate a bit on "read key off of the labels". Suppose we go back to Stage 2 and copy the middle table into the right column. What happens if, when we perform the swaps shown in the middle column, we modify *both* the labels and the contents? This is shown in the right column. Observe that *again*, no matter whether we take the path Stage 0 ← Stage 1 ← Stage 2 or Stage 2 → Stage 3 → Stage 4 we end up with a table that is (close to) the original, i.e. we have unscrambled frequencies. Furthermore, in both paths, we can indeed read off the encryption key!

Thus, we use swaps to generate and search for the best table, i.e. the table closest to the table of intrinsic frequencies (as approximated from training text). When we do a swap, we modify both the labels and the corresponding rows and columns of frequencies. (Conceptually, this means we do not rename letters during the search, merely reorganize the frequency information.) When we get a close match to training text — which we assume is close to plaintext— we read off the encryption/decryption key from the labels, which allows us to decipher the ciphertext.

It remains now only to clarify how to perform the search.

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
	Section 5.1	Sorting unigram frequencies does not work, but "almost" does. (Hope) When the frequency table for ciphertext matches the table for training text, read the encryption key off of the labels
	Section 5.2	"Sort bigram frequencies" is problematic.
	Section 5.3	***Exhaustive*** search is too slow; therefore, need an incomplete search.
	Section 6.1	(Idea) Hill-Climbing: pick the best "neighbor"
	Section 6.2	Why swaps are so important, e.g. used to generate "neighbors"
	Section 6.3	The effect of swaps on bigram tables
(→)	Section 7.1	Clarifying Hill-Climbing

7 Decryption by Optimization

The problem of optimization is to find (search for) the "best" solution, where we have the following two ingredients:

- Some way of measuring "best".
- Some way to generate all ***candidate*** solutions, i.e. all possible solutions.

We meet these two conditions: "best" is "closest in L^1 distance to training text" and Section 6.2 explained that swaps generate all possible keys. Therefore, we can look at known algorithms for solving optimization problems:

- We have already seen exhaustive search and eliminated it as being infeasible.
- Let us now look more closely at hill-climbing.

7.1 Hill-Climbing: Don't worry, be happy/optimistic: choose the best neighbor

From any given rearrangement, using a single swap produces a comparatively small set of additional candidate rearrangements. We think of these as ***neighbors*** that are reachable in one step. If you think about paths starting from the current candidate, including paths to the best solution, they all start off by taking a next step. The neighbors are exactly those next steps.

Recasting optimization in terms of paths and neighbors, we want to know how to pick a path —hopefully the shortest path, but we'll take what we can get— that takes us to the best solution.

We have seen earlier that we cannot simply try out all paths. Therefore, we must somehow throw away some possibilities. One way to do that is to be optimistic and pick the path that looks "best". What might make a path look good? Well, if it takes us closer to a solution. Therefore, we might simply decide to look at all the neighbors, and pick the best one. Ties may be broken arbitrarily. If the current solution is better than all the neighbors, then we're done. Otherwise keep taking steps until improvements are marginal or some time limit is reached.

Let us now apply hill-climbing to an example a little bit bigger than in Section 6.1. Figure 34 shows the training text and plaintext that we use, plus the encryption key we use to generate the ciphertext.

Figures 35 through 38 show the steps that hill-climbing takes. Keep in mind that we are trying to make ciphertext frequencies match the frequencies of the training text — pretend we don't know the plaintext and are trying to extract it by cracking the ciphertext. At each step, all the neighboring bigram tables are computed one at a time and their distances from the training frequencies computed. The one that is closest to training frequencies is chosen as the next table to continue searching from.

Figure 38 shows the final step, where we see that all neighbors of the table with labels "`raset`" are farther than it is from the training frequencies. This tells us the canonical (sorted top-line) encryption key is

Encryption Key: `aerst` / `raset`

Training text

```
aerate arrear arrest as
aster at ear east eat
era erase ere err estate
ester eta raster rate
rater sat seat see start
state stater stearate
strata stress tart tater
terse test treat tsar
```

Plaintext

```
a are area art ass
assert assess asset ate
attest ease eater errata
rare rat rear rest sea
sear seater set star
stare steer street tar
tartar taste tat tea
tear tease tee teeter
tree tress
```

Ciphertext

```
r rsa rsar rst ree
reeast reeaee reeat rta
rttaet area artas assrtr
srsa srt sars saet ear
ears eartas eat etrs
etrsa etaas etsaat trs
trstrs treta trt tar
tars tarea taa taatas
tsaa tsaee
```

	-	a	e	r	s	t
-	0	6	10	3	9	6
a	3	0	1	8	5	14
e	9	7	1	12	5	1
r	10	8	5	3	1	2
s	2	2	4	0	1	13
t	10	8	14	3	1	0

	-	a	e	r	s	t
-	0	10	3	4	8	11
a	5	0	0	12	7	7
e	9	10	5	6	4	4
r	10	3	9	1	0	3
s	3	0	9	0	6	7
t	9	8	12	3	0	1

	-	a	e	r	s	t
-	0	3	8	10	4	11
a	9	5	4	10	6	4
e	3	9	6	0	0	7
r	5	0	7	0	12	7
s	10	9	0	3	1	3
t	9	12	0	8	3	1

Distance from `aerst` to training text: 178

Figure 34: Training text, Plaintext, Encryption Key, and Ciphertext

hopefully `aerst` / `raset`, and *indeed it is!* Thus, our hope that unscrambling frequencies would lead to cracking ciphertext is (probably) validated!

Observe that we looked at 44 keys, counting duplicates (we looked at 36 unique keys), which is better than looking at all 5! = 120 keys. This again suggests that hill-climbing is effective in this situation at finding the best table, whereas exhaustive search would take much too long.

Swap 'a'↔'e'; 150

	-	e	a	r	s	t
-	0	8	3	10	4	11
e	3	6	9	0	0	7
a	9	4	5	10	6	4
r	5	7	0	0	12	7
s	10	0	9	3	1	3
t	9	0	12	8	3	1

Swap 'a'↔'r'; 150

	-	r	e	a	s	t
-	0	10	8	3	4	11
r	5	0	7	0	12	7
e	3	0	6	9	0	7
a	9	10	4	5	6	4
s	10	3	0	9	1	3
t	9	8	0	12	3	1

Swap 'a'↔'s'; 182

	-	s	e	r	a	t
-	0	4	8	10	3	11
s	10	1	0	3	9	3
e	3	0	6	0	9	7
r	5	12	7	0	0	7
a	9	6	4	10	5	4
t	9	3	0	8	12	1

Swap 'a'↔'t'; 146

	-	t	e	r	s	a
-	0	11	8	10	4	3
t	9	1	0	8	3	12
e	3	7	6	0	0	9
r	5	7	7	0	12	0
s	10	3	0	3	1	9
a	9	4	4	10	6	5

Swap 'e'↔'r'; 148

	-	a	r	e	s	t
-	0	3	10	8	4	11
a	9	5	10	4	6	4
r	5	0	0	7	12	7
e	3	9	0	6	0	7
s	10	9	3	0	1	3
t	9	12	8	0	3	1

Swap 'e'↔'s'; 150

	-	a	s	r	e	t
-	0	3	4	10	8	11
a	9	5	6	10	4	4
s	10	9	1	3	0	3
r	5	0	12	0	7	7
e	3	9	0	0	6	7
t	9	12	3	8	0	1

Swap 'e'↔'t'; 148

	-	a	t	r	s	e
-	0	3	11	10	4	8
a	9	5	4	10	6	4
t	9	12	1	8	3	0
r	5	0	7	0	12	7
s	10	9	3	3	1	0
e	3	9	7	0	0	6

Swap 'r'↔'s'; 144∗

	-	a	e	s	r	t
-	0	3	8	4	10	11
a	9	5	4	6	10	4
e	3	9	6	0	0	7
s	10	9	0	1	3	3
r	5	0	7	12	0	7
t	9	12	0	3	8	1

Swap 'r'↔'t'; 156

	-	a	e	t	s	r
-	0	3	8	11	4	10
a	9	5	4	4	6	10
e	3	9	6	7	0	0
t	9	12	0	1	3	8
s	10	9	0	3	1	3
r	5	0	7	7	12	0

Swap 's'↔'t'; 160

	-	a	e	r	t	s
-	0	3	8	10	11	4
a	9	5	4	10	4	6
e	3	9	6	0	7	0
r	5	0	7	0	7	12
t	9	12	0	8	1	3
s	10	9	0	3	3	1

Figure 35: Hill-Climbing, Step 1: Go from `aerst`, 178 to `aesrt`, 144, marked "∗".

Swap 'a'↔'e'; 138

	-	e	a	s	r	t
-	0	8	3	4	10	11
e	3	6	9	0	0	7
a	9	4	5	6	10	4
s	10	0	9	1	3	3
r	5	7	0	12	0	7
t	9	0	12	3	8	1

Swap 'a'↔'s'; 142

	-	s	e	a	r	t
-	0	4	8	3	10	11
s	10	1	0	9	3	3
e	3	0	6	9	0	7
a	9	6	4	5	10	4
r	5	12	7	0	0	7
t	9	3	0	12	8	1

Swap 'a'↔'r'; 164

	-	r	e	s	a	t
-	0	10	8	4	3	11
r	5	0	7	12	0	7
e	3	0	6	0	9	7
s	10	3	0	1	9	3
a	9	10	4	6	5	4
t	9	8	0	3	12	1

Swap 'a'↔'t'; 156

	-	t	e	s	r	a
-	0	11	8	4	10	3
t	9	1	0	3	8	12
e	3	7	6	0	0	9
s	10	3	0	1	3	9
r	5	7	7	12	0	0
a	9	4	4	6	10	5

Swap 'e'↔'s'; 150

	-	a	s	e	r	t
-	0	3	4	8	10	11
a	9	5	6	4	10	4
s	10	9	1	0	3	3
e	3	9	0	6	0	7
r	5	0	12	7	0	7
t	9	12	3	0	8	1

Swap 'e'↔'r'; 106∗

	-	a	r	s	e	t
-	0	3	10	4	8	11
a	9	5	10	6	4	4
r	5	0	0	12	7	7
s	10	9	3	1	0	3
e	3	9	0	0	6	7
t	9	12	8	3	0	1

Swap 'e'↔'t'; 118

	-	a	t	s	r	e
-	0	3	11	4	10	8
a	9	5	4	6	10	4
t	9	12	1	3	8	0
s	10	9	3	1	3	0
r	5	0	7	12	0	7
e	3	9	7	0	0	6

Swap 's'↔'r'; 178

	-	a	e	r	s	t
-	0	3	8	10	4	11
a	9	5	4	10	6	4
e	3	9	6	0	0	7
r	5	0	7	0	12	7
s	10	9	0	3	1	3
t	9	12	0	8	3	1

Swap 's'↔'t'; 126

	-	a	e	t	r	s
-	0	3	8	11	10	4
a	9	5	4	4	10	6
e	3	9	6	7	0	0
t	9	12	0	1	8	3
r	5	0	7	7	0	12
s	10	9	0	3	3	1

Swap 'r'↔'t'; 136

	-	a	e	s	t	r
-	0	3	8	4	11	10
a	9	5	4	6	4	10
e	3	9	6	0	7	0
s	10	9	0	1	3	3
t	9	12	0	3	1	8
r	5	0	7	12	7	0

Figure 36: Hill-Climbing, Step 2: Go from aesrt, 144 to arset, 106, marked "∗".

Swap 'a'↔'r'; 88∗

	-	r	a	s	e	t
-	0	10	3	4	8	11
r	5	0	0	12	7	7
a	9	10	5	6	4	4
s	10	3	9	1	0	3
e	3	0	9	0	6	7
t	9	8	12	3	0	1

Swap 'a'↔'s'; 130

	-	s	r	a	e	t
-	0	4	10	3	8	11
s	10	1	3	9	0	3
r	5	12	0	0	7	7
a	9	6	10	5	4	4
e	3	0	0	9	6	7
t	9	3	8	12	0	1

Swap 'a'↔'e'; 138

	-	e	r	s	a	t
-	0	8	10	4	3	11
e	3	6	0	0	9	7
r	5	7	0	12	0	7
s	10	0	3	1	9	3
a	9	4	10	6	5	4
t	9	0	8	3	12	1

Swap 'a'↔'t'; 100

	-	t	r	s	e	a
-	0	11	10	4	8	3
t	9	1	8	3	0	12
r	5	7	0	12	7	0
s	10	3	3	1	0	9
e	3	7	0	0	6	9
a	9	4	10	6	4	5

Swap 'r'↔'s'; 150

	-	a	s	r	e	t
-	0	3	4	10	8	11
a	9	5	6	10	4	4
s	10	9	1	3	0	3
r	5	0	12	0	7	7
e	3	9	0	0	6	7
t	9	12	3	8	0	1

Swap 'r'↔'e'; 144

	-	a	e	s	r	t
-	0	3	8	4	10	11
a	9	5	4	6	10	4
e	3	9	6	0	0	7
s	10	9	0	1	3	3
r	5	0	7	12	0	7
t	9	12	0	3	8	1

Swap 'r'↔'t'; 128

	-	a	t	s	e	r
-	0	3	11	4	8	10
a	9	5	4	6	4	10
t	9	12	1	3	0	8
s	10	9	3	1	0	3
e	3	9	7	0	6	0
r	5	0	7	12	7	0

Swap 's'↔'e'; 148

	-	a	r	e	s	t
-	0	3	10	8	4	11
a	9	5	10	4	6	4
r	5	0	0	7	12	7
e	3	9	0	6	0	7
s	10	9	3	0	1	3
t	9	12	8	0	3	1

Swap 's'↔'t'; 140

	-	a	r	t	e	s
-	0	3	10	11	8	4
a	9	5	10	4	4	6
r	5	0	0	7	7	12
t	9	12	8	1	0	3
e	3	9	0	7	6	0
s	10	9	3	3	0	1

Swap 'e'↔'t'; 144

	-	a	r	s	t	e
-	0	3	10	4	11	8
a	9	5	10	6	4	4
r	5	0	0	12	7	7
s	10	9	3	1	3	0
t	9	12	8	3	1	0
e	3	9	0	0	7	6

Figure 37: Hill-Climbing, Step 3: Go from arset, 106 to table raset, 88, marked "∗".

Roadmap

	Section 3.7	Encipher plaintext ⇒ ***scramble*** frequencies.
	Section 3.7	Decipher ciphertext ⇒ ***unscramble*** frequencies.
	Section 4.2	(Hope) Unscramble frequencies ⇒ decipher ciphertext
	Section 4.3	Unscramble = Bring "close" to ***intrinsic*** frequencies Approximate ***intrinsic*** frequencies with ***training text*** Assume ciphertext is medium to large so that unscrambled frequencies resemble intrinsic frequencies
	Section 4.4	Use the L^1 ***distance*** to measure "closeness"; ignore labels.
	Section 5.1	Sorting unigram frequencies does not work, but "almost" does. (Hope) When the frequency table for ciphertext matches the table for training text, read the encryption key off of the labels
	Section 5.2	"Sort bigram frequencies" is problematic.
	Section 5.3	***Exhaustive*** search is too slow; therefore, need an incomplete search.
	Section 6.1	(Idea) Hill-Climbing: pick the best "neighbor"
	Section 6.2	Why swaps are so important, e.g. used to generate "neighbors"
	Section 6.3	The effect of swaps on bigram tables
	Section 7.1	Hill-climbing and unscrambling frequencies appear to work
(→)	Section 7.2	Hill-climbing in the abstract
	Section 7.3	Improvement/alternative to hill-climbing

Swap 'r'↔'a'; 106

	-	a	r	s	e	t
-	0	3	10	4	8	11
a	9	5	10	6	4	4
r	5	0	0	12	7	7
s	10	9	3	1	0	3
e	3	9	0	0	6	7
t	9	12	8	3	0	1

Swap 'r'↔'s'; 132

	-	s	a	r	e	t
-	0	4	3	10	8	11
s	10	1	9	3	0	3
a	9	6	5	10	4	4
r	5	12	0	0	7	7
e	3	0	9	0	6	7
t	9	3	12	8	0	1

Swap 'r'↔'e'; 138

	-	e	a	s	r	t
-	0	8	3	4	10	11
e	3	6	9	0	0	7
a	9	4	5	6	10	4
s	10	0	9	1	3	3
r	5	7	0	12	0	7
t	9	0	12	3	8	1

Swap 'r'↔'t'; 152

	-	t	a	s	e	r
-	0	11	3	4	8	10
t	9	1	12	3	0	8
a	9	4	5	6	4	10
s	10	3	9	1	0	3
e	3	7	9	0	6	0
r	5	7	0	12	7	0

Swap 'a'↔'s'; 122

	-	r	s	a	e	t
-	0	10	4	3	8	11
r	5	0	12	0	7	7
s	10	3	1	9	0	3
a	9	10	6	5	4	4
e	3	0	0	9	6	7
t	9	8	3	12	0	1

Swap 'a'↔'e'; 164

	-	r	e	s	a	t
-	0	10	8	4	3	11
r	5	0	7	12	0	7
e	3	0	6	0	9	7
s	10	3	0	1	9	3
a	9	10	4	6	5	4
t	9	8	0	3	12	1

Swap 'a'↔'t'; 120

	-	r	t	s	e	a
-	0	10	11	4	8	3
r	5	0	7	12	7	0
t	9	8	1	3	0	12
s	10	3	3	1	0	9
e	3	0	7	0	6	9
a	9	10	4	6	4	5

Swap 's'↔'e'; 124

	-	r	a	e	s	t
-	0	10	3	8	4	11
r	5	0	0	7	12	7
a	9	10	5	4	6	4
e	3	0	9	6	0	7
s	10	3	9	0	1	3
t	9	8	12	0	3	1

Swap 's'↔'t'; 108

	-	r	a	t	e	s
-	0	10	3	11	8	4
r	5	0	0	7	7	12
a	9	10	5	4	4	6
t	9	8	12	1	0	3
e	3	0	9	7	6	0
s	10	3	9	3	0	1

Swap 'e'↔'t'; 136

	-	r	a	s	t	e
-	0	10	3	4	11	8
r	5	0	0	12	7	7
a	9	10	5	6	4	4
s	10	3	9	1	3	0
t	9	8	12	3	1	0
e	3	0	9	0	7	6

Figure 38: Hill-Climbing, Step 4: Stay at `raset`, 88 — neighbors are worse.

7.2 Hill-Climbing in the Abstract

Let us continue to consider in the abstract how to solve optimization problems. Suppose you're placed on hilly terrain in a vehicle with tons of controls (e.g. 26 of them) for movement but only sensors for your current position (so you can tell if you're going up or down). Further suppose you wish to ascend to the highest peak. The difficulties you face are the large number of choices and a lack of knowledge of how the choices (controls) interact. What do you do?

Hill-climbing assumes there is a single, smooth hill. In this case, your best choice is to take the steepest step.

But what if there are lots of bumps — lots of big hills, each with medium hills, each with small hills, etc.? Then hill-climbing is likely to get stuck on a local max, a small hill on a medium hill that is not the biggest hill. The problem with hill-climbing is that being optimistic gets you trapped locally. This is because the "best" local choice is always taken, even if to get to the best solution requires taking some steps down. Therefore, one must occasionally take steps "backwards", which is what *simulated annealing does*.

7.3 Simulated Annealing

(Never mind the name, which is for historical reasons that many don't find helpful.) The key idea behind ***simulated annealing*** is to randomly take steps "backwards" every now and then. Initially, backwards steps are taken relatively frequently, but as time goes on, they are taken less frequently.

The idea is as follows. First find the biggest hill. So go up and down and all around. If it is the biggest/steepest, then chances are you'll end up near it, rather than some medium or small hill. However, on the biggest hill, there can still be local maxima that would trap hill-climbing. Therefore, on the biggest hill, you need to explore to find the true peak, rather than a distracting medium hill. Therefore, still take backwards steps occasionally, but since you want to stay on the biggest hill overall, don't go backwards too often. Once you're on the true peak, there are still perhaps small hills, and then tiny hills, and then eensy-weensy hills. Therefore, still go backwards from time to time, but with progressively rarer frequency.

If you're not clear why this should work, that is not surprising. There are reasonably good justifications for it, but they're a bit complicated.

If you're not clear on what rate to use to reduce the probability of backwards steps, that is because the answer is not known. Generally, people just try something *ad hoc*, which is just a fancy way of saying, they make something up. If it works, great; otherwise, they try changing the rate in a different way.